

ENSC-201X-N°YYY



THÈSE DE DOCTORAT DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Présentée par

Monsieur Robert Künnemann

Pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Domaine :

INFORMATIQUE

Sujet de la thèse :

FOUNDATIONS FOR ANALYZING SECURITY APIS IN THE SYMBOLIC AND COMPUTATIONAL MODEL

Thèse présentée et soutenue à Cachan le 07/01/2014 devant le jury composé de :

Ralf Küsters	Professeur	Rapporteur
Matteo Maffei	Professeur	Rapporteur
Hubert Comon-Lundh	Professeur	Examinateur
Joshua Guttman	Professeur	Examinateur
Thomas Jensen	Directeur de recherche	Examinateur
David Pointcheval	Directeur de recherche	Examinateur
Steve Kremer	Directeur de recherche	Directeur de thèse
Graham Steel	Chargé de recherche	Directeur de thèse

Laboratoire Spécification et Vérification
ENS de Cachan, UMR 8643 du CNRS
61, avenue du Président Wilson
94235 CACHAN Cedex, France

FOUNDATIONS FOR ANALYZING SECURITY APIS IN THE SYMBOLIC AND COMPUTATIONAL MODEL

ROBERT KÜNNEMANN

October 2013

Robert Künnemann: *Foundations for analyzing security APIs in the symbolic and computational model,* © October 2013

A map is not the territory it represents, but, if correct, it has a similar structure to the territory, which accounts for its usefulness.

— Alfred Korzybski

Für meine Eltern.

ABSTRACT

Security critical applications often store keys on dedicated Hardware Security Modules (HSMs) or key-management servers to separate sensitive cryptographic operations from more vulnerable parts of the network. Access to such devices is given to protocol parties by the means of *Security APIs*, e.g., the RSA PKCS#11 standard, IBM's CCA and the Trusted Platform Module (TPM) API, all of which protect keys by providing an API that allows to address keys only indirectly.

This thesis has two parts. The first part deals with formal methods that allow for the identification of secure configurations in which Security APIs improve the security of existing protocols, e. g., in scenarios where parties can be corrupted. A promising paradigm is to regard the Security API as a participant in a protocol and then use traditional protocol analysis techniques. But, in contrast to network protocols, Security APIs often rely on the state of an internal database. When it comes to an analysis of an unbounded number of keys, this is the reason why current tools for protocol analysis do not work well. We make a case for the use of Multiset Rewriting (MSR) as the back-end for verification and propose a new process calculus, which is a variant of the applied pi calculus with constructs for manipulation of a global state. We show that this language can be translated to MSR rules while preserving all security properties expressible in a dedicated first-order logic for security properties. The translation has been implemented in a prototype tool which uses the tamarin prover as a back-end. We apply the tool to several case studies among which a simplified fragment of PKCS#11, the Yubikey security token, and a contract signing protocol.

The second part of this thesis aims at identifying security properties that (*a*) can be established independent of the protocol, (*b*) allow to catch flaws on the cryptographic level, and (*c*) facilitate the analysis of protocols using the Security API. We adapt the more general approach to API security of Kremer et al. to a framework that allows for composition in form of a universally composable key-management functionality. The novelty, compared to other definitions, is that this functionality is parametric in the operations the Security API allows, which is only possible due to universal composability. A Security API is secure if it implements correctly both key-management (according to our functionality) and all operations that depend on keys (with respect to the functionalities defining those operations). We present an implementation which is defined with respect to arbitrary functionalities (for the operations that are not concerned with key-management), and hence represents a general design pattern for Security APIs.

RÉSUMÉ

Dans une infrastructure logicielle, les systèmes critiques ont souvent besoin de garder des clés cryptographiques sur des Hardware Security Modules (HSMs) ou des serveurs consacrés à la gestion de clés. Ils ont pour but de séparer les opérations cryptographiques, très critiques, du reste du réseau, qui est plus vulnérable. L'accès aux clés est fourni au travers des *APIs de sécurité*, comme par exemple le standard PKCS#11 de RSA, CCA d'IBM, ou encore l'API du Trusted Platform Module, qui ne permettent qu'un accès indirect aux clés.

Cette thèse est composée de deux parties. La première introduit des méthodes formelles qui ont pour but l'identification des configurations dans lesquelles les APIs de sécurité peuvent améliorer le niveau de sûreté des protocoles existants, par exemple en cas de compromission d'un participant. Un paradigme prometteur considère les APIs de sécurité comme des participants d'un protocole afin d'étendre l'emploi des méthodes traditionnelles d'analyse de protocole à ces interfaces. À l'opposé des protocoles réseau, les APIs de sécurité utilisent souvent une base de données interne. Ces outils traditionnels d'analyse ne sont adaptés que pour le cas où le nombre des clés est borné a priori.

Nous exposons également des arguments pour l'utilisation de la réécriture de multi-ensembles MSR (Multiset Rewriting), lors de la vérification. De plus, nous proposons un langage dérivant du pi-calcul appliqué possédant des opérateurs qui permettent la manipulation d'un état explicite. Ce langage se traduit en règles MSR en préservant des propriétés de sécurité qui s'expriment dans une logique de premier ordre. La traduction a été implémentée sous forme d'un prototype produisant des règles spécifiquement adapté au prouveur tamarin. Plusieurs études de cas, dont un fragment de PKCS#11, le jeton d'authentification Yubikey, et un protocole de signature de contrat optimiste ont été effectuées.

Le deuxième partie de la présente thèse a pour but l'identification des propriétés de sécurité qui *a*) peuvent être établies indépendamment du protocole *b*) permettent de trouver des failles au niveau de la cryptographie *c*) facilitent l'analyse des protocoles utilisant cette API de sécurité.

Nous adoptons ici l'approche plus générale de Kremer et al. dans un cadre qui permet la composition universelle, à travers une fonctionnalité de gestion de clés. La nouveauté de ce genre de définition est qu'elle dépend des opérations mises à disposition par l'API. Ceci est seulement possible grâce à la composition universelle. Une API de sécurité n'est sûre que si elle réalise correctement la gestion des clés (selon la fonctionnalité présentée dans ce travail) et les opérations

utilisant les clés (selon les fonctionalités qui les définissent). Pour finir, nous présentons aussi une implémentation de gestion de clés définie par rapport à des opérations arbitraires utilisant des clés non concernées par la gestion des clés. Cette réalisation représente ainsi un modèle générique pour le design des APIs de sécurité.

PUBLICATIONS

Some ideas have appeared previously in the following publications:

[1] Steve Kremer, Robert Künnemann, and Graham Steel. 'Universally Composable Key-Management.' In: *European Symposium on Research in Computer Security*. Springer, 2013, pp. 327–344.

[2] Robert Künnemann and Graham Steel. 'YubiSecure? Formal Security Analysis Results for the Yubikey and YubiHSM.' In: *Workshop on Security and Trust Management*. Springer, 2012, pp. 257–272.

ACKNOWLEDGMENTS

First things first, I would like to thank my supervisors, Steve Kremer and Graham Steel for their patient guidance, as well as the time and effort they have invested in me. I always found an open ear and valuable advice in scientific questions, as well as question outside this matter.

While preparing my thesis I was lucky to work in a relaxing and stimulating environment, which I found in the PROSECCO group. In particular Iro Bartzia and Alfredo Pironti, whom I shared office with, have been great partners in discussion and in crime. Many laughs were had. I would also like to thank Benjamin (*notre benjamin*), who joined us for an internship in my last year and helped me with the résumé in French.

Furthermore, I wish to thank my friends Esfandiar Mohammadi and Guillaume Scerri, as well as the other members of the Laboratoire Spécification et Vérification for their help and for the conversations we had. I am likewise indebted to Dominique Unruh for his discussion and advice about all things UC.

A very special thank you goes to Milla, for all the love and patience, encouragement and distraction she had for me. Well, and for reading the manuscript, of course.

Most importantly, none of this would have been possible without the love and support of my parents and my brother, who read the manuscript almost in full and gave excellent critique.

CONTENTS

Introduction 1

1 INTRODUCTION 3
- 1.1 Cryptographic security APIs 6
- 1.2 security APIs and network protocols 10
- 1.3 security APIs in context of arbitrary protocols 11

I ANALYSIS OF STATEFUL PROTOCOLS IN THE SYMBOLIC MODEL 15

2 PRELIMINARIES 17
- 2.1 Terms and equational theories 17
- 2.2 Facts 18
- 2.3 Substitutions 18
- 2.4 Sets, sequences and multisets 18

3 A SURVEY ON EXISTING METHODS 21
- 3.1 Running Examples 22
- 3.2 StatVerif 24
  - 3.2.1 Running Example 4: Left-or-right encryption in StatVerif 25
  - 3.2.2 Running Example 5: Wrap/Dec in StatVerif 25
- 3.3 Abstraction by set-membership 26
  - 3.3.1 Running Example 5: Wrap/Dec using abstraction by set-membership 28
  - 3.3.2 Running Example 4: Left-or-right encryption 30
- 3.4 Why Horn clause approaches are unsuitable 31
- 3.5 Other Approaches 35
- 3.6 The tamarin-prover 35
  - 3.6.1 Labelled multiset rewriting 36
  - 3.6.2 Adversarial deduction 39
  - 3.6.3 Security Properties 39

4 ANALYSIS OF THE YUBIKEY PROTOCOL AND THE YUBIHSM 43
- 4.1 Yubikey and Yubikey Authentication Protocol 44
  - 4.1.1 About the Yubikey Authentication Protocol 44
  - 4.1.2 Formal Analysis (Uncompromised Server) 47
- 4.2 The YubiHSM 50
  - 4.2.1 About the YubiHSM 50
  - 4.2.2 Two Attacks on the Implementation of Authenticated Encryption 51
  - 4.2.3 Analysis in the Case of Server Compromise 53
- 4.3 Evaluation 55
  - 4.3.1 Positive Results 55
  - 4.3.2 Negative Results 56

4.3.3 Possible changes to the YubiHSM 57
4.3.4 Methodology 58
4.3.5 Future work 59
5 A PROCESS CALCULUS WITH STATE 61
5.1 Related work 62
5.2 A cryptographic pi calculus with explicit state 62
5.2.1 Syntax and informal semantics 63
5.2.2 Semantics 66
5.2.3 Discussion 69
5.3 A translation from processes to multiset rewrite rules 71
5.3.1 Translation of processes 71
5.3.2 Translation of trace formulas 77
5.3.3 Discussion 78
5.4 Correctness of the translation 85
5.4.1 Lemmas about message deduction 89
5.4.2 Inclusion I 89
5.4.3 Inclusion II 93
5.5 Case studies 94
5.5.1 Security API à la PKCS#11 95
5.5.2 Needham-Schoeder-Lowe 96
5.5.3 Yubikey 96
5.5.4 The GJM contract signing protocol 97
5.5.5 Further Case Studies 97

II WHEN IS A SECURITY API "SECURE"? 99
6 WHEN IS A SECURITY API SECURE? 101
6.1 Criteria for persuasive definitions 101
6.2 Characteristics of a "secure" security API 102
6.3 Composability 102
6.4 Overview 103
6.5 Related work 103
7 INTRODUCTION TO GNUC 105
7.1 Preliminaries 105
7.2 Machines and interaction 106
7.3 Defining security via ideal functionalities 107
8 KEY-MANAGEMENT FUNCTIONALITY AND REFER. IMPLEMENTATION 111
8.1 Design Rationals 112
8.1.1 Policies 112
8.1.2 Sharing Secrets 113
8.1.3 Secure Setup 113
8.1.4 Operations required 114
8.2 Key-usage (ku) functionalities 114
8.2.1 Credentials 115
8.2.2 Key-manageable functionalities 115
8.3 Policies 116

8.4 $\mathcal{F}_{KM}$ and the reference implementation 117
8.4.1 Structure and Network setup 118
8.4.2 Setup phase 120
8.4.3 Executing commands in $\mathcal{C}^{priv}$ 120
8.4.4 Creating keys 121
8.4.5 Wrapping and Unwrapping 122
8.4.6 Changing attributes of keys 124
8.4.7 Corruption 124
8.4.8 Public key operations 125
8.4.9 Formal definition of $\mathcal{F}_{KM}$ 126
8.4.10 Formal definition of the security token network 127
9 ANALYSIS OF THE KEY-MANAGEMENT FUNCTIONALITY 129
9.1 Properties 129
9.2 Limitations 131
9.3 Discussion 133
10 PROOF OF EMULATION. 137
11 A SIMPLE CASE STUDY 141
11.1 Realizing $\mathcal{F}_{KM}$ for a static key-hierarchy 141
11.2 An example implementation of the authenticated channel functionality 143

Conclusion 149

12 CONCLUSION AND PERSPECTIVES 151
12.1 Analysis of protocols using security APIs 151
12.2 Analysis of security APIs 154

Appendix 159

A LISTINGS FOR PART I 161
A.1 Listings for Chapter 3 161
A.2 Listings for Chapter 4 166
A.3 Listings for Chapter 5 183
B PROOFS FOR PART I 197
B.1 Correctness of tamarin's solution procedure for translated rules 197
B.2 Proofs for Section 5.4 199
B.2.1 Proofs for Section 5.4.2 204
B.2.2 Proofs for Section 5.4.3 220
C INITIALISATION AND SETUP IN $\mathcal{F}_{KM}$ 241
C.1 Initialisation phase 241
C.2 Handling of the setup phase in $\mathcal{F}_{KM}$ and ST_i 241
C.3 Setup assumptions for the implementation 242
D THE COMPLETE PROOF OF EMULATION 243

BIBLIOGRAPHY 265

Figure 1 Policy graph: strict role separation. 9
Figure 2 Policy graph: no separation, but still avoiding the attack in Example 1. 9
Figure 3 Syntax of the StatVerif calculus. 24
Figure 4 Example 4 in StatVerif. The deconstructors car and cdr extract the first respectively the last element of a pair. 25
Figure 5 Modelling for Example 5 in AVISPA Intermediate Format (AIF). The full code can be found in Listing 16 in Section A.1, p. 164. 29
Figure 6 Modelling for Example 4 in AIF. The full code can be found in Listing 14 in Section A.1, p. 162. 31
Figure 7 Structure of the One-time Password (OTP) (session: 19, token: 16). 46
Figure 8 Advanced Encryption Standard (AES) in counter mode (simplified). 52
Figure 9 Syntax. 63
Figure 10 Deduction rules. 66
Figure 11 Operational semantics. 68
Figure 12 Definition of $[\![P, p, \tilde{x}]\!]$. 75
Figure 13 Example trace for translation of $!P_{new}$. 77
Figure 14 Definition of $[\![P, p, \tilde{x}]\!]_{=p_t}$ where $\{\cdot\}_{a \stackrel{?}{=} b} = \{\,\cdot\,\}$ if $a = b$ and $\emptyset$ otherwise. 91
Figure 15 Distributed security tokens in the network (left-hand side) and idealized functionality $\mathcal{F}_{KM}$ in the same network (right-hand side). 118
Figure 16 Visualisation of Case 2. 217
Figure 17 Visualisation of Case 2a. 218
Figure 18 Visualisation of Case 2b. 218
Figure 19 Visualisation of Case 1. 219
Figure 20 Visualisation of Case 2. 219
Figure 21 Visualisation of Case 1. 239

Figure 22 Visualisation of Case 2. 239

LIST OF TABLES

Table 1 Case studies. 95

LISTINGS

Listing 1 $\mathcal{F}_{\text{ach}}$ with session parameters $\langle P_{pid}, Q_{pid}, label\rangle$. Note that in this example, every step can only be executed once. 107
Listing 2 Executing command C on a handle h with data m ($\mathcal{F}_{\text{KM}}$ above, ST_i below). 121
Listing 3 Creating keys of type $\mathcal{F}$, and attribute a ($\mathcal{F}_{\text{KM}}$ above, ST_i below). 121
Listing 4 Wrapping key h_2 under key h_1 with additional information *id* ($\mathcal{F}_{\text{KM}}$ above, ST_i below). 122
Listing 5 Unwrapping w created with attribute a_2, F_2 and *id* using the key h_1. $\exists!x.p(x)$ holds if there exists exactly one x such that $p(x)$ holds ($\mathcal{F}_{\text{KM}}$ above, ST_i below). 123
Listing 6 Changing the attribute of h to a' ($\mathcal{F}_{\text{KM}}$ above, ST_i below). 124
Listing 7 Corruption procedure used in steps `corrupt` and `wrap`. 125
Listing 8 Corrupting h ($\mathcal{F}_{\text{KM}}$ above, ST_i below). 125
Listing 9 Computing the public commands C using the inputs *public* and m ($\mathcal{F}_{\text{KM}}$, note that ST_i does not implement this step). 125
Listing 10 A signature functionality $\mathcal{F}_{\text{SIG}}$. 142
Listing 11 $\mathcal{F}_{ach}$ with session parameters $\langle P_{pid}, Q_{pid}, label\rangle$. Note that in this example, every step can only be executed once. 143
Listing 12 The modelling of Example 4 in StatVerif by Arapinis et al. [5]. 161

Listing 13 An attempt to express Example 5 in StatVerif. Note that the key store is modelled as single state cell, which refers to a list of keys. This list grows in size for every newly generated key. ProVerif did not terminate on the horn clauses produced by StatVerif's translation procedure in the experiments we conducted. 161
Listing 14 A modelling of Example 4 in AIF. 162
Listing 15 Another modelling of Example 4 in AIF. 163
Listing 16 A modelling of Example 5 in AIF. 164
Listing 17 Modelling of the Yubikey authentication protocol in tamarin's MSR calculus. The counter is modelled as a term constituting a multiset of constants, where the cardinality of the multiset equals the size of the counter. 166
Listing 18 Modelling of the Yubikey authentication protocol in conjunction with the YubiHSM, formalised in tamarin's MSR calculus. The counter is modelled as a term constituting a multiset of constants, where the cardinality of the multiset equals the size of the counter. 168
Listing 19 Modelling of the Yubikey authentication protocol formalised in tamarin's MSR calculus. The counter is modelled using a "successor of" constructor. The permanent fact !IsSmaller can be produced by the adversary to proof that one counter value is smaller than another. We need to enforce transitivity in order to be able to proof our claims – that means, we require (using an axiom) each trace to contain IsSmaller(a, c), should IsSmaller(a, b) and !IsSmaller(a, c) be present, for $a, b, c \in \mathcal{M}$. !Succ models a functional relation: If !Succ(a, b), then the adversary was able to show that b is the successor of a. The relation modelled by !Smaller is not functional: If !Smaller(a, b), then the adversary was able to show that a is smaller than b. The Theory() action is used to enforce that this relation (to the extend it is needed in this trace) has to be build up before running the first protocol actions. 173
Listing 20 Modelling of the Yubikey authentication protocol in conjunction with the YubiHSM, modelling the "smaller-than" relation as described in Listing 19. 177
Listing 21 The running example from Chapter 5. 183

Listing 22 Security à la Public-Key Cryptography Standards, Number 11 (PKCS#11). 184
Listing 23 Needham-Schoeder-Lowe protocol. 186
Listing 24 Yubikey protocol. 187
Listing 25 Garay, Jakobsson and MacKenzie's optimistic contract signing protocol. 190
Listing 26 Key-server example from [74]. 192
Listing 27 Key-server example from [74], modelled under the use of insert and lookup. 194
Listing 28 Left-right example from [5], described in Example 4 on p.22. 195
Listing 29 Initilisation ($\mathcal{F}_{KM}$). 241
Listing 30 Initilisation (ST_i). 241
Listing 31 The setup phase: sharing keys ($\mathcal{F}_{KM}$). 241
Listing 32 The setup phase: sharing keys (ST_i). 241
Listing 33 The setup phase: terminating the setup phase ($\mathcal{F}_{KM}$). 242
Listing 34 The setup phase: terminating the setup phase (ST_i). 242
Listing 35 procedure for corrupting a credential c. 245
Listing 36 Procedure for corrupting a credential c. 252

ACRONYMS

AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AIF	AVISPA Intermediate Format
API	Application Programming Interface
ATM	Automated teller machine
AVISPA	Automated Validation of Internet Security Protocols and Applications
CBC	Cipher-block chaining mode of operation
CCA	IBM Common Cryptographic Architecture
CCM	Counter mode of operation with CBC-MAC
CC	Common Criteria for Information Technology Security Evaluation
CPSA	Cryptographic Protocol Shapes Analyzer

CRC	Cyclic Redundancy Check
EMV	Europay, MasterCard and Visa
GNUC	GNUC is Not UC
HMAC	keyed-hash Message Authentication Code
HOTP	keyed-hash Message Authentication Code (HMAC)-based One-time Password Algorithm
HSM	Hardware Security Module
IV	Initialisation Vector
MAC	Message Authentication Code
MSR	Multiset Rewriting
OATH	Initiative For Open Authentication
OTP	One-time Password
PIN	Personal Identification Number
PKCS#11	Public-Key Cryptography Standards, Number 11
PPT	probabilistic polynomial-time
RFC	Request For Comments
RFID	Radio-frequency identification
RSA	RSA Security, Inc.
SIM	Subscriber Identity Module
SIV	Synthetic Initialization Vector
TLS	Transport Layer Security
TPM	Trusted Platform Module
UC	Universal Composability
USB	Universal Serial Bus

INTRODUCTION

1 INTRODUCTION

The more complex a system, the more difficult it is to verify that it is secure. Computers used in a network protocol are often used for other tasks, such as reading emails and surfing on the Internet, too. The hardware and software components necessary to perform all those tasks not only add to the complexity of analysing the system, but also provide an increased attack surface for compromising parts of the system that are important for the security of the protocol.

The computer at your bank that verifies your Personal Identification Number (PIN) should not be used for reading emails, for example. It should in fact only do what is necessary to check your PIN. It should be behind locked doors, so it can only be accessed by authorized personnel – this way, the interface between a potential outside attacker and the computer is reduced. If the computer that checks PINs is separate from the computer that provides you with a keypad and transfers your PIN, and both are separate from the computer emails are read on, then there is actually hope that we can analyse those systems in isolation. Furthermore, if, e. g., the computer that checks PINs can be shown to never accept an incorrect PIN, a certain amount of security can be guaranteed for the system as a whole, no matter if the computer that emails are read on can be trusted or not.

There are companies that sell devices dedicated to performing PIN verification. Those devices can only be accessed using a clearly defined Application Programming Interface (API). Furthermore, there are a number of hardware measures in place that are supposed to prevent any other way of accessing the device, or more precisely, the information contained on this device. Such measures include, for example, strong enclosures that make it difficult to obtain physical access to the main board, but also tamper detection/response circuitry that overwrites sensitive cryptographic information in case an intrusion of the enclosure is detected [43].

The literature often uses the term *security token* for such a device. While most people are aware what kind of device a PC is, and many of them are aware that there are attacks, for example viruses, worms, trojan horses etc. on PCs, to most people, the term "security token" means nothing. Moreover, the idea of a tamper-resistant device with cryptographic functions appears to be a very specific solution for a very specific problem; something that is, for example, part of the operations of a financial institution. Many are not aware of the ubiquity of security tokens in our everyday lives. Not only is it the case that security tokens are indispensable for money transfers of all kinds, be

it online banking, the use of ATMs (cash machines) or credit card terminals. Also e-commerce, whose security hinges on the security of the channel between client and merchant via the Transport Layer Security (TLS) protocol, relies on devices dedicated for cryptographic operations. Many large vendors employ such devices to establish TLS connections. Even smaller vendors who cannot afford such devices rely on the integrity of the keys stored with the certification authorities, for the use of the TLS protocol, which themselves (should) store their signature keys on security tokens.

Security tokens have their part in most aspects of our lives that have to do with money, but they are with us in a quite literal sense, too: Many security tokens are integrated circuits we carry in our pockets, so-called smart cards. In some cities, they are used as tickets for public transport, as part of a public bicycle renting system or for public libraries. They are frequently employed as part of the public health system, too. Of course smart cards also play a role in our financial activities: We use them for payments and money withdrawal. Eurosmart reports that in 2012 alone, 6.970 billion smart cards where shipped in telecommunication, financial services, government, transport, pay TV and other areas [42]. Telecommunication makes up for 5.1 billion of this figure, the reason is the Subscriber Identity Module. It is used with practically every mobile phone subscription and securely stores the information necessary to identify and authenticate subscribers on mobile telephony devices such as mobile phones and computers. There are an estimated 6.8 billion active mobile phone subscriptions in 2013, hence about the same number of Subscriber Identity Module (SIM) cards *in use* [52]. Considering that there are around 7.1 billion people in world (as of 2013), this number is surprisingly high. The number of PCs seems modest in comparison: At the beginning of 2013, Deloitte Canada estimated that in 2013, there will be 1.6 billion PCs in use [75]. These numbers bear witness of the impact that the development and use of security tokens have had on modern life. The analysis of the secure operating of those device is thus of utmost importance.

In this thesis we will use the term *security API* for the interface that devices like security tokens provide to the outside. The concept of a security API entails the following characteristics.

A. They provide (often limited) access to sensitive resources to the outside, which is untrusted.

B. The implementation of the security API is trusted.

C. Access is only possible via the security API.

A security API is often implemented using an external piece of hardware with a tamper-resistant enclosure, for example in form of a security token. But the concept extends to other domains, too. A

virtual machine running in a secure environment or an API accessible to some untrusted code that runs in a sandbox shares the same characteristics, as well as JavaScript code embedded in a website (untrusted) accesses via the web browser's implementation of JavaScript. Consider the following examples:

- Smart cards, pocket-sized cards with an integrated circuit, for example the SIM used in mobile phones and the EMV standard used for debit cards as well as credit cards provide a security API to the mobile phone in use.

- PKCS#11 [82] defines a platform-independent API to security tokens, for example smart cards, but also HSMs. HSMs are physical computing devices that can be attached directly to a server and communicated via Ethernet, Universal Serial Bus (USB) or other services, providing logical and physical protection for sensitive information and algorithms.

- The IBM Common Cryptographic Architecture (CCA, [25]) is a security API supported by the IBM 4764 and IBM 4758 Cryptographic Coprocessors. Since CCA provides many functions that are of special interest in the finance industry, it is widely used in the ATM network, for example, to encrypt, decrypt or verify PINs. Both PKCS#11 and CCA are commonly used interfaces to HSMs.

- Small security tokens, for example the Yubikey manufactured by the Swedish company Yubico provide a security API. They can provide one-time passwords used for secure authentication.

- The Trusted Platform Module (TPM, [87]) is the specification of a secure cryptographic processor and hence defines a security API that can store keys. A TPM chip is built into most PCs and notebooks that are sold today. It can be used to provide full disk encryption, password protection and to assure the integrity of a platform, e. g., the operating system running with a set of applications.

- Key-management protocols, for instance the OpenPGP HTTP Keyserver Protocol [91] can also be seen as security APIs. The key-server in such protocols also provides a limited interface to the outside. In particular, the computer running the key-server is likely to be protected against physical access. This example shows that a security API is not necessarily the interface to a security token.

- The Google maps API provides an interface between a set of servers trusted by Google and the Internet, which is untrusted. This shows that a security API is not necessarily a cryptographic API.

Although security APIs mainly describe the interface between the trusted implementation and the outside, we will sometimes use this term to refer to the trusted implementation, which maybe a piece of hardware, a library or a network of computers.

1.1 CRYPTOGRAPHIC SECURITY APIS

Many network protocols rely on cryptographic operations. Cryptographic operations, in turn, rely on secrets which *a*) should not be extractable by the adversary, and *b*) often need to be honestly generated. Let us assume for the rest of this thesis that the implementation of the security API has a higher level of confidence than the rest of the system. This assumption is based on the fact that it is easier to verify than the overall system, since it is less complex. Furthermore, it can, and often is, designed with security as a main goal, something that is hardly true for an all-purpose computer. Under this assumption, it seems rational to put the cryptographic secrets on the security API, and forbid all direct access to them.

This has two consequences which further outline how a cryptographic security API operates. First, all algorithms that depend on cryptographic secrets need to be implemented inside the security API. Second, all other parts of the system shall stay outside the device (as it should be as small as possible). Assuming such a system — a network of all-purpose computers that run some protocol together and access one or many security APIs to compute cryptographic values used in the protocol — there are four layers where things could go wrong:

LAYER 1 The overall system, including the security API and other protocol parties, from the perspective of a network adversary.

Example: The protocol for PIN verification is faulty – an adversary can simply replay a previous PIN authentication to the server. The authentication server communicates with the security API to verify that the PIN was signed by the terminal, but it does not check for its freshness, i. e., whether the same query has been emitted before, and thus performs an unauthorized transaction.

LAYER 2 The security API according to its specification, from the perspective of any machine that has full or partial access to it.

Example: The security API contains a command to export a PIN derivation key, i. e., a cryptographic secret that allows to generate a PIN given an account number. The command is there for backup uses, but it allows the adversary to export the key in a way that reveals its value. Now the adversary can compute the PIN entry herself and perform an unauthorised transaction. Note that the attacker only uses a sequence of perfectly legal

commands that all work within their specification. Sometimes such attacks on security APIs are called *logical attacks*.

LAYER 3 The actual algorithms that the security API computes in order to meet its specification, from the perspective of any machine that has full or partial access to it.

Example: The algorithm used to derive PINs is flawed. The adversary can forge a PIN without the knowledge of the derivation key.

LAYER 4 The physical implementation of the security API, whatever form it has. Attacks can make use of the concrete implementation of the algorithm, as well as of the hardware it runs on.

Example: The adversary can measure the power consumption during computation of the PIN. He is able to do so, because the enclosure does not protect the charger. The peaks in his reading allow him to conclude the value of the generated PIN, because the PIN derivation algorithm produces a different processor load depending on its output.

This thesis concentrates on attacks on layer one to three. Hardware attacks are out of scope of this work.

The following attack was discovered by Jolyon Clulow [27] and illustrates that requests which are harmless on their own can be composed to form an attack. It also shows that it is not always clear how to classify attacks according to the above scheme.

Example 1 (Wrap/Decrypt Conflict)*:* Consider a security API that contains a store for an arbitrary number of keys. The security API allows the following five kinds of request:

1. When it receives a creation request, it draws a random key k and another random value h, the handle. In its database it stores that h points to the key k, and that the attribute of h is the value "init".

2. When it receives a request "allow decryption" for a handle h, no matter what the current attribute of h is, it is changed to "decrypt"

3. Similarly, when it receives a request "allow wrapping" for a handle h, no matter what the current attribute of h is, it is changed to "wrap"

4. When it receives a wrapping request for h_1 and h_2, it performs a database lookup for the keys k_1 and k_2, that h_1 and h_2, respectively, point to, and the attribute a_1 associated to h_1. If a_1 is "wrap", it outputs the encryption of k_2 under k_1. If a lookup fails, no output is produced.

5. When it receives a decryption request for a handle h and some data c, it performs a database lookup for the key k that h points to, as well as the attribute a associated to h. If a_1 is "decrypt", it outputs the decryption of c using k, if decryption succeeds. Otherwise, if a lookup fails or the decryption fails, no output is produced.

Furthermore, the security API would be expected to specify the dual operations to wrapping and decryption, namely unwrapping (which decrypts a wrapping using the key a handle points to, and stores the result as a key, giving a new handle to the imported key to the user) and encryption (which encrypts a plaintext using the key a given handle points to). For this example, however, theses operations are not of importance.

Although there is no single request that reveals the value of a key k created on the device, it is possible for an attacker to learn this value. She proceeds as follows:

1. The attacker requests the creation of a key and obtains a handle h. The store of the security API consists of a mapping from h to a randomly drawn key k, and a mapping from h to the value "init",

2. She requests the security API to "allow wrapping" for the handle h. Now, h is mapped to "wrap" instead of "init". Otherwise, the store remains unchanged.

3. She requests a wrapping for h with itself. Since the attribute of h is "wrap", she obtains k encrypted under itself.

4. She requests the security API to "allow decryption" for the handle h. Now, h is mapped to "decrypt" instead of "wrap". Otherwise, the store remains unchanged.

5. She requests a decryption of the previously obtained encryption of k under k, using the handle h. Since h has the attribute "decrypt", and h points to k, the decryption step succeeds and the attacker obtains the key k in the clear. Since the secrecy of keys generated on the device is typically guaranteed by security APIs, we consider this an attack.

This attack can be regarded as an attack on the second layer, or as an attack on the third layer. Let us discuss those views one after the other: It is not important which encryption scheme is really used, any security API that follows this specification is prone to this attack. The attack succeeds because the security API fails to separate the roles between a key used for wrapping and unwrapping and a key used for encryption and decryption. Role separation can be implemented by using the attribute to mark the role of each key. This attack is

only possible because a key is allowed to switch roles from "wrap" to "decrypt". The "allow wrapping" and "allow decrypt" requests should only succeed if the attribute associated to the handle is "init". This implements what we call a *policy* for keys: Only keys with the attribute "wrap" are allowed to wrap and unwrap, only keys with the attribute "decrypt" can encrypt and decrypt, and the attributes of each key can only move as indicated by the graph in Figure 1.

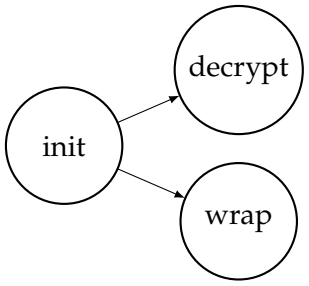


Figure 1: Policy graph: strict role separation.

Another, more permissive policy is secure, despite not strictly separating the roles associated to the attributes "wrap" and "decrypt"(we will formally prove this claim in Chapter 5). In addition to the previous policy, this one allows to change the attribute of a key from "decrypt" to "wrap", but not the other way around (see Figure 2).

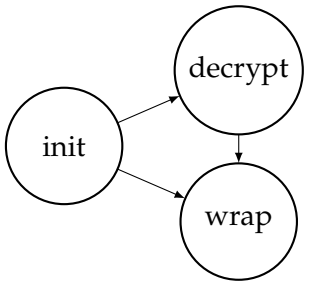


Figure 2: Policy graph: no separation, but still avoiding the attack in Example 1.

On the one hand, the nature of Clulow's attack suggests that it is an attack on layer two. On the other hand, it can be argued that the encryption scheme used for wrapping and unwrapping should be a completely different encryption scheme from the one used for encryption and decryption, since the requirements for those schemes are different. It should, e. g., be impossible to forge a wrapping, that is, to produce a wrapping outside the device that can be unwrapped (imported) with a key generated on the device. Encryption and decryption have different requirements, so one could consider the specification as incomplete in this regard, and the implementation of encryption and wrapping using the same algorithm as an unfortunate choice.

Had two completely different encryption scheme been used for wrapping and decryption, then this attack could have been circumvented – depending on whether wrapping and decryption share the same keys and, if they do, how exactly those schemes interact with each other.

The first part of this thesis discusses the analysis of security APIs as participants in a protocol, and on their own, both in the symbolic model of cryptography, which allows to capture attacks on layer one and two. We will provide techniques to discover logical attacks like Clulow's attack following an approach that takes the point of view standpoint as the first view on the attack.

On the other hand, the security definition we propose in Part ii allows to capture attacks on layer three, but requires security APIs to separate the roles of wrapping and decryption, as they have altogether different requirements. Hence, the definition excludes (secure, but debatable) policies like the policy from Figure 2 in the first place, taking the same standpoint as the second view on the attack.

1.2 SECURITY APIS AND NETWORK PROTOCOLS

The overall goal of a security API in the context of network protocols is to improve the security of the protocol that makes use of it, for instance to make it more resistant in case protocol parties are under adversary control. What security means in this context depends on the protocol.

A commonly used approach is to model the security API as a participant in a protocol, and use traditional protocol analysis techniques to establish protocol-specific security properties. Automated analysis tools that rely on a protocol representation in the symbolic model of cryptography, where messages are represented as formal terms instead of bit strings (also called the Dolev-Yao model [39]) effectively describe the security API on the specification level. Therefore, they are able to detect attacks on layer one and two. The security properties can be specific to the outside protocol, e. g., fairness in a contract signing protocol that uses a security API as a trusted third party (see Section 5.5.4 for a definition of contract signing protocols). It is also possible to show properties that are protocol independent, by modelling the security API as the only entity in the protocol and by giving the adversary full access to the security API. This allows for establishing properties like the secrecy of keys that are generated on the device — no matter which protocol makes use of the security API.

While security protocol analysers can be used for the analysis of security APIs represented as protocols, they often perform badly when protocols are stateful. But, as we have seen in Example 1, the security depends on how attributes in the database can change over time. J.

Herzog [49] advocates improving protocol analysers to overcome this hindrance:

> For API analysis, on the other hand, the idea of mutable state is crucial. A cryptographic device's internal state can be changed by some function calls (migration or key-loading calls, for example), and API-level attacks often use state-changing functions in crucial ways. To use them for interface analysis, therefore, we must extend protocol-analysis techniques to track the device's internal state, [..]

Understanding and overcoming theses difficulties is the goal of the first part of this thesis. In Chapter 2 we introduce some preliminaries, before giving a survey on existing techniques for the analysis of protocols with state in Chapter 3. A large case study discusses the aforementioned Yubikey and the analysis of the authentication protocol that builds upon its use. We present this case study in Chapter 4. We develop our own calculus which allows to express protocols that rely on state manipulation. We devise a method that allows to translate protocols from this higher-level language into a protocol representation on a much lower level. This translation allows us to use existing methods for the analysis of stateful protocols, as a number of case studies confirm. The description of this protocol, the translation, the case studies and the proof of correctness for this translation can be found in Chapter 5.

1.3 SECURITY APIS IN CONTEXT OF ARBITRARY PROTOCOLS

For the producer of a security API, the situation is different: They are unaware of what protocols the security API she is developing might be used for, but would like to verify security properties that can be established

A. independent of the protocol,

B. that guarantee that there are no flaws on layer 3 and layer 2, and

C. that facilitate the analysis of higher-level protocols for her customers.

A definition of what constitutes a "secure" security API in the computational setting, where messages are represented as actual bit strings and cryptographic functions are not necessarily perfect, could establish such properties. The greatest challenge in formulating this definition is that security APIs are quite diverse, as the previous examples have shown. In the second part of this thesis, we describe an approach to finding a definition of security that is reasonable for a wide range of APIs, but still not for all. This definition uses a framework that allows for modular analysis in the computational model,

supporting the analysis of higher-level protocols. Especially in scenarios where several devices implement the same security API – which is often the case for reasons of redundancy and high-throughput, as security APIs are often mission-critical – our definition facilitates the analysis of higher-level protocols. By modelling the case of several devices implementing the same security API, our definition is able to ensure that a given policy is respected not only on each individual device, but on the global level. The following examples shows that a policy that is respected locally can still be violated on the global level:

Example 2: The security API from Example 1 additionally allows to reimport a key using an unwrap request. When it receives an unwrap request for a handle h and some data c, it performs a database lookup for the key k, that h points to, as well as the attribute a, associated to h. If a_1 is "wrap", it encrypts c using k. Should the decryption succeed, a new handle h' is drawn. If the result of the decryption, say, k' is already on the device, let a' be the attribute associated to some handle pointing to k' (it should be the same for any handle pointing to some key equal to k'). If k' is not already on the device, let a' be "init". In its database, the security API stores that h' points to the key k', and that the attribute of h' is the value a'. To be on the safe side, the second handle for a wrapping request has to be associated to "wrap" or "decrypt", but not "init".

Even if the device implements the policy from Figure 1 on a local scale, it is possible to use the same key for wrapping as well as for encryption: An attacker can wrap some key that is either associated to "wrap" or "decrypt", and import the key on a second device using an unwrap request (given that both devices share a common wrapping key). The key will be new on the second device, so its handle is associated the attribute "init". Using either an "allow decryption" or "allow wrapping" request, a copy on the key can be produced that has "wrap" set on one device, and "decrypt" on the other. While the policy was respected on each of the two devices, it is not respected globally.

In the second part of this thesis, we establish a template for the design of security APIs that are secure with respect to our definition. It offers insight in some "best practices" for the design of a security API and provides a template for a secure implementation for a wide range of security APIs, as it can be easily extended without losing security.

To this end, we introduce the concept of key-usage functionalities. Some features are vital for the security of a set of security APIs, but most features only have to be secure with respect to the function they perform. The second kind of feature performs what we call key-usage (as opposed to key-management). The distinction between the two allows us to derive a design that allows to safely add the second kind of feature. If the second kind of feature can be implemented securely,

then this implementation can be integrated into the system, preserving the security of the resulting, extended system. This is important; Ross Anderson makes following observation about a number of attacks [4, 17, 18] on security APIs [3]:

> At a practical level, [those attacks illustrate] one of the main reasons APIs fail. They get made more and more complex, to accommodate the needs of more and more diverse customers, until suddenly there's an attack.

Considering the apparent need to adapt security APIs to changing requirements, we propose this "generic" implementation, and subsequently show it secure, as a means to the designer of a security API to easily tell whether a certain function can be added without further ado.

The second part of this thesis is organized as follows: We discuss our approach for the definition of security for security API implementations, as well as previous definitions in Chapter 6. In Chapter 7, we discuss the framework in which we will formulate this definition. In Chapter 8, we introduce our notion of security, as well as our design template. In Chapter 9, we discuss this notion and its limitations. In Chapter 10, we show that our design template is secure with respect to this notion. In Chapter 11, we instantiate this design template and show how our notion of security helps the analysis of protocols using security APIs.

Part I

ANALYSIS OF STATEFUL PROTOCOLS IN THE SYMBOLIC MODEL

Security APIs, such as the RSA PKCS#11 standard [81] or IBM's CCA [25], have been known to be vulnerable to logical attacks for quite a while [64, 17]. Formal analysis is necessary to identify secure configurations. Several models have been proposed, for example [44], however, manual inspection is tedious and error-prone. A promising paradigm is to regard the Security API as a participant in a protocol and then use traditional protocol analysis techniques, e. g., symbolic techniques [20, 29, 36], for analysis.

This section presents a survey on existing techniques to such analysis. We introduce two examples that help pointing out the differences in existing methods and where they are lacking. We propose a modelling on the basis of multiset rewriting, and show the advantages by verifying security properties of the Yubikey-protocol and the YubiHSM. We introduce our own process calculus, which include operators for manipulation of a global store. This calculus can be translated into MSR rules. We show this translation to be both sound and complete.

2

PRELIMINARIES

2.1 TERMS AND EQUATIONAL THEORIES

As usual in symbolic protocol analysis we model messages by abstract terms. Therefore we define an order-sorted term algebra with the sort *msg* and two incomparable subsorts *pub* and *fresh*. For each of these subsorts we assume a countably infinite set of names, *FN* for fresh names and *PN* for public names. Fresh names will be used to model cryptographic keys and nonces while public names model publicly known values. We furthermore assume a countably infinite set of variables for each sort s, $\mathcal{V}_s$ and let $\mathcal{V}$ be the union of the set of variables for all sorts. We write $u : s$ when the name or variable u is of sort s. Let Σ be a signature, i.e., a set of function symbols, each with an arity. We denote by $\mathcal{T}_\Sigma$ the set of well-sorted terms built over Σ, *PN*, *FN* and $\mathcal{V}$. For a term t we denote by $names(t)$, respectively $vars(t)$ the set of names, respectively variables, appearing in t. The set of ground terms, i.e., terms without variables, is denoted by $\mathcal{M}_\Sigma$. When Σ is fixed or clear from the context we often omit it and simply write $\mathcal{T}$ for $\mathcal{T}_\Sigma$ and $\mathcal{M}$ for $\mathcal{M}_\Sigma$.

We equip the term algebra with an equational theory E, e. g., a finite set of equations of the form $M = N$ where $M, N \in \mathcal{T}$. From the equational theory we define the binary relation $=_E$ on terms, which is the smallest equivalence relation containing equations in E that is closed under application of function symbols, bijective renaming of names and substitution of variable by terms of the same sort. Furthermore, we require E to distinguish fresh names, i. e., $\forall a, b \in FN : a \neq b \Rightarrow a \neq_E b$.

Example 3: Symmetric encryption can be modelled using a signature $\Sigma = \{senc/2, sdec/2, encSucc/2, true/0\}$ and an equational theory defined by $dec(enc(m, k), k) = m$ and $encSucc(senc(x, y), y) = true$. Using the last equation, one can check whether a term can be decrypted with a certain key.

For the rest of Part i we assume that E refers to some fixed equational theory and that the signature and equational theory always contain symbols and equations for pairing and projection, i.e., $\{\langle .,. \rangle, \mathsf{fst}, \mathsf{snd}\} \subseteq \Sigma$ and equations $\mathsf{fst}(\langle x, y \rangle) = x$ and $\mathsf{snd}(\langle x, y \rangle) = y$ are in E.

We also employ the usual notion of positions for terms. A position p is a sequence of positive integers and $t|_p$ denotes the subterm of t at position p.

2.2 FACTS

We assume an unsorted signature Σ_{fact}, disjoint from Σ. The set of *facts* is defined as

$$\mathcal{F} := \{F(t_1, \ldots, t_k) | t_i \in \mathcal{T}_\Sigma, F \in \Sigma^k_{fact}\}.$$

Facts will be used both to annotate protocols, by the means of events, and for defining multiset rewrite rules. We partition the signature Σ_{fact} into linear and persistent fact symbols. In multiset rewriting, which will be introduced in Section 3.6.1, linear facts may disappear, while persistent facts may not. We suppose that Σ_{fact} always contains a unary, persistent symbol !K and a linear, unary symbol Fr. Given a sequence or set of facts S we denote by $lfacts(S)$ the multiset of all linear facts in S and $pfacts(S)$ the set of all persistent facts in S. By notational convention facts whose identifier starts with '!' will be persistent. $\mathcal{G}$ denotes the set of ground facts, i.e., the set of facts that does not contain variables. For a fact f we denote by $ginsts(f)$ the set of ground instances of f. This notation is also lifted to sequences and sets of facts as expected.

2.3 SUBSTITUTIONS

A substitution σ is a partial function from variables to terms. We suppose that substitutions are well-typed, i.e., they only map variables of sort s to terms of sort s, or of a subsort of s. We denote by $\sigma = \{^{t_1}/_{x_1}, \ldots, ^{t_n}/_{x_n}\}$ the substitution whose domain is $\mathbf{D}(\sigma) = \{x_1, \ldots, x_n\}$ and which maps x_i to t_i. As usual we homomorphically extend σ to apply to terms and facts and use a postfix notation to denote its application, e.g., we write $t\sigma$ for the application of σ to the term t. A substitution σ is grounding for a term t if $t\sigma$ is ground. We use $f := g[a \mapsto b]$ to denote $f := g(x)$ for $x \neq a$ and $f := b$ otherwise, even if f or g are not substitutions.

2.4 SETS, SEQUENCES AND MULTISETS

We write $\mathbb{N}_n$ for the set $\{1, \ldots, n\}$. Given a set S we denote by S^* the set of finite sequences of elements from S and by $S^\#$ the set of finite multisets of elements from S. We use the superscript $^\#$ to annotate usual multiset operation, e.g. $S_1 \cup^\# S_2$ denotes the multiset union of multisets S_1, S_2. We will furthermore use the hash symbol to distinguish multisets $\{\ldots\}^\#$ from sets $\{\ldots\}$. Given a multiset S we denote by $set(S)$ the set of elements in S. The sequence consisting of elements $e_1, \ldots, e_n$ will be denoted by $[e_1, \ldots, e_n]$ and the empty sequence is denoted by $[]$. We denote by $|S|$ the length, i.e., the number of elements of the sequence. We use $\cdot$ for the operation of adding an element either to the start or to the end, e.g.,

$e_1 \cdot [e_2, e_3] = [e_1, e_2, e_3] = [e_1, e_2] \cdot e_3$. Given a sequence S, we denote by $idx(S)$ the set of positions in S, i.e., $\mathbb{N}_n$ when S has n elements, and for $i \in idx(S)$, S_i denotes the ith element of the sequence. Application of substitutions and $=_E$ are lifted to sets, sequences and multisets as expected. Set membership modulo E is denoted by $\in_E$, i.e., $e \in_E S$ if $\exists e' \in S.\ e' =_E e$. By abuse of notation we sometimes interpret sequences as sets or multisets; the applied operators should make the implicit cast clear.

3

A SURVEY ON EXISTING METHODS

Automated analysis of security protocols has known increasing success. Using automated tools, flaws have been discovered,e. g., in the Google Single Sign On Protocol [7], in commercial security tokens implementing the PKCS#11 standard [20], as well as Lowe's attack [66] on the Needham-Schroeder public key protocol 17 years after its publication. While efficient tools such as ProVerif [14], AVISPA [6] or Maude-NPA [41] exist, these tools fail to analyze protocols that require a *non-monotonic global state*, i.e., some database, register or memory location that can be read and altered by different parallel threads. In particular ProVerif, one of the most efficient and widely used protocol analysis tools, relies on an abstraction of protocols as Horn clauses which are inherently non-monotonic: Once a fact is true it cannot be set to false anymore. This abstraction is well suited for the monotonic knowledge of an attacker (who never forgets), makes the tool extremely efficient for verifying an unbounded number of sessions and allows for building on existing techniques for Horn clause resolution. While ProVerif's input language, a variant of the applied pi calculus [2], supports a priori encodings of a global memory, the abstractions performed by ProVerif introduce false attacks.

Security APIs have been known to be vulnerable to logical attacks for some time [64, 17]. Therefore, formal analysis is necessary to identify secure configurations. Several models have been proposed, some of which require manual inspection [44], which is tedious and error-prone. A promising paradigm is to regard the security API as a participant in a protocol. Is it possible to translate the success protocol analysers like ProVerif have had in the domain of network protocols to this application? In contrast to most network protocols, security APIs often rely on the state of an internal database, which we can model as a set of facts, i. e., predicates on ground terms. When moving from one state to another, facts might be added or subtracted, therefore we say that the state of a security API is *non-monotonic*. When it comes to the analysis of an unbounded number of keys, this non-monotonicity is the reason why current tools for protocol analysis do not work well. Herzog [49] already identified not accounting for mutable global state as a major barrier to the application of security protocol analysis tools to verify security APIs. There are other protocols besides security APIs that maintain databases, too. Key servers need to store the status of keys, in optimistic contract signing protocols a trusted party maintains the status of a contract, Radio-frequency identification (RFID) protocols maintain the status of tags

and, more generally, websites may need to store the current status of transactions.

We present a survey on how different techniques to deal with state in protocol analysis perform on the application of security APIs. This includes two abstractions for the use with ProVerif (a protocol verifier based on a representation of the protocol by Horn clauses) [5, 74], an extension of the strand space model [47], and the tamarin prover [89], which represents protocols as multiset rewrite rules.

While the first two approaches, both based on Horn clause resolution, are not sufficient for modelling even simple fragments of PKCS#11, they show quite well why Horn clauses are not suited for this kind of problems: If the intruder knowledge at a point in time is coupled with the current state of the security API, ProVerif is very likely not to terminate. If it is not coupled with the current state of the security API, the intruder can derive knowledge and then "rewind" the security API, which introduces false attacks.

The strand space model captures the *causal* relations between protocol input/output, adversarial deductions and transitions from one state to another by modelling an execution of protocol and adversary as an acyclic graph. As in the tamarin prover, the state itself is modelled as a multiset of facts. As of now, protocol analysis in the strand space model with state has not been mechanized yet.

The tamarin prover borrows ideas from the strand space model, but uses classical multiset rewriting for protocol description and adversarial deduction. It implements a backward induction technique that explicitly models the causality between a message deduced by the adversary and the protocol output allowing the deduction. Although the tool is in an early development stage, we will show why it has the potential to make fully automated verification of security APIs possible.

3.1 RUNNING EXAMPLES

The first example was introduced by Arapinis et al. as a running example for the StatVerif tool [5].

Example 4 (Left-or-right encryption)*:* A device is initialized to act in either "left" or "right" mode. A key k is chosen on initialisation, and the attacker receives the encryption of a pair of secret values under that key. Once it is initialized, the only possible operation is to send an encryption of a pair (x, y) under k to the device. If the device is in "left" mode, it outputs x, otherwise it outputs y. The attacker's goal is to learn *both* secret values.

The security of this example relies on the fact that the output that the adversary receives depends on the state of the system, and that the states "left", "right" and the initial state are exclusive.

The second example is inspired by problems that occur with key-management APIs in the style of PKCS#11: Those APIs support creating a large number of keys, which are supposed to stay secret. Nevertheless it is necessary to export such keys. For this reason, there is a mechanism called wrapping: A key can be encrypted using a second key, making it safe to store the key outside the device. The secrecy of the key inside the wrapping depends on the secrecy of the second key.

As mentioned in Example 1 on page 7, the existence of a wrap command might allow for an attack, if a decryption command is available, too. The second example is a minimal example with respect to this conflict.

Example 5 (Wrap/Dec)*:* A security device accepts the following four kinds of request:

1. When it receives a creation request, it draws a random key k and another random value h, the handle. In its database it stores that h points to the key k, and that the attribute of h is the value "init".

2. When it receives a wrapping request for h_1 and h_2, it performs a database lookup for the keys k_1 and k_2 that h_1 and h_2, respectively, point to, and the attribute a_1 associated to h_1. If a_1 is "wrap", it outputs the encryption of k_2 under k_1. If the lookup fails, no output is produced.

3. When it receives a decryption request for a handle h and some data c, it performs a database lookup for the key k that h points to, as well as the attribute a associated to h. If a_1 is "dec", it outputs the decryption of c using k, if decryption succeeds. Otherwise, if the lookup fails, no output is produced.

4. When it receives a request to change the attribute of h to a', it performs a database lookup for the current attribute of h. Depending on its current value, the attribute in the database is changed to a', or nothing is done. We consider the following three policies:

   a) No matter what the current attribute is, it may be changed to "dec" or "wrap". This allows for the attack sketched in Example 1.

   b) Attributes may only change from "init" to "wrap" and from "init" to "decrypt". This enforces a role separation of wrapping and decryption keys.

   c) Like the previous policy, but attributes may also change from "dec" to "wrap".

$\langle M, N \rangle ::=$	(terms)
$\mid x, y, z$	(variables)
$\mid a, b, c, s$	(names)
$\mid f(M_1, \ldots, M_n)$	(constructor appl.)
$\langle P, Q \rangle ::=$	(processes)
$\mid 0$	(nil)
$\mid P \mid Q$	(parallel composition)
$\mid !P$	(replication)
$\mid \nu n; P$	(restriction)
$\mid \text{out}(M, N); P$	(output)
$\mid \text{in}(M, N); P$	(input)
$\mid \text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q$	(destructor appl.)
$\mid \text{if } M = N \text{ then } P \text{ else } Q$	(conditional)
$\mid [s \mapsto M];\ P$	(state initialisation)
$\mid \text{read } s \text{ as } x; P$	(read)
$\mid s := M;\ P$	(assign)
$\mid \text{lock};\ P$	(locked section)
$\mid \text{unlock};\ P$	(locked section)

Figure 3: Syntax of the StatVerif calculus.

3.2 STATVERIF

StatVerif is a compiler from the StatVerif calculus to Horn clauses which can either be handled by ProVerif, or by the automated theorem prover SPASS [5, 97]. The StatVerif calculus extends ProVerif's process calculus by the following additional constructs that handle global state:

- initialisation of a state cell, $[s \mapsto M]$,
- reading a value off a state cell, `read` s `as` x,
- writing a value to a state cell, $s := M$, and
- two operators for locking and unlocking the whole state[1],

where s is a name addressing a cell, M can be any term and x is a variable that will be bound to the value of the cell. We see that the store at some point in time is a function from names (cell names) to terms. It is global in the sense that it might be addressed anywhere

1 Newer versions support locking individual cells.

```
let device =
   out(c,pk(k)) |
   (!in(c,x); lock(s); read s as y;
      if y = init then
         (if x = left then s := x; unlock(s)
         else if x = right then s := x; unlock(s)) ) |
   (!in(c,x); lock(s); read s as y;
      if y = left then out(c,car(adec(k,x))); unlock(s)
      else if y = right then out(c,cdr(adec(k,x))); unlock(s)).

let user = ν r; out(c,aenc(pk(k),r,pair(sl,sr))).

process
   ν k; ν s; [s ↦init] | device | ! user
```

Figure 4: Example 4 in StatVerif. The deconstructors car and cdr extract the first respectively the last element of a pair.

in a process, but not by the adversary. See Figure 3 for the complete syntax. See [5] for the semantics. We just note that StatVerif, as well as ProVerif, divide the function symbols in the signature Σ into constructors and destructors.

The following syntactical restrictions apply:

A. $[s \mapsto M]$ may only occur once for a given cell name s, and only in scope of a new, a parallel and a replication, but no other construct.

B. In every branch of the syntax tree, every lock must be followed by precisely one corresponding unlock and in between, no parallel, replication and unlock are allowed.

3.2.1 *Running Example 4: Left-or-right encryption in StatVerif*

The listing in Figure 4 shows the modelling of Example 4. Using the Horn clauses produced by the StatVerif translation as input, ProVerif can show that, for secret values sl and sr, the adversary is unable to produce the pair pair(sl,sr).

3.2.2 *Running Example 5: Wrap/Dec in StatVerif*

The modelling is more difficult in the case of Example 5. In many practical systems, the handle pointing to a key is some encoding of its address in memory. This would correspond to modelling the handle as a cell name in StatVerif. Developing this idea further, the handle is in the scope of a restriction, which in turn is in the scope of a replication, to allow for an unbounded number of keys. The pro-

cess handling creation requests would look similar to the following process:

```
!(in(c,create); ν handle; ν key; handle:=key; out(c,handle)).
```

To treat other requests, for instance, a decryption request, the process needs to read the key that belongs to some given handle. It can only address the key using a cell name, in this case, the corresponding handle. Since the cell name is in the scope of a restriction, the process that handles decryption requests needs to be in the same scope.

A wrapping request has the same requirement for the wrapping key, but also for the key to be wrapped. Since both keys might be different, they could have different handles, which may point to different keys. But there is only one cell name in scope of the above restriction.

We argue that StatVerif is not suitable for modelling Example 5 for an unbounded number of keys, since *a*) the subprocess treating wrapping queries cannot access all keys, unless there is a cell in the store whose value is not bound a priori, and *b*) our experiments in case we have such a value, e. g., a single cell storing the list of keys, suggest neither ProVerif not SPASS handle this case well.

The argument for *a*) is as follows: Let $\mathtt{P}'$ be a subprocess that treats a wrapping query. It has to access two possibly different keys to produce a wrapping. Since processes are finite, $\mathtt{P}'$ can only be a subprocess of a finite number of processes that handle creation request. This means, if $\mathtt{P}'$ is in the scope of one of the key's restriction, it cannot be in the scope of the other key's restriction. The number of those keys is not bound a priori, so to be able to address all keys, $\mathtt{P}'$ would need to be able to access an unbounded number of keys via the store. But assuming that each term referred in the store is of constant length, the sum of the length of terms in the store $\mathtt{P}'$ can access is fixed as well, since the number of cells is fixed a priori.

To support part *b*) of the previous claim, we conducted an experiment where we modelled the store using a single state cell and stored the keys as a list of pairs of handles and keys, see Listing 13 in Section A.1, page 161. Whenever a key is looked up, there is a fetch process that traverses the list. Not very surprisingly, this kind of modelling turns out to be difficult to handle using either ProVerif or SPASS. Both theorem provers did not terminate during our experiments. In Section 3.4 we investigate further on why translations into Horn clauses are not suitable for this kind of problems in general.

3.3 ABSTRACTION BY SET-MEMBERSHIP

Similar to StatVerif, this approach aims at translating a protocol specification into Horn clauses, which can then be verified using SPASS or ProVerif. The main idea is the following: Given a finite number

of sets, there are so-called *abstractable symbols* which might be elements of those sets. Those symbols can occur in terms as variables or names. They are abstracted by the available information about which sets they are members of. For instance, there might be the sets *Wrap*, *Dec* and *Init* of keys that have the attribute "wrap", "dec" and/or the initial attribute, respectively. Every key that is in *Wrap*, not in *Dec* and not in *Init* is represented by the term $val(1,0,0)$, where the first position represents the membership in *Wrap*, the second the membership in *Dec* and the third the membership in *Init*.

The infinite set of keys is represented by a finite amount of representatives, if there are finitely many sets. Consider now that the adversary knows a message containing, as a subterm, a key k in "init", but not in "dec" or "wrap". If k might move from "init" to "dec", it would be represented by `val(0,0,1)`. The abstraction handles the intruder knowledge as follows: All previous facts, including those modelling the intruder knowledge, are maintained, but additionally, all facts where k appears in its old form `val(1,0,0)` are added, but with `val(1,0,0)` substituted by `val(0,0,1)`.

The protocol specification language, called AVISPA Intermediate Format (AIF) [83], is not a high-level language as in the case of StatVerif, but a much simpler calculus based on set-rewriting. The state of the protocol is modelled as a set of facts, for instance, the fact iknows has arity 1 and represents the intruder knowledge, i. e., if $\mathsf{iknows}(m)$ is in the state, the intruder knows the message m. In addition, the protocol state might contain *set conditions*, which are of the form $t \in M$ or $t \notin M$, where t is an abstractable symbol and M is a ground term free of abstractable symbols. Abstractable symbols can appear in other terms, for example in facts like iknows.

A transition rule has the following form:

$$LF \cdot S_+ \cdot S_- \dashv[F]\rightarrow RF \cdot RS.$$

Loosely speaking, the left-hand side of a rule describes to which states the rule can be applied, while the right-hand side describes the changes to the state after the transition: LF represents a set of facts that is required to be present in the state before the transition, RF is the set of facts that is added after the transition, S_+ and S_- are sets of positive and negative set conditions that are required to be present in the state before the transition. RS is the set of positive set conditions that is added after the transition and F is a set of newly introduced abstractable symbols. Formally, a transition from a set $\mathcal{S}$ to a set $\mathcal{S}'$ is possible, written $\mathcal{S} \Rightarrow_r \mathcal{S}'$, if there is a grounding substitution σ of a rule of the above form, and the following conditions are fulfilled:

- $(LF \cdot S_+)\sigma \subset \mathcal{S}$
- $(S_- \cap \mathcal{S} = \emptyset)$ and $(\mathcal{S}' = \mathcal{S} \setminus S_+\sigma) \cup RF\sigma \cup RS\sigma$

- $F\sigma$ are fresh (i. e., they do not occur in $\mathcal{S}$ or in any rule).

Facts and negative set conditions are persistent, but positive set conditions can be removed, namely if they appear in S_+ but not in RS. A state $\mathcal{S}$ is reachable using the set of transition rules R, if and only if $\emptyset \Rightarrow_R^* \mathcal{S}$, where $\Rightarrow_R = \cup_{r \in R} \Rightarrow_r$ and $\Rightarrow_R^*$ is the reflexive transitive closure of $\Rightarrow_R$.

Let us have a look at the following example:

```
−[SK]→ SK∈init(t)·iknows(senc(SK,pair(sl,sr)))
```

This rule has an empty left-hand side, and introduces a new abstractable symbol *SK*. Since *SK* is fresh, the state is extended by a fact `iknows(senc(SK,pair(sl,sr)))` and a positive set condition `SK`$\in$ `init(t)`. This models a protocol step where a secret key is freshly drawn, which is then added to the "init" database of a token `t` (`t` is a constant in this case). Finally, the encryption of a pair $\langle sl, sr \rangle$ under this freshly drawn key is output to the adversary. Assuming there are three sets, `init(t)`, `dec(t)` and `wrap(t)`, this rule is translated to the following Horn clause:

```
⟶ iknows(senc(iknows(1,0,0),pair(sl,sr)))
```

This expresses that the adversary, without any preconditions, can learn the encryption of the secret pair under a key that is in `init(t)`, but not in `dec(t)` or `wrap(t)`. The translation will produce additional clauses, but this should give the gist of it.

3.3.1 *Running Example 5: Wrap/Dec using abstraction by set-membership*

Abstraction by set-membership works remarkably well on Example 5. See Figure 5 for the code of this model. We use a unary function `h` to model the handle that belongs to a given key. Since the handle should reveal no information about the value of the key, there is no rule to extract a term `t` from a term `h(t)`.

The first four lines model the intruder's deductive capabilities. The modelling of the security device uses four sets in total: `store`, which models the set of keys that are available on the device, as well as `init`, `dec` and `wrap`, which model the set of keys that have the respective attributes set.

Wrapping is modelled as a rule that for two handles that are known to the adversary, checks if they belong to keys in the store, if the wrapping key has "wrap" set and then adds the fact `iknows(senc(KEY1,KEY2))`, but does not alter the state any further. The same holds for decryption. The security goal is secrecy of keys: If the attacker knows a key that is at the same time in the store, we consider this an attack. We modelled different policies. The following two pairs of rules both model a policy where, after initialisation, keys might

```
% The intruder's deduction capabilities:
⟶ iknows(Token);
iknows(K)·iknows(M) ⟶ iknows(senc(K,M));
iknows(senc(K,M))·iknows(K) ⟶ iknows(M);

% Generate a key
⊣[KEY]⊢ iknows(h(KEY))·KEY∈store(Token)·KEY∈init(Token);

% Wrap
iknows(h(KEY1))·iknows(h(KEY2))
·KEY1∈wrap(Token)·KEY1∈store(Token)·KEY2∈store(Token)
⟶ iknows(senc(KEY1,KEY2))·
KEY1∈wrap(Token)·KEY1∈store(Token)·KEY2∈store(Token);

% SDecrypt
iknows(h(KEY))·iknows(senc(KEY,M))·KEY∈store(Token)·KEY∈dec(
    Token)
⟶ iknows(M)·KEY∈store(Token)·KEY∈dec(Token);

% Security goal
iknows(KEY)·KEY∈store(Token) ⟶attack;
```

Figure 5: Modelling for Example 5 in AIF. The full code can be found in Listing 16 in Section A.1, p. 164.

become either wrapping keys or decryption keys. Otherwise, those roles are permanent and thus exclusive.

```
% Policy B
KEY∈store(Token)· KEY∈init(Token)
        ⟶ KEY∈store(Token)· KEY∈wrap(Token);
KEY∈store(Token)· KEY∈init(Token)
        ⟶ KEY∈store(Token)· KEY∈dec(Token);
```

The second pair of rule does not check whether a key is in init or not, but instead uses a negative set condition to determine if the key is in an initial state, i. e., has not been assigned the role of either a decryption or a wrapping key yet.

```
% Policy B' - simplified version of Policy B
KEY∈store(Token)· KEY∉dec(Token)
        ⟶ KEY∈store(Token)· KEY∈wrap(Token);
KEY∈store(Token)· KEY∉wrap(Token)
        ⟶ KEY∈store(Token)· KEY∈dec(Token);
```

For either choice of policy, ProVerif confirms within milliseconds that a state where the fact `attack` is present cannot be reached. The same holds for the following policy, where a key might move from "dec" to "wrap", but not the other way around.

```
% Policy C
KEY∈store(Token)· KEY∈init(Token)
```

```
        ⟶ KEY ∈ store(Token) · KEY ∈ wrap(Token);
KEY ∈ store(Token) · KEY ∈ init(Token)
        ⟶ KEY ∈ store(Token) · KEY ∈ dec(Token);
KEY ∈ store(Token) · KEY ∈ dec(Token)
        ⟶ KEY ∈ store(Token) · KEY ∈ wrap(Token);
```

Finally, the following example policy allows for an attack:

```
% Policy A -- an attack is reachable·
KEY ∈ store(Token) · KEY ∈ init(Token)
        ⟶ KEY ∈ store(Token) · KEY ∈ wrap(Token);
KEY ∈ store(Token) · KEY ∈ init(Token)
        ⟶ KEY ∈ store(Token) · KEY ∈ dec(Token);
KEY ∈ store(Token) · KEY ∈ wrap(Token)
        ⟶ KEY ∈ store(Token) · KEY ∈ dec(Token);
```

This policy allows the attacker to change a key's attribute from "init" to "wrap", produce a wrapping of a second key (possibly the wrapping key itself), move the key from "wrap" to "dec" and finally obtain a decryption of the key that was just wrapped. The translation correctly allows ProVerif to conclude that such an attack exists, showing that this model captures the attack described in Example 1.

3.3.2 *Running Example 4: Left-or-right encryption*

The left-or-right encryption example (Example 4) shows the limitations of abstraction by set-membership. A straight-forward modelling of this example (see Figure 6) shows where the abstraction by set-membership is too coarse. The modelling first generates an abstractable name `sk` which is in the set `init(token)`, but may be in other sets, too. It is not possible to specify negative set conditions on the right-hand side, which is why a second rule allows the adversary to obtain the encryption of the secret pair only if `sk`, the only abstractable name ever generated, is in `init(token)`, not in any of the other two sets. It is then possible to move the set membership from "init" to "left", or from "init" to "right". Decryption gives the left or the right element of the pair depending on the set membership. If the adversary can derive both the left and the right part of the secret, she has successfully mounted an attack.

ProVerif finds a (false) attack on the Horn clauses produced by this abstraction. The reason is the following: The adversary can learn the secret pair encrypted under the abstraction of this key, i. e., `iknows(senc(var(1,0,0),pair(sl,sr))` is a derivable fact. If this is the case, so is `iknows(senc(var(0,1,0),pair(sl,sr))`, since a key that is abstracted by `val(1,0,0)` can move to "left", and therefore be substituted by `(val(0,1,0)`. If `iknows(senc(var(0,1,0),pair(sl,sr))` is present, `sl` can be derived. With the same argument, `iknows(senc(var(0,0,1),pair(sl,sr))`, and therefore `sr` can be derived, which shows that the abstraction introduces a false attack in this case.

```
sk ⟶ sk∈init(token);
SK∈init(token)·  SK∉left(token)·  SK∉right(token)·
        ⟶ SK∈init(token)·iknows(senc(pair(sl,sr),SK));

%Set left - or - set right:
SK∈init(token) ⟶ SK∈left(token);
SK∈init(token) ⟶ SK∈right(token);

%decryption
SK∈left(token)·iknows(senc(pair(X,Y),SK)) ⟶ SK∈left(token)·
    iknows(X);
SK∈right(token)·iknows(senc(pair(X,Y),SK)) ⟶ SK∈right(token)·
    iknows(Y);

%Security goal
iknows(sl)·iknows(sr) ⟶attack;
```

Figure 6: Modelling for Example 4 in AIF. The full code can be found in Listing 14 in Section A.1, p. 162.

The main problem here is that, in this abstraction, every key which has the same set-membership is equivalent. This means that there are really only three encryptions of the secret pair: one under `val(1,0,0)`, one under `val(0,1,0)`, and one under `val(0,0,1)`. At the same time, the abstraction produces two clauses which allow the attacker to obtain an encryption under `val(0,1,0)` from an encryption under `val(1,0,0)` as well as an encryption under `val(0,0,1)`. Both clauses can be used as often as necessary to obtain this (false) attack.

3.4 WHY HORN CLAUSE APPROACHES ARE UNSUITABLE

Why approaches based on Horn clause resolution are often not suitable

While protocols can be accurately represented as multiset rewrite systems [89, 37] or in linear logic [40], the reason for ProVerif's efficiency, both in terms of computational resources and degree of automation, lies in the use of a Horn clause representation for the protocol. The attacker knowledge is modelled as a fact $\mathtt{att}(m)$, which is present if the attacker knows a message m. The attacker's deductive capabilities are modelled as Horn clauses that derive a fact $\mathtt{att}(m)$ from one or more facts $\mathtt{att}(m_1), \ldots, \mathtt{att}(m_n)$. This part of the modelling seems accurate, because indeed: If an attacker knows m at some point, much like the fact $\mathtt{att}(m)$ stays available for future deductions, the attacker never forgets this message. Furthermore, if the attacker can derive m from m', she needs to know m' in advance.

Protocols are modelled as oracles: If the attacker knows a message m at some point, then he can potentially learn a new message by

sending m to the protocol and thereby learning a message m' which might depend on m. These protocol steps are modelled as Horn clauses, they can be repeated arbitrarily often, even if the real system prevents that with challenge-response mechanisms or timestamps. In the context of security APIs, this often results in false attacks, as often a database lookup is guarding critical protocol steps – e. g., in Example 5, the decryption operation validates that a key has the "dec" attribute set. The statement "key x has attribute a" might be true at one point in time, and false in another – in Example 5, for instance, a key can have "init" set, but this attribute could be removed later. However, facts produced by the Horn clause presentation of the protocol persist.

The approach StatVerif takes is quite obvious: The translation embeds the state into the `att`-fact. There is an explicit state given by the number of state cells that appear in the protocol. The attacker knowledge is now given in relation to the explicit state of the system, so, $\texttt{att}(\sigma, t)$ means that the attacker can derive t when the system is in state σ. If the state can change from σ to σ', then the attacker knowledge is preserved, so $\texttt{att}(\sigma, t)$ implies $\texttt{att}(\sigma', t)$ in this case. Note that this is not the complete state of a protocol, as the implicit state of the protocol, i. e., which part is at what point of the execution, is not recorded.

The resulting Horn clauses are meant to be verified using ProVerif, which attempts a proof by refutation. The number of terms in the store is hence fixed a priori by the number of cell names appearing in the protocol description. Even if we "cheat" our way around this restriction by encoding a list of terms in a single cell using, e. g., nested pairs, as we did in Section 3.2.2, the resolution algorithm used by ProVerif does not terminate. The reason is the following: In order to access an element of the list, e. g., to test of membership, or to retrieve a certain item, a recursion on the list is necessary. This translates to a Horn clause with the term encoding the list in the hypothesis and a subterm of this term (typically the tail of the list) in the conclusion. Backwards induction on this Horn clause, which represents the recursive step, and the Horn clause that represents the initial call of the recursion, produces an infinite set of clauses.[2] A recent extension to ProVerif by Blanchet and Paiola represents lists using a generalized notion of Horn clauses and achieves successful verification of a number of stateless protocols involving lists of unbounded length [16]. Combining this extension with StatVerif's translation procedure could remedy this problem.

StatVerif avoids this issue, at least for the case where only terms of fixed length are stored in cells, by imposing an artificial limit on the

2 The recursive step can be unified with the call, resulting in a clause that represents the state as a list of one element and a tail. This clause can again be unified with clause representing the recursive step, resulting in a similar clause, which represents the state as a list of two elements and a tail. This continues ad infinitum.

number of state cells, effectively imposing a bound on the total state. As a consequence, previous works using the same abstraction [35] and more recent work using StatVerif [5, 9] suffer from limitations on the environment (the model of the TPM for instance requires a bound on the number of times the computer can be rebooted [35]), limitations on the number of fresh values that can be stored, as well as on the number or security devices that can be modelled (the modelling of Example 4 in StatVerif cannot capture the interaction between an unbounded number of security devices in the network) and often require additional soundness theorems that justify simplifications on a per-case basis [35, 9].

The "abstraction by set-membership" approach circumvents this problem in a clever way: There are only certain parts of the state of a protocol (abstractable objects in their terminology) that have properties that change over time and that are important for analysis. Rather than incorporating the complete state of a system into each fact, this abstraction incorporates only the set membership of objects that appear in terms relevant to each fact. As a side effect, the state space is made "finite": Every abstractable object is substituted by a representative of its equivalence class, where two objects are equivalent if they are members of exactly the same sets. Given that the number of sets is finite, there are only finitely many such classes, 2^n for n the number of sets. This greatly reduces the amount of terms that might appear inside the `att`-facts and therefore works very well with ProVerif's refutation algorithm.

Unfortunately, this abstraction is too coarse for some cases that are relevant for our target application, security APIs. Example 4 (see Section 3.2.1) is such a case, albeit contrived. The abstraction excludes the analysis of scenarios in which keys might leak. Such scenarios are very relevant in practice, where a single incident of key leakage in a large system can result from:

- an attack on a specific key via exhaustive key-space exploration,
- not sufficiently tamper-resistant hardware that is used "in the field" but only holds keys of lower security levels or
- incorrect handling of key-data (for example backups) by personnel.

The proposal for a generic security API by Cortier and Steel [30], for instance, proposes a policy that limits the damage in case that a key is leaked. This security API assigns a security level $l \in \mathbb{N}$ to each handle. Handles of security level l can only be used to wrap handles of lower security levels, which enforces a so-called *strict key hierarchy*. Under these circumstances, it is possible to show that an attacker, assuming he can learn a single key of any security level l, is not able to learn any other key that has a higher or equal security

level than l. A strict key-hierarchy allows large companies to keep the costs of securing lower-level keys reasonable while at the same time containing the risk of exposing a higher-level key. Other policies that are less restrictive, but enjoy similar properties could be investigated using automated verification tools, if those tools support modelling key leakage.

In the case of abstraction by set-membership, the keys are the objects to be abstracted, since we want to have arbitrary many of them. They would be abstracted by the security level, which means that each key of the same security level falls into the same equivalence class. Therefore, if a key of level l is leaked, every key of level l is leaked (for the abstraction). The problem persists, even if in order to distinguish corrupted keys from uncorrupted keys, the modelling introduces a set for corrupted keys. If any key that is uncorrupted can become a key of this set, the abstraction introduces a set of rules permitting the substitution of every corruptible, but uncorrupted key (i. e., keys of level l) by a corrupted key. Hence, within the abstraction, the attacker would be able to decrypt any encryption with a key of level l, nor matter if this key itself was corrupted or not, which also restricts the use of this abstraction for higher-level protocols.

Sometimes the state of protocols cannot be modelled by a fixed number of sets: The Yubikey protocol introduced in Chapter 4 binds keys to a counter value to produce one-time passwords. Those protocol could not benefit from this abstraction, although we should note that, in practice, due to hardware restrictions there *is* a maximum number of states the counter can be in: 65'536. Still, this is infeasible for an analysis in this model.

Besides those two approaches, there has also been work tailored to particular applications: In 2010, Delaune et al. showed, using a dedicated hand proof, that for analyzing PKCS#11, it is possible to bound the message size [36]. Their analysis still requires to artificially bound the number of keys. Similarly in spirit, in 2011, Delaune et al. gave a dedicated result for analyzing protocols based on the TPM and its registers [35] . However, the number of reboots (which reinitialize registers) needs to be limited for this approach. Those two works use the same abstraction that StatVerif employs, but for a specific case, in other words: StatVerif is a generalisation of those approaches.

We have seen that techniques based on Horn clause resolution [14, 98] have to represent the sequence of states of a protocol through persistent facts, in order to be useful for our application, the analysis of security APIs. The abstraction employed in StatVerif restricts the size of the explicit part of the state and is therefore unsuitable for the analysis of security APIs, since our goal is to be able to deal with an unbounded number of keys. Abstraction by set-membership manages to partly overcome theses restrictions, but there are plenty of cases where this abstraction is unsuitable (see above). Our conclusion

is that Horn clause resolution is not of advantage in our case, and that we need an approach that takes into account the causality between the protocol outputs that give information to the adversary and the state transitions that allow them.

3.5 OTHER APPROACHES

In the following, we give a concise summary of other approaches we have looked at and decided not to pursue further.

The two back-ends OFMC and CL-AtSe of the Automated Validation of Internet Security Protocols and Applications (AVISPA) tool-suite [6] have support for state manipulation. Unfortunately, both require concrete bounds on the number of sessions and nonces, which contradicts our goal of an analysis for an unbounded number of keys. Although SATMC can avoid this restriction in princple [45], according to [34], SATMC performs poorly in experiments where the message length is high.

The strand space model [92] captures the causal relations between protocol input/output and adversarial deductions by modelling an execution of the protocol and the adversary as an acyclic graph. To be able to analyze optimistic fair exchange protocols, Guttman introduced an extension to the strand space model [47]. In addition to the existing kinds of nodes, modelling transmission and reception of messages, the extension adds a new kind of node which allows computations on a global state. Similar to the tamarin prover, which we will introduce in the following, the global state is modelled as a multiset of facts.

As of now, protocol analysis in the strand space model with state has not been mechanized yet. The paradigm behind verification in the original strand space model is the search for a finite number of so-called *shapes*, which are characterisations of an infinite number of protocol runs. The Cryptographic Protocol Shapes Analyzer [38] is able to reduce the set of protocol runs to a finite number of shapes. CPSA is usually able to compute a relatively small number of shapes and verify correspondence and secrecy properties on each of them. Unfortunately, CPSA does not support the additions accounting for state. It is not clear how to extend the notion of shapes to state computations, i. e., sequences of multisets.

3.6 THE TAMARIN-PROVER

In the following, we will introduce the tamarin protocol prover [89], a tool for the symbolic analysis of security protocols. The tamarin prover borrows ideas from the strand space model, but uses classical multiset rewriting for protocol description and adversarial deduction. It implements a backward induction technique that explicitly mod-

els the causality between a message deduced by the adversary and the protocol output allowing the deduction. Although the tool is relatively new, we will show why it has the potential to make fully automated verification of security APIs possible.

Protocols are specified using multiset rewrite rules, a formalism expressive enough to encode state. However, multiset rewrite rules constitute a "low-level" specification language with no direct support for concurrent message passing, making the encoding of protocols difficult and error-prone. We propose a translation from a higher-level language in Chapter 5 with the aim of mitigating this problem.

Since the underlying formalism, Multiset Rewriting (MSR), is expressive enough to encode explicit state, we are able to encode security APIs without any abstractions that enforce a monotonic state. Additionally, tamarin supports both falsification and verification of security goals that can be expressed as first-order formulas. Since the tool is sound and complete, but allows to express the secrecy problem with unbounded nonces, which is undecidable [40], there is no guarantee that the tool terminates. In order to achieve termination, some intervention is necessary: Lemmas need to be used to cut branches in the proof attempt.

Both Example 4 and Example 5 can be analyzed for an unbounded number of keys and security devices using tamarin. Furthermore, it is able to reproduce, and improve on, existing results on PKCS#11 [36] and the TPM [35] that required a bound on the number of keys, respectively the number of security devices and reboots of those devices without such restriction. In Chapter 4 we show how tamarin can be used for the analysis of an authentication protocol based on hardware-generated one-time passwords, an example that we consider to be out of the scope of the previously discussed tools. Both the case studies presented with abstraction by set-membership [74], as well as those presented with StatVerif [5] were successfully verified with the tamarin prover [71, 68, 70].

Contrary to ProVerif, tamarin sometimes requires additional *typing lemmas* which are used to guide the proof. These lemmas need to be written by hand but are proven automatically. In four out of seven case studies we needed to provide at least one such lemma manually.

3.6.1 *Labelled multiset rewriting*

We now define the syntax and semantics of labelled multiset rewrite rules, which are the input language of the tamarin tool. We largely follow the presentation by Meier, Schmidt et al. [89]. We remind the reader that we have introduced facts in Section 2.2 on page 18.

Definition 1 (Multiset rewrite rule)*:* A labelled multiset rewrite rule ri is a triple (l, a, r), $l, a, r \in \mathcal{F}^*$, written $l -[a]\rightarrow r$. We call $l = \mathit{prems}(ri)$

the premises, $a = \mathit{actions}(ri)$ the actions, and $r = \mathit{conclusions}(ri)$ the conclusions of the rule.

Example 6: The following rule has the fact $\mathit{Init}(k)$ as premise and the permanent fact (see Section 2.2) $!\mathit{Left}(k)$ as conclusion. It is part of the modelling of the Left/Right device from Example 4. This rule models that an initialized, but unconfigured security device can be put in "left" mode, identifying the device by its secret key k. The action $\mathit{SetLeft}(k)$ serves the purpose of logging that this step happened.

$$[\mathsf{Init}(k)] \,-[\mathsf{SetLeft}(k)]\!\to [!\mathsf{Left}(k)]$$

Definition 2 (Labelled multiset rewriting system)*:* A labelled multiset rewriting system is a set of labelled multiset rewrite rules R, such that each rule $(l \,-[a]\!\to r) \in R$ satisfies the following conditions:

- l, a, r do not contain fresh names
- r does not contain Fr-facts
- for each $(l' \,-[a']\!\to r') \in_E \mathit{ginsts}(l \,-[a]\!\to r)$ we have that $\cap_{r''=_E r'} \mathit{names}(r'') \cap FN \subseteq \cap_{l''=_E l'} \mathit{names}(l'') \cap FN$

The third condition ensures that protocol rules cannot create fresh names. Instead, we define one distinguished rule FRESH which is the only rule allowed to have Fr-facts on the right-hand side and to create fresh names, i. e., all fresh names originate from this rule.

$$\textsc{Fresh} : [] \,-[]\!\to [\mathsf{Fr}(x : \mathit{fresh})]$$

Example 7: The complete modelling of Example 4 consists of the following five rules:

$$\begin{aligned} P := \{ \quad [\mathsf{Fr}(k), \mathsf{Fr}(s_l), \mathsf{Fr}(s_r)] \,-[\mathsf{New}(k, s_l, s_r)]\!\to\ & \\ [\mathsf{Init}(k), \mathsf{Out}(\mathit{senc}(\langle s_l, s_r\rangle, k))], & \\ [\mathsf{Init}(k)] \,-[\mathsf{SetLeft}(k)]\!\to [!\mathsf{Left}(k)], & \\ [\mathsf{Init}(k)] \,-[\mathsf{SetRight}(k)]\!\to [!\mathsf{Right}(k)], & \\ [!\mathsf{Left}(k), \mathsf{In}(\mathit{senc}(\langle m_l, m_r\rangle, k))] \,-[\mathsf{Read}(m_l)]\!\to [\mathsf{Out}(m_l)], & \\ [!\mathsf{Right}(k), \mathsf{In}(\mathit{senc}(\langle m_l, m_r\rangle, k))] \,-[\mathsf{Read}(m_r)]\!\to [\mathsf{Out}(m_r)]\} & \end{aligned}$$

The first rule initialises a device with a fresh key and fresh secrets. The encrypted pair of secrets is output and a temporary fact marks the device as freshly initialised. The two next rules allow for setting the device to either "left" or "right" mode by replacing the temporary fact by the corresponding permanent fact. The two last rules allow for retrieving one of the two secrets if a permanent fact is present to witness that the device has been set to "left" or "right" mode.

The semantics of the rules is defined by a labelled transition relation.

Definition 3 (Labelled transition relation for multiset rewriting)*:* Given a multiset rewriting system R we define the *labeled transition relation* $\rightarrow_R \subseteq \mathcal{G}^\# \times \mathcal{P}(\mathcal{G}) \times \mathcal{G}^\#$ as

$$S \xrightarrow{a}_R ((S \setminus^\# \mathit{lfacts}(l)) \cup^\# r)$$

if and only if $l \dashv[\, a] \rightarrow\ r \in_E \mathit{ginsts}(R \cup \text{FRESH})$, $\mathit{lfacts}(l) \subseteq^\# S$ and $\mathit{pfacts}(l) \subseteq S$.

Example 8*:* Given the previous multiset rewriting system P, the following transition corresponds to setting one out of two initialised security devices to "left" mode:

$$\{\, \mathsf{Init}(k_1), \mathsf{Init}(k_2) \,\}^\# \xrightarrow{\mathsf{SetLeft}(k_2)}_P \{\, \mathsf{Init}(k_1), \mathsf{Left}(k_2) \,\}^\#$$

Definition 4 (Executions)*:* Given a multiset rewriting system R we define its set of executions as

$$\mathit{exec}^{msr}(R) = \Big\{ \emptyset \xrightarrow{A_1}_R \ldots \xrightarrow{A_n}_R S_n \mid \forall a, i, j\colon 0 \leqslant i \neq j < n.(S_{i+1} \setminus^\# S_i) = \{\mathsf{Fr}(a)\} \Rightarrow (S_{j+1} \setminus^\# S_j) \neq \{\mathsf{Fr}(a)\} \Big\}$$

The set of executions consists of transition sequences that respect freshness, i. e., for a given name a the fact $\mathsf{Fr}(a)$ is only added once, or in other words the rule FRESH fires at most once for each name.

Example 9*:* The following execution is in $\mathit{exec}^{msr}(P \cup \{\text{FRESH}\})$ (for readability, we write $\rightarrow$ instead of $\rightarrow_{P \cup \{\text{FRESH}\}}$):

$$\begin{aligned}
\emptyset &\rightarrow \{\, \mathsf{Fr}(k) \,\}^\# && (\text{FRESH})\\
&\rightarrow \{\, \mathsf{Fr}(k), \mathsf{Fr}(l) \,\}^\# && (\text{FRESH})\\
&\rightarrow \{\, \mathsf{Fr}(k), \mathsf{Fr}(l), \mathsf{Fr}(r) \,\}^\# && (\text{FRESH})\\
&\xrightarrow{\mathsf{New}(k,l,r)} \{\, \mathsf{Init}(k), \mathsf{Out}(\mathit{senc}(\langle l, r\rangle, k)) \,\}^\# && (1^{\text{st}} \text{ rule in } P)\\
&\xrightarrow{\mathsf{SetLeft}(k)} \{\, !\mathsf{Left}(k), \mathsf{Out}(\mathit{senc}(\langle l, r\rangle, k)) \,\}^\# && (3^{\text{rd}} \text{ rule in } P)
\end{aligned}$$

Definition 5 (Traces)*:* The set of traces is defined as

$$\mathit{traces}^{msr}(R) = \Big\{ [A_1, \ldots, A_n] \mid \forall\, 0 \leqslant i \leqslant n.\ A_i \neq \emptyset \text{ and } \emptyset \overset{A_1}{\Longrightarrow}_R \ldots \overset{A_n}{\Longrightarrow}_R S_n \in \mathit{exec}^{msr}(R) \Big\}$$

where $\overset{A}{\Longrightarrow}_R$ is defined as $\xrightarrow{\emptyset}{}^*_R \xrightarrow{A}_R \xrightarrow{\emptyset}{}^*_R$.

Note that the set of traces is a sequence of sets of facts.

Example 10*:* From the previous example we conclude that:

$$[\mathsf{New}(k, l, r), \mathsf{SetLeft}(k)] \in \mathit{traces}^{msr}(P \cup \{\text{FRESH}\}).$$

3.6.2 *Adversarial deduction*

To model the adversary's message deduction capabilities, we introduce the following set of rules MD:

$$\begin{aligned}
\mathsf{Out}(x) &\multimap[\quad]\to \mathsf{!K}(x) && \text{(MDOut)}\\
\mathsf{!K}(x) &\multimap[\mathsf{K}(x)]\to \mathsf{In}(x) && \text{(MDIn)}\\
&\multimap[\quad]\to \mathsf{!K}(x : \mathit{pub}) && \text{(MDPub)}\\
\mathsf{Fr}(x : \mathit{fresh}) &\multimap[\quad]\to \mathsf{!K}(x : \mathit{fresh}) && \text{(MDFresh)}\\
\mathsf{!K}(x_1), \ldots, \mathsf{!K}(x_k) &\multimap[\quad]\to \mathsf{!K}(f(x_1, \ldots, x_k)) && \\
&\qquad \text{for } f \in \Sigma^k && \text{(MDAppl)}
\end{aligned}$$

Example 11: The following execution is in $\mathit{exec}^{msr}(P \cup \{\textsc{Fresh}\} \cup MD)$ (for readability, we write $\to$ instead of $\to_{P \cup \{\textsc{Fresh}\} \cup MD}$):

$$\begin{aligned}
\emptyset &\to \{\mathsf{Fr}(k)\}^{\#} && \text{(Fresh)}\\
&\to \{\mathsf{Fr}(k), \mathsf{Fr}(l)\}^{\#} && \text{(Fresh)}\\
&\to \{\mathsf{Fr}(k), \mathsf{Fr}(l), \mathsf{Fr}(r)\}^{\#} && \text{(Fresh)}\\
&\xrightarrow{\mathsf{New}(k,l,r)} \{\mathsf{Init}(k), \mathsf{Out}(\mathit{senc}(\langle l, r\rangle, k))\}^{\#} && (1^{\text{st}} \text{ rule in } P)\\
&\xrightarrow{\mathsf{SetLeft}(k)} \{\mathsf{!Left}(k), \mathsf{Out}(\mathit{senc}(\langle l, r\rangle, k))\}^{\#} && (3^{\text{rd}} \text{ rule in } P)\\
&\to \{\mathsf{!Left}(k), \mathsf{!K}(\mathit{senc}(\langle l, r\rangle, k))\}^{\#} && \text{(MDOut)}\\
&\xrightarrow{\mathsf{K}(\mathit{senc}(\langle l,r\rangle,k))} \{\mathsf{!Left}(k), \mathsf{!K}(\mathit{senc}(\langle l, r\rangle, k)),\\
&\qquad \mathsf{In}(\mathit{senc}(\langle l, r\rangle, k))\}^{\#} && \text{(MDIn)}\\
&\xrightarrow{\mathsf{Read}(l)} \{\mathsf{!Left}(k), \mathsf{!K}(\mathit{senc}(\langle l, r\rangle, k)), \mathsf{Out}(l)\}^{\#} && (5^{\text{th}} \text{ rule in } P)\\
&\to \{\mathsf{!Left}(k), \mathsf{!K}(\mathit{senc}(\langle l, r\rangle, k)), \mathsf{!K}(l)\}^{\#} && \text{(MDOut)}\\
&\xrightarrow{\mathsf{K}(l)} \{\mathsf{!Left}(k), \mathsf{!K}(\mathit{senc}(\langle l, r\rangle, k)), \mathsf{In}(l)\}^{\#} && \text{(MDIn)}
\end{aligned}$$

Therefore, we have the following trace:

$$\begin{aligned}
(\mathsf{New}(k, l, r), \mathsf{SetLeft}(k), &\mathsf{K}(\mathit{senc}(\langle l, r\rangle, k)), \mathsf{Read}(l), \mathsf{K}(l))\\
&\in \mathit{traces}^{msr}(P \cup \{\textsc{Fresh}\} \cup MD)
\end{aligned}$$

This trace contains the action $\mathsf{K}(l)$, which is often used to witness the fact that the adversary is able to deduce a term, in this case l.

3.6.3 *Security Properties*

In the tamarin tool, security properties are described in an expressive two-sorted first-order logic. The sort *temp* is used for time points, $\mathcal{V}_{\mathit{temp}}$ are the temporal variables [89].

Definition 6 (Trace formulas)*:* A trace atom is either false $\bot$, a term equality $t_1 \approx t_2$, a time point ordering $i \lessdot j$, a time point equality $i \doteq j$, or an action $F@i$ for a fact $F \in \mathcal{F}$ and a time point i. A trace formula is a first-order formula over trace atoms.

To define the semantics, we let each sort s have a domain $\mathbf{D}(s)$. Let $\mathbf{D}(\mathit{temp}) = \mathcal{Q}$, $\mathbf{D}(\mathit{msg}) = \mathcal{M}$, $\mathbf{D}(\mathit{fresh}) = \mathit{FN}$, and $\mathbf{D}(\mathit{pub}) = \mathit{PN}$. A function $\theta : \mathcal{V} \to \mathcal{M} \cup \mathcal{Q}$ is a valuation if it respects sorts, that is, $\theta(v) \in \mathbf{D}(s)$ for all sorts s and all $v \in \mathcal{V}_s$. If t is a term, $t\theta$ is the application of the homomorphic extension of θ to t.

Definition 7 (Satisfaction relation)*:* The satisfaction relation $(\mathit{tr}, \theta) \vDash \varphi$ between trace tr, valuation θ and trace formula φ is defined as follows:

$$
\begin{array}{lll}
(\mathit{tr}, \theta) \vDash \bot & & \text{never} \\
(\mathit{tr}, \theta) \vDash F@i & \text{iff} & \theta(i) \in \mathit{idx}(tr) \text{ and } F\theta \in_E \mathit{tr}_{\theta(i)} \\
(\mathit{tr}, \theta) \vDash i \lessdot j & \text{iff} & \theta(i) < \theta(j) \\
(\mathit{tr}, \theta) \vDash i \doteq j & \text{iff} & \theta(i) = \theta(j) \\
(\mathit{tr}, \theta) \vDash t_1 \approx t_2 & \text{iff} & t_1\theta =_E t_2\theta \\
(\mathit{tr}, \theta) \vDash \neg\varphi & \text{iff} & \text{not } (\mathit{tr}, \theta) \vDash \varphi \\
(\mathit{tr}, \theta) \vDash \varphi_1 \wedge \varphi_2 & \text{iff} & (\mathit{tr}, \theta) \vDash \varphi_1 \text{ and } (\mathit{tr}, \theta) \vDash \varphi_2 \\
(\mathit{tr}, \theta) \vDash \exists x : s.\varphi & \text{iff} & \text{there is } u \in \mathbf{D}(s) \text{ such that} \\
 & & \quad (\mathit{tr}, \theta[x \mapsto u]) \vDash \varphi
\end{array}
$$

When φ is a ground formula we sometimes simply write $\mathit{tr} \vDash \varphi$ as the satisfaction of φ is independent of the valuation.

Definition 8 (Validity, satisfiability)*:* Let $\mathit{Tr} \subseteq (\mathcal{P}(\mathcal{G}))^*$ be a set of traces. A trace formula φ is said to be *valid* for Tr, written $\mathit{Tr} \vDash^\forall \varphi$, if for any trace $\mathit{tr} \in \mathit{Tr}$ and any valuation θ we have that $(\mathit{tr}, \theta) \vDash \varphi$.

A trace formula φ is said to be *satisfiable* for Tr, written $\mathit{Tr} \vDash^\exists \varphi$, if there exist a trace $\mathit{tr} \in \mathit{Tr}$ and a valuation θ such that $(\mathit{tr}, \theta) \vDash \varphi$.

Example 12*:* The following formula is valid for the trace from Example 11:

$$\exists k, s_l, s_r : \mathit{msg}, i, j : \mathit{temp}.\ \mathrm{New}(k, s_l, s_r)@i \wedge \mathrm{K}(s_l)@j,$$

since for any valuation θ such that

$$
\begin{array}{lll}
\theta(k : \mathit{msg}) = k & \theta(s_l : \mathit{msg}) = l & \theta(s_r : \mathit{msg}) = r \\
\theta(i : \mathit{temp}) = 0 & \theta(j : \mathit{temp}) = 4 &
\end{array}
$$

we have that

$$
\begin{aligned}
&((\mathsf{New}(k, l, r), \mathsf{SetLeft}(k), \mathsf{K}(\mathit{senc}(\langle l, r\rangle, k)), \mathsf{Read}(l), \mathsf{K}(l)), \theta) \\
&\qquad \vDash \exists k, s_l, s_r : \mathit{msg}, i, j : \mathit{temp}.\, \mathrm{New}(k, s_l, s_r)@i \wedge \mathrm{K}(s_l)@j.
\end{aligned}
$$

Note that $Tr \models^{\forall} \varphi$ iff $Tr \not\models^{\exists} \neg\varphi$. Given a multiset rewriting system R we say that φ is valid, written $R \models^{\forall} \varphi$, if $traces^{msr}(R) \models^{\forall} \varphi$. We say that φ is satisfied in R, written $R \models^{\exists} \varphi$, if $traces^{msr}(R) \models^{\exists} \varphi$. Similarly, given a ground process P we say that φ is valid, written $P \models^{\forall} \varphi$, if $traces^{pi}(P) \models^{\forall} \varphi$, and that φ is satisfied in P, written $P \models^{\exists} \varphi$, if $traces^{pi}(P) \models^{\exists} \varphi$.

Example 13: The following trace formula expresses the secrecy of the pair generated by the security device in Example 4:

$$\phi := \neg(\exists k, s_l, s_r : msg, i, j : temp.\ \mathrm{New}(k, s_l, s_r)@i \wedge \mathrm{K}(\langle s_l, s_r \rangle)@j)$$

The security of Example 4 is therefore expressed via the property $P \cup \{\textsc{Fresh}\} \cup MD \models^{\forall} \phi$.

Now that the notions of multiset rewrite rules and security properties of the form tamarin uses are layed out, we can proceed to the analysis of the Yubikey protocol in the next chapter.

4

ANALYSIS OF THE YUBIKEY PROTOCOL AND THE YUBIHSM

The problem of user authentication is central to computer security and of increasing importance as the cloud computing paradigm becomes more prevalent. Many efforts have been undertaken to replace or supplement user passwords with stronger authentication mechanisms [19]. The Yubikey is one such effort. Manufactured by Yubico, a Swedish company, the Yubikey itself is a low cost ($25), thumb-sized USB device. In its typical configuration, it generates a OTPs based on encryptions of a secret value, a running counter and some random values using a unique AES-128 key contained in the device. The Yubikey authentication server accepts an OTP only if it decrypts under the correct AES key to a valid secret value containing a counter larger than the last counter accepted. The counter is thus used as a means to prevent replay attacks. To date, over a million Yubikeys have been shipped to more than 30,000 customers including governments, universities and enterprises, e.g. Google, Microsoft, Agfa and Symantec [102]. Despite missing certification for governmental standards, several contractors of the U.S. Department of Defense switched to the Yubikey, after RSA's SecureID database was hacked in March 2012 [101].

Despite its widespread deployment, the Yubikey protocol has received little independent security analysis. Yubico themselves present some security arguments on their website [104]. A first independent analysis was given by blogger Fredrik Björck in 2009 [13], raising issues that Yubico responded to in a subsequent post [12]. Oswald, Richter, et al. analyse the Yubikey, version 2.0, for side-channel attacks [76]. They show that non-invasive measurements of the power consumption of the device allow retrieving the AES-key within approximately one hour of access. The authors mention a more recent version of the Yubikey, Yubikey Neo [103] which employs a CC certified smart-card controller that was designed with regard to implementation attacks and is supposed to be more resilient to power consumption analysis. Our analysis, however, is focussed on version 2.0 of the original Yubikey.

The only formal analysis of the protocol itself that we are aware of was carried out by Vamanu [96], who succeeded in showing the absence of replay attacks for an abstract version of the Yubikey OTP protocol for a bounded number of fresh OTPs. In the first section of this chapter, we give a formal model for the operation of the Yubikey OTP protocol, version 2.0, in the form of multiset rewrite rules

for tamarin, which was not available at the time of Vamanu's analysis. We prove security properties of the protocol for an unbounded number of fresh OTPs using the tamarin prover.

In the second section of this chapter, we analyze the security of the protocol with respect to an adversary that has temporary access to the authentication server. To address this scenario, Yubico offers a small Hardware Security Module (HSM) called the YubiHSM, intended to protect keys even in the event of server compromise. We show that, if the same YubiHSM configuration is used both to set up Yubikeys and run the authentication protocol, there is inevitably an attack that leaks all of the keys to the attacker. Our discovery of this attack lead to a Yubico security advisory in February 2012 [105]. In the case where separate servers are used for the two tasks, we give a configuration for which we can show (using tamarin) that if an adversary can compromise the server running the Yubikey-protocol, but not the server used to set up new Yubikeys, she cannot obtain the keys used to produce one-time passwords. The third section of this chapter evaluates our results and methodology.

All our analysis and proofs are in an abstract model of cryptography in the Dolev-Yao style and make various assumptions (that we will make explicit) about the behaviour of the Yubikey and YubiHSM. When we refer to the Yubikey, or the Yubikey protocol, we mean version 2.0 of the hardware token and the authentication protocol. We analysed version 0.9.8 beta of the YubiHSM and validated our attack on this device. According to the documentation, the model presented in Section 4.2 remains unchanged for the publicly available version 1.0 of the hardware device.

The analysis described in this chapter is joint work with Graham Steel and was published at STM 2012 [60].

4.1 YUBIKEY AND YUBIKEY AUTHENTICATION PROTOCOL

This section first discusses the Yubikey and the Yubikey authentication protocol, before describing the modelling used for security analysis.

4.1.1 *About the Yubikey Authentication Protocol*

In the remainder, we will cover the authentication protocol as it pertains to version 2.0 of the Yubikey device [93].

The Yubikey is connected to the computer via the USB port. It identifies itself as a standard USB keyboard in order to be usable *out-of-the-box* in most environments, using the operating system's native drivers. Since USB keyboards send "scan codes" rather than actual characters, the character the Yubikey transmits to the systems depends on the mapping from scan codes to characters, i. e., keyboard layout on the

system. In order to transmit the correct characters in the majority of environments, the Yubikey employs a modified hexadecimal encoding, called *modhex*, which uses characters that have the same position on many different keyboard layouts, including the French AZERTY, the German QUERTZ and the US QWERTY layout. Each keystroke carries 4 bits of information [93, Section 6.2].

The Yubikey can be configured to work in any of the following modes [93, Section 2.1]:

- *Yubikey OTP*, which is the method that is typically employed
- *OATH-HOTP*, where the OTP is generated according to the standard RFC 4226 HOTP algorithm,
- *Challenge-response mode*, where a client-side API is used to retrieve the OTP, instead of the keyboard emulation, and
- *Static mode*, where a (static) password is output instead of an OTP.

We will focus only on the Yubikey OTP mode, which we will explain in detail. Depending on the authentication module used on the server, there are four basic authentication modes [107, Section 3.4.1]:

- User Name + Password + YubiKey OTP
- User Name or YubiKey OTP + Password
- YubiKey OTP only
- User Name + Password

As the security provided by a user-chosen password is an orthogonal topic and the OTP is the main feature of the Yubikey, we will only focus on the third authentication mode.

The Yubikey has exactly one button. If this button is pressed, it emits a string via USB. The string emitted by the Yubikey is a 44-character string (i. e., 22 bytes of information in modhex encoding, see above) and consists of the unique public ID (6 bytes) and the OTP (16 bytes), encrypted under its AES key [55]. The length of the OTP is exactly the block-length of AES. It contains the following information (in order) [93, Section 6.1].

- the unique secret ID (6 bytes)
- a session counter (2 byte)
- a timecode (3 byte)
- a token counter (1 byte)
- a pseudo-random value (2 bytes)

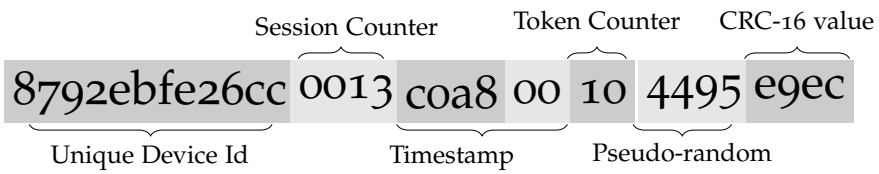


Figure 7: Structure of the OTP (session: 19, token: 16).

- a CRC-16 checksum (2 byte)

See Figure 7 for an example.

Yubico assigns an AES key and a public and secret ID to the Yubikey before shipment, but they can be overwritten. The Yubikey is *write-only* in this regard, thus it is not possible to retrieve the secret ID nor the AES key. The session counter is incremented whenever the Yubikey is plugged in. Once it reaches its limit of $2^{16} = 65536$, it cannot be used anymore. The timecode is incremented by an 8Hz internal clock. When it reaches its limit, the session is terminated, i. e., no more OTPs can be generated. This happens after approximately 24 days. The token counter is incremented whenever an OTP is generated. When it reaches its limit of 256, it restarts at 1 instead of terminating the session. The pseudo-random value of length two bytes is supposed to add entropy to the plaintext, while the Cyclic Redundancy Check (CRC) is supposed to detect transmission errors. It does not provide cryptographic integrity.

A Yubikey stores public and secret ID *pid* and *sid*, and the AES key k, and is used in the following authentication protocol: The user provides a client C with the Yubikey's output *pid*, *otp*,e. g., by filling it in a web form. The infix operator "||" denotes concatenation.

$$\begin{aligned} C \to S&: \ pid \,||\, otp \,||\, nonce \\ S \to C&: \ otp \,||\, nonce \,||\, hmac \,||\, status \end{aligned}$$

Here *nonce* is a randomly chosen value between 8 and 20 bytes and *hmac* is a Message Authentication Code (MAC) over the parameters using a key present on the server and the client. By *status*, we denote additional status information given in the response, containing an error code that indicates either success or where the verification of the OTP failed, plus (in case of success) the value of the internal timestamp, session counter and token counter when the key was pressed and more [94].

The server S accepts the token if and only if either the session counter is bigger than the last one received, or the session counter has the same value but the token counter is incremented. It is possible to verify if the timestamp is in a certain window with respect to the previous timestamp received, however, our model does not include the timing of messages, therefore we ignore this (optional) check.

4.1.2 *Formal Analysis (Uncompromised Server)*

We are using tamarin for the analysis because it supports the modelling of explicit state, which is an important part of this protocol. Consider, for example, that the last counter received from some Yubikey is saved on the server. The protocol relies on the fact that once an OTP with a counter value has been accepted, the last counter value is updated. Certain OTP values that would have been accepted before will be rejected from this moment on. We argued in Chapter 3, specifically Section 3.4, why this kind of "non-monotonicity" does not work well with many abstractions that are used in the context of automated protocol verification. As we will show in the course of the next chapters, tamarin allows for the analysis of this protocol.

An important part of the modelling of the protocol is to determine whether one counter value is smaller than another. To this end, our modelling employs a feature added to the development version of tamarin as of October 2012, a union operator $\cup^{\#}$ for multisets of message terms. The operator is denoted with a plus sign ("+") in tamarin; to avoid confusion we will use the same notation in the multiset rewriting calculus. We model the counter as a multiset only consisting of the symbol "one". The cardinality of the multiset is the value of the counter. A counter value is considered smaller than another one, if the first multiset is included in the second. We enforce those semantics by adding an axiom that requires, for all instantiations of rules annotated with $\mathsf{Smaller}(a, b)$, that a is a subset of b:

$$\forall i : \mathit{temp}, a, b : \mathit{msg}.\ \mathsf{Smaller}(a, b)@i \Rightarrow \exists z : \mathit{msg}.a + z = b$$

Recall that i is a time-points and $\mathsf{Event}@i$ means that the trace contains a rule instantiation that produces the action Event at time point i.

We had to simplify the modelling of the session and token counter: Instead of having two counters, we just model a single counter. Since the Yubikey either increases the session counter and resets the token counter, or increases the token counter and leaves the session counter untouched, it implements a complete lexicographical order on the pair (session counter, token counter). Since the authentication server accepts an OTP if either the session counter is bigger than the last one received, or the session counter has the same value but the token counter is incremented, we let it accept the single counter if it is larger than previously received counter.

The following rule models the initialisation of a Yubikey. A fresh public ID (*pid*), secret ID (*sid*) and Yubikey-key (k) are drawn, and saved on the server and the Yubikey.

$$\begin{aligned}&\mathsf{Fr}(k), \mathsf{Fr}(pid), \mathsf{Fr}(sid)\\ &-[\ \mathsf{Protocol}(), \mathsf{Init}(pid, k), \mathsf{ExtendedInit}(pid, sid, k)]\rightarrow\\ &\quad !\mathsf{Y}(pid, sid), \mathsf{Y\_counter}(pid, \ '1'), \mathsf{Server}(pid, sid, \ '1'),\\ &\quad\ !\mathsf{SharedKey}(pid, k)\end{aligned}$$

The Fr-facts guarantee that each instantiation of this rule replaces k, *pid* and *sid* by different fresh names. The Yubikey is identified by its public ID, thus the permanent fact !Y stores the corresponding secret ID, and !SharedKey the corresponding key k, which is shared with the server, i. e., rules both on the side of the authentication server and the Yubikey make use of this fact. Y_counter is the current counter value stored on the Yubikey. Server models the table on the server-side, storing which Yubikey is associated with which secret ID and the value of the last counter received.

The next rule models how the counter is increased when a Yubikey is plugged in. As mentioned before, we model both the session and the token counter as a single counter. We over-approximate in the case that the Yubikey increases the session token by allowing the adversary to instantiate the rule for any counter value that is higher than the previous one, using the Smaller action.

$$\begin{aligned}&\mathsf{Y\_counter}(pid, otc), \mathsf{In}(tc) \ -[\ \mathsf{Yubi}(pid, tc), \mathsf{Smaller}(otc, tc)]\rightarrow\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Y\_counter}(pid, tc)\end{aligned}$$

Note that the adversary has to input *tc*. We can only express properties about the set of traces in tamarin, e. g., the terms the adversary constructs in a given trace, but not the terms she *could* construct in this trace. By requiring the adversary to produce all counter values, we can ensure that they are in !K, i. e., the adversary's knowledge.

When the button is pressed, an encryption is output in addition to increasing the counter:

$$\begin{aligned}&!\mathsf{Y}(pid, sid), \mathsf{Y\_counter}(pid, tc), !\mathsf{SharedKey}(pid, k), \mathsf{In}(tc), \mathsf{Fr}(npr),\\ &\quad \mathsf{Fr}(nonce)\\ &-[\ \mathsf{YubiPress}(pid, tc)]\rightarrow\\ &\quad \mathsf{Y\_counter}(pid, tc +' 1'), \mathsf{Out}(\langle pid, nonce, senc(\langle sid, tc, npr\rangle, k)\rangle)\end{aligned}$$

The output can be used to authenticate with the server, in case that the counter inside the encryption is larger than the last counter stored on the server:

$$\begin{aligned}&\mathsf{Server}(pid, sid, otc), \mathsf{In}(\langle pid, nonce, otp\rangle), !\mathsf{SharedKey}(pid, k), \mathsf{In}(otc)\\ &-[\mathsf{Login}(pid, sid, tc, otp), \mathsf{LoginCounter}(pid, otc, tc),\\ &\quad \mathsf{Smaller}(otc, tc) \qquad\qquad]\rightarrow\\ &\quad\quad \mathsf{Server}(pid, sid, tc)\end{aligned}$$

for $otp = senc(\langle sid, tc, pr\rangle, k)$. Tamarin is able to prove the following properties.

1. The absence of replay attacks:

$$\begin{aligned}&\neg(\exists i, j, pid, sid, x, otp_1, otp_2.\\ &\quad \mathsf{Login}(pid, sid, x, otp_1)@\mathsf{i} \wedge\ \mathsf{Login}(pid, sid, x, otp_2)@\mathsf{j}\\ &\quad \wedge \neg(i = j)).\end{aligned}$$

   There are no two distinct logins that accept the same counter value. Note that this property is stronger than showing that no two distinct logins for the same OTP exists, as two OTPs that are equal necessarily contain the same counter value.

2. Injective correspondence between pressing the button on a Yubikey and a successful login:

$$\begin{aligned}&\forall pid, sid, x, otp, t_2.\\ &\quad \mathsf{Login}(pid, sid, x, otp)@\mathsf{t}_2 \Rightarrow\\ &\qquad \exists t_1.\mathsf{YubiPress}(pid, x)@\mathsf{t}_1 \wedge \mathsf{t}_1 \lessdot \mathsf{t}_2\\ &\qquad\quad \wedge \forall otp_2, t_3.\mathsf{Login}(pid, sid, x, otp_2)@\mathsf{t}_3 \Rightarrow \mathsf{t}_3 = \mathsf{t}_2\end{aligned}$$

   A successful login must have been preceded by a button press for the same counter value. Furthermore, there is not second, distinct login for this counter value.

3. The fact that the counter values associated to logins are monotonically increasing in time, which implies that a successful login invalidates previously collected OTPs.

$$\begin{aligned}&\forall pid, otc_1, tc_1, otc_2, tc_2, t_1, t_2, t_3.\ \mathsf{Smaller}(tc_1, tc_2)@\mathsf{t}_3\\ &\quad \wedge \mathsf{LoginCounter}(pid, otc_1, tc_1)@\mathsf{t}_1\\ &\quad \wedge \mathsf{LoginCounter}(pid, otc_2, tc_2)@\mathsf{t}_2\\ &\qquad \Rightarrow \mathsf{t}_1 \lessdot \mathsf{t}_2\end{aligned}$$

The absence of replay attacks is proven by showing the following, stronger property, which we use as an invariant:

$$\begin{aligned}&\forall pid, otc_1, tc_1, otc_2, tc_2, t_1, t_2.\ \mathsf{LoginCounter}(pid, otc_1, tc_1)@\mathsf{t}_1\\ &\quad \wedge \mathsf{LoginCounter}(pid, otc_2, tc_2)@\mathsf{t}_2 \wedge \mathsf{t}_1 \lessdot \mathsf{t}_2\\ &\qquad \Rightarrow \exists z.tc_2 = z + tc_1\end{aligned}$$

Intuitively, this means that counter values are strictly increasing in time. If counter values are increasing strictly in time, any two logins with the same counter value are in fact the same, i. e., there are no replay attack. Tamarin is able to prove the invariant, as well as the security properties completely automatically. Note that in this model, the adversary has no direct access to the server, she can only control the network (as well as the Yubikey according to its specification). A stronger attack model is discussed in the next section. For the complete input to tamarin, including the multiset rewrite rules constituting the model, as well as the security properties and helping lemmas, see Listing 17 in Section A.2. All source files and proofs are available online, too [59].

4.2 THE YUBIHSM

The results from the previous chapter assume that the authentication server remains secure. Unfortunately, such servers are valuable targets, and consequently successful attacks occur from time to time – as in the case of the RSA Security, Inc. (RSA) SecurID system where attackers were able to compromise the secret seed values stored on the server and thereby fake logins for sensitive organisations such as Lockheed Martin [56]. RSA now use an HSM to protect seeds in the event of server compromise. Yubico also offer (and use themselves) an application specific HSM, the YubiHSM to protect the Yubikey AES keys in the event of an authentication server compromise by encrypting them under a master key stored inside the YubiHSM. In the second part of our chapter, we analyse the security of the YubiHSM API. First we show that due to an apparent oversight in the cryptographic design, an attacker with access to the server where Yubikey AES keys are generated is able to decrypt the encrypted keys and obtain them in clear. We then prove secrecy of keys in various configurations of YubiHSMs and servers, and suggest design changes that would allow a single server to be used securely.

4.2.1 *About the YubiHSM*

The YubiHSM is also a USB device about 5 times thicker than a Yubikey. According to Yubico, it "provides a low-cost [$500] way to move out sensitive information and cryptographic operations away from a vulnerable computer environment without having to invest in expensive dedicated Hardware Security Modules (HSMs)" [106]. The YubiHSM stores a very limited number of AES keys in a way that the server can use them to perform cryptographic operations without the key values ever appearing in the server's memory. These 'master keys' are generated during configuration time and can neither be modified nor read at runtime. The master keys are used to

encrypt working keys which can then be stored safely on the server's hard disk. The working keys are encrypted inside so-called AEADs (blocks produced by authenticated encryption with associated data). In order to produce or decrypt an AEAD, an AES key and a piece of associated data is required. The YubiHSM uses CCM mode to obtain an AEAD algorithm from the AES block cipher [99].

In the case of the Yubikey protocol, AEADs are used to store the keys the server shares with the Yubikey tokens, and the associated data is the public ID and the key-handle used to reference the AES key. The idea here is that since the master keys of the YubiHSM cannot be extracted, the attacker never learns the value of any Yubikey AES keys, even if she successfully attacks the server. While she is in control of the server, she is (of course) able to grant or deny authentication to any client at will. However, if the attack is detected and the attacker loses access to the server, it should not be necessary to replace or rewrite the Yubikeys that are in circulation in order to re-establish security.

4.2.2 *Two Attacks on the Implementation of Authenticated Encryption*

The YubiHSM provides access to about 22 commands that can be activated or de-activated globally, or per key, during configuration. We first examined the YubiHSM API in its default configuration, discovering the following two attacks which led to a security advisory issued by Yubikey in February 2012 [105].

The attacks use the following two commands:

- AES_ECB_BLOCK_ENCRYPT takes a handle to an AES key and a plaintext of length of one AES block (16 Bytes) and applies the raw block cipher.
- YSM_AEAD_GENERATE takes a nonce, a handle to an AES key and some data and outputs an AEAD. More precisely, but still simplified for our purposes, it computes:

$$AEAD(nonce,kh,data) = \left(\left(\Big\|_{i=0}^{\lceil \frac{|data|}{blocksize} \rceil} AES(k, counter_i)\right) \bigoplus data\right) \| mac$$

  where k is the key referenced by the key-handle kh, $counter_i$ is a counter that is completely determined by $kh, nonce$, i and the length of *data* and *blocksize* is 16 bytes. For the precise definition of *mac* and *counter*, we refer to Request For Comments (RFC) 3610 [99].

Figure 8 depicts the counter mode of operation, used to calculate the ciphertext body of the AEAD to which the MAC will be appended.

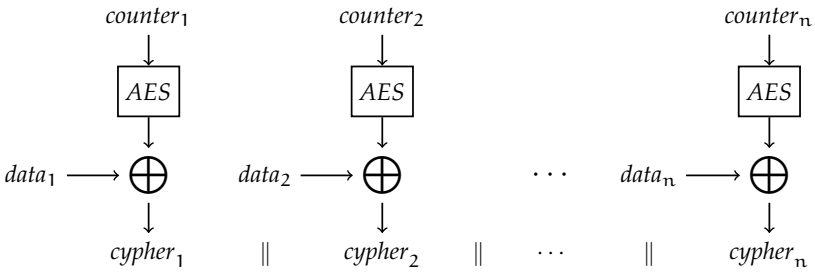


Figure 8: AES in counter mode (simplified).

The AEADs used to store keys for decrypting OTPs in the Yubikey protocol are special cases: The plaintext is a concatenation of the respective Yubikey's AES key and secret device ID (22 bytes in total), and *nonce* consists of the Yubikey's public ID.

An attacker with access to the command AES_ECB_BLOCK_ENCRYPT is able to decrypt an AEAD by recreating the blocks of the key-stream, i. e., $AES(k, counter_i)$. She xors the result with the AEAD truncated by 8 bytes (the length of *mac*) and yields *data*. When the attacker is able to compromise the server, she learns the AEAD and the key-handle used to produce it. Since the nonce is the public ID of the Yubikey, she can compute $counter_i$ and, using AES_ECB_BLOCK_ENCRYPT the key-stream. It is in the nature of the counter-mode that encryption and decryption are the same operation. According to the reference manual[106, Section 4.3], "the YubiHSM intentionally does not provide any functions [sic] that decrypts an AEAD and returns it in clear text, either fully or partial [sic].". We therefore consider the protection of the AEAD's contents a security goal of the YubiHSM, which is violated by this attack. The attack can be prevented by disabling the AES_ECB_BLOCK_ENCRYPT command on the relevant key handles at configuration time.

The second attack uses only YSM_AEAD_GENERATE: An attacker can produce $AEAD(nonce, kh, 0^l)$ for the same handle *kh* and a value *nonce* that was previously used to generated another AEAD of length l. This way, an attacker can recover the key-stream directly (discarding *mac*). Once again, it is possible to decrypt AEADs. This attack is worse than the first one, because YSM_AEAD_GENERATE is necessary for the set-up of Yubikeys. Note that the attack applies also to the commands YSM_RANDOM_AEAD_GENERATE and YSM_BUFFER_AEAD_GENERATE [106, p. 28-29].

This second attack is harder to prevent, since in order to set up Yubikeys with their AES keys, the YSM_AEAD_GENERATE command must be enabled at some point. The security advisory suggests that the threat can be "mitigated by observing that a YubiHSM used to generate AEADs is guarded closely to not permit maliciously crafted input." In the next section, we try to interpret this advice into a concrete configuration for which we can prove security of the sensitive

keys. Then, in section 4.3, we describe practical ways in which this configuration could be used.

Remark 1: Bellare et al. gave a proof of security for counter mode[10]: Under the assumption that AES constitutes are pseudo-random permutation, this mode of operation should transform the block cipher AES into a secure encryption scheme. The flaw lies in the implementation: The YubiHSM allows an arbitrary nonce to be supplied as Initialisation Vector (IV) for the counter. The security relies on the fact that those nonce values are unique: Bellare et al. propose counter mode as a statefull encryption scheme that increments the nonce value for every produced cipher-text. The uniqueness of the nonces cannot be assured neither in the case that AES_ECB_BLOCK_ENCRYPT is available, nor in the case where YSM_AEAD_GENERATE is available.

4.2.3 *Analysis in the Case of Server Compromise*

From now on, we will assume the following corruption scenario: In addition to the capacities described in Section 4.1.2, the attacker can read the AEADs stored on the server and she can access the HSM. Every AEAD is created using the same key on the HSM, the handle to this key is made public. The public ID is given to the adversary when a Yubikey is set up. Counter values are deducible, so there is no need to give the adversary explicit access to this data. The adversary is still not able to directly read the data stored on the Yubikey or YubiHSM. Note that in this situation, the attacker can trivially approve or deny authorisation requests to the server, hence we cannot expect to show absence of replay attacks. We are rather interested in whether the attacker can recover the secret keys used to create OTPs, which would allow her to continue to obtain authorisation even once she is denied access to the server.

We model the xor operator in a simplified manner. The equational theory we employ allows for rediscovering the two attacks described in Section 4.2.2, but it does not capture all attacks that xor might permit in this context. For this reason, the positive security results in this section have to be taken with caution. We model xor with the function symbols *xor* and the following five equations:

$$xor(xor(a,b),a) = b, \qquad xor(0,a) = a,$$
$$xor(xor(a,b),b) = a, \qquad xor(a,0) = a, \text{ and}$$
$$xor(a,a) = 0.$$

These equations ignore commutativity and associativity of xor. Using this equational theory, we are able to rediscover the attacks described in the previous section. The current version of tamarin (0.8.2.1) does not have built-in support yet, but it is planned for future releases.

Remark 2: There is a technique by Küsters and Truderung which reduces protocol analysis including xor to the xor-free case using an

approach based on Horn clauses [85]. This technique cannot be employed here, since the protocol contains steps where xor is applied to two terms which both contain variables, see for example the rules for YSM_AEAD_YUBIKEY_OTP_DECODE below, i. e., the protocol is not $\oplus$-linear in the sense of their work.

The counter values are modelled as before. We initialise the YubiHSM with exactly one key-handle:

$$\mathsf{Fr}(k), \mathsf{Fr}(kh) \multimap[\ \mathsf{MasterKey}(k), \mathsf{OneTime}()]\to !\mathsf{HSM}(kh, k), \mathsf{Out}(kh), \\ !\mathsf{YSM\_AEAD\_YUBIKEY\_OTP\_DECODE}(kh)$$

We make sure that this rule is only instantiated once by adding a corresponding axiom $\forall i, j.\ \mathsf{OneTime}()@i \wedge \mathsf{OneTime}()@j \Rightarrow i = j$.

The following rules model the fact that the adversary can communicate with the YubiHSM

$$\mathsf{OutHSM}(x) \multimap[\mathsf{HSMRead}(x)\ ]\to [\mathsf{Out}(x)] \\ \mathsf{In}(x) \multimap[\mathsf{HSMWrite}(x)]\to [\mathsf{InHSM}(x)]$$

and can read the list of AEADs stored on the authentication server.

$$!\mathsf{S\_AEAD}(pid, aead) \multimap[\mathsf{AEADRead}(aead), \mathsf{HSMRead}(aead)]\to \\ [\mathsf{Out}(aead)]$$

The next rules aim at modelling the HSM. We defined a set of 4 rules in total, but only YSM_AEAD_YUBIKEY_OTP_DECODE is actually used: The initialisation rule produces the permanent fact !YSM_AEAD_YUBIKEY_OTP_DECODE, but not the fact required by the other rules. This way, we model a configuration that allows only the execution of YSM_AEAD_YUBIKEY_OTP_DECODE while the authentication server might be compromised. The rule YSM_AEAD_GENERATE is deactivated in this sense, but it is directly incorporated into the rule BuyANewYubikey, see below. This models a configuration where YSM_AEAD_GENERATE is active during setup, but only during setup.

$$\mathsf{InHSM}(\langle did, kh, aead, otp\rangle), !\mathsf{HSM}(kh, k), \\ !\mathsf{YSM\_AEAD\_YUBIKEY\_OTP\_DECODE}(kh) \\ \multimap[\ \mathsf{OtpDecode}(k_2, k, \langle did, sc, r\rangle, sc, xor(senc(ks, k), \langle \mathsf{k}_2, did\rangle), mac) \\ \mathsf{OtpDecodeMaster}(k_2, k)]\to \quad \mathsf{OutHSM}(sc)$$

where

$$ks = \mathsf{keystream}(kh, N), \\ mac = \mathsf{mac}(\langle k_2, did\rangle, k), \\ aead = \langle \mathsf{xor}(senc(ks, \mathsf{k}), \langle \mathsf{k}_2, did\rangle), mac\rangle, \text{ and} \\ otp = \mathsf{senc}(\langle did, sc, r\rangle, k_2).$$

The rules for emitting the OTP and the login are modelled in a way that is very similar to the rules described in Section 4.1.2, but of course we model the encryption used inside the AEAD in more detail. Here, the server-side rule for the login.

$$\begin{aligned}&\mathsf{In}(\langle pid, nonce, senc(\langle sid, tc, pr\rangle, k_2)\rangle), !\mathsf{HSM}(kh, k),\\ &\quad !\mathsf{S\_sid}(pid, sid), !\mathsf{S\_AEAD}(pid, aead), \mathsf{S\_Counter}(pid, otc)\\ &\multimap[\ \mathsf{Login}(pid, tc, senc(\langle sid, tc, pr\rangle, k_2)), \mathsf{Smaller}(otc, tc)]\rightarrow\\ &\quad \mathsf{S\_Counter}(pid, tc)\end{aligned}$$

where ks, *mac* and *aead* are defined as before.

Tamarin is able to prove that, within our limited model of xor, the adversary never learns a Yubikey AES key or a YubiHSM master key - in other words, AEAD, as well as the key used to produce them, stay confidential. The proof does not need human intervention, however, some additional typing invariants are needed in order to reach termination. For instance, the following invariant is used to proof that a key k_2 shared between the authentication server and the Yubikey can only be learned when the key k, used to encrypt the AEADs, is leaked.

$$\begin{aligned}&\forall t_1, t_2, pid, k_2.\ \mathsf{Init}(pid, k_2)@t_1 \wedge \mathsf{K}(k_2)@t_2\\ &\quad \Rightarrow \exists t_3, t_4, k.\ \mathsf{K}(k)@t_3 \wedge \mathsf{MasterKey}(k)@t_4 \wedge t_3 \lessdot t_2\end{aligned}$$

For the complete input to tamarin, including the multiset rewrite rules constituting the model, as well as the security properties and helping lemmas, see Listing 18 in Section A.2.

4.3 EVALUATION

The positive and negative results in this chapter provide formal criteria to evaluate the security of the Yubikey protocol in different scenarios.

4.3.1 *Positive Results*

Under the assumption that the adversary can control the network, but is not able to compromise the client or the authentication server, we have shown that she cannot mount a replay attack. Furthermore, if a YubiHSM is configured such that YSM_AEAD_YUBIKEY_OTP_-DECODE is the only available command, then even in case the adversary is able to compromise the server, the Yubikey AES keys remain secure. All these results are subject to our abstract modelling of cryptography and the algebraic properties of xor.

Since the Yubikeys need to be provisioned with their AES keys and secret identities must be stored in the AEADs, we propose two set-ups that can be used to obtain the configuration used in the analysis:

1. *One Server, One YubiHSM:* There is a set-up phase which serves the purpose of producing AEADs (using one of the AEAD generation commands, e. g., YSM_AEAD_GENERATE) and writing the key and secret/public identity on the Yubikey. This phase should take place in a secure environment. Afterwards, the YubiHSM is returned to configuration mode and all commands are disabled except YSM_AEAD_YUBIKEY_OTP_DECODE. In this set-up, only one YubiHSM is needed, but it is not possible to add new Yubikeys once the second phase has begun without taking the server off-line and returning the YubiHSM to configuration mode. Note that switching the YubiHSM into configuration mode requires physical access to the device, hence would not be possible for an attacker who has remotely compromised the server.

2. *Two Servers, Two YubiHSMs:* There is one server that handles the authentication protocol, and one that handles the set-up of the Yubikeys. The latter is isolated from the network and only used for this very purpose, so we consider it a secure environment. We configure two YubiHSMs such that they store the same master-key (the key used to produce AEADs). The first is used for the authentication server and has only YSM_AEAD_YUBIKEY_OTP_DECODE set to true, the second is used in the set-up server and has only YSM_AEAD_GENERATE set to true. The set-up server produces the list of public ids and corresponding AEADs, which is transferred to the authentication server in a secure way, for example in fixed intervals (every night) using fresh USB keys. The transfer does not necessarily have to provide integrity or secrecy (as the adversary can block the authentication via the network, anyway), but it should only be allowed in one direction.

Reading between the lines (since no YubiHSM configuration details are given) it seems that Yubico themselves use the second set-up to provision Yubikeys [100].

4.3.2 *Negative Results*

In case either of the permissions AES_ECB_BLOCK_ENCRYPT or YSM_AEAD_GENERATE are activated on a master key handle (by default both are), the YubiHSM does protect the keys used to produce one-time passwords encrypted under that master key. Since YSM_AEAD_GENERATE (or YSM_BUFFER_AEAD_GENERATE) are necessary in order to set up a Yubikey, this means that separate setup and authorisation configurations have to be used in order to benefit from the use of the YubiHSM, i. e., have a higher level of security than in the case where the keys are stored unencrypted on the hard

disk. Unfortunately, open source code available on the web in e.g. the yhsmpam project [48], designed to use the YubiHSM to protect passwords from server compromise, uses the insecure configuration, i.e. one YubiHSM with both YSM_AEAD_GENERATE and (in this case) YSM_AEAD_DECRYPT_CMP enabled, and hence would not provide the security intended.

4.3.3 *Possible changes to the YubiHSM*

We will now discuss two possible counter-measures against this kind of attack that could be incorporated into future versions of the YubiHSM to allow a single device to be used securely, the first which may be seen as a kind of stop-gap measure, the second which is a more satisfactory solution using more suitable crypto:

1. *AEAD_GENERATE with a randomly drawn nonce:* All three YubiHSM commands to generate AEADs, i. e.,
   - YSM_AEAD_GENERATE,
   - YSM_BUFFER_AEAD_GENERATE and
   - YSM_RANDOM_AEAD_GENERATE,

   let the user supply the nonce that is used as Initialisation Vector (IV). This would not be possible if they were replaced by a command similar to YSM_AEAD_GENERATE that chooses the nonce randomly and outputs it at the end, so it is possible to use the nonce as the public ID of the Yubikey. However, even in this case there is an online guessing attack on the HSM: An AEAD can be decrypted if the right nonce is guessed. We can assume that the adversary has gathered a set of honestly generated OTPs, so she is able to recognize the correct nonce. Since the nonce space is rather small (2^{48}) in comparison to the key-space of AES-128, the adversary could perform a brute-force search. We measured the amount of time it takes to perform 10 000 YSM_AEAD_GENERATE operations on the YubiHSM. The average value is 0.2178 ms, so it would take approximately 1900 years to traverse the nonce space. Even so this is not completely reassuring.

2. *SIV-mode:* The Synthetic Initialization Vector (SIV) mode of operation [86] is designed to be resistant to repeated IVs. It is an authenticated encryption mode that works by deriving the IV from the MAC that will be used for authentication. As such it is deterministic - two identical plaintexts will have an identical ciphertext - but the encryption function cannot be inverted in the same way as Counter mode of operation with CBC-MAC (CCM) mode by giving the same IV (the encryption function does not take an IV as input, the only way to force the same IV to be used is to give the same plaintext). This would seem to suit the

requirements of the YubiHSM very well, since it is only keys that will be encrypted hence the chances of repeating a plaintext are negligible. Even if the same key is put on two different Yubikeys, they will have different private IDs yielding different AEADs. In our view this is the most satisfactory solution.

4.3.4 *Methodology*

Since the tamarin prover could not derive the results in this chapter without auxiliary lemmas, we think it is valuable to give some information about the way we derived those results.

We first modelled the protocol using multiset rewrite rules, which was a relatively straight-forward task. However, note that our model consists of only four rules, and is therefore quite coarse. In particular, since, e. g., the emission of a Yubikey, and the authentication by the server, e. g., are modelled using a single rule each, we assume those steps to be atomic. This idealisation seems common for multiset rewrite models, but is somewhat unrealistic, as it removes a lot of potential scheduling problems from the model.

After we modelled the protocol, we stated a sanity lemma saying "There does not exist a trace that corresponds to a successful protocol run" to verify that our model is sane. Tamarin should be able to derive a counter-example, which is a proof that a correct protocol run is possible.

We stated the security property we wanted to prove, e. g., the absence of replay attacks. Since tamarin did not produce a proof on its own, we investigated the proof derivation in interactive mode. We deduced lemmas that seemed necessary to cut branches in the proof that are looping, so-called typing invariants. An example is YSM_AEAD_YUBIKEY_OTP_DECODE: It outputs a subterm of its input, namely the counter value. Whenever tamarin tries to derive a term t, it uses backward induction to find a combination of AEAD and OTP that allows to conclude that the result of this operation is t. Meier, Cremers and Basin propose a technique they call *decryption-chain reasoning* in [73, Section 3b] that we used to formulate our typing invariant. Once the invariant is stated, it needs to be proven. Sometimes the invariant depends on another invariant, that needs to be found manually. Finding the right invariants took a bit of time, and to a certain degree, was a trial and error task, as we did not have much experience with tamarin. We eventually found a set of invariants that lead to a successful verification of the security property, and then minimised this set of lemmas by deleting those that were not necessary.

All in all, it took about 1 month to do the analysis presented in this work. The modelling of the protocol took no more than half a day. Finding a first modelling of the natural numbers and the "Is smaller than" relation suitable for analysis in tamarin took a week, since at

this time, the multiset union operator was not available in tamarin. We employed a modelling that builds the "Is smaller than" as a set of permanent facts from an "Is Successor" of relation:

$$\mathsf{In}(o), \mathsf{In}(S(o)) \; -[\mathsf{IsSucc}(o, S(o)), \mathsf{IsZero}(o)]\!\rightarrow \; !\mathsf{Succ}(o, S(o))$$

$$\mathsf{In}(y), \mathsf{In}(S(y)), \; !\mathsf{Succ}(x, y) \; -[\mathsf{IsSucc}(y, S(y))\;]\!\rightarrow \; !\mathsf{Succ}(y, S(y))$$

$$!\mathsf{Succ}(x, y) \; -[\mathsf{IsSmaller}(x, y)\;]\!\rightarrow \; !\mathsf{Smaller}(x, y)$$

$$!\mathsf{Smaller}(x, y), \; !\mathsf{Succ}(y, z) \; -[\mathsf{IsSmaller}(x, z)\;]\!\rightarrow \; !\mathsf{Smaller}(x, z)$$

An additional axiom was needed to enforce transitivity. Using this modelling, it was also possible to derive all results covered in this chapter. We consider the modelling using the multiset data type more natural, whereas this modelling does not rely on the support of associativity, commutativity and a neutral element in equational theories, as is needed to model multisets with the empty multiset and multiset union. It might be interesting for similar use cases with other tools that may not support such equational theories. The complete input to tamarin, including the multiset rewrite rules constituting the model described, as well as the security properties, helping lemmas and additional axioms can be found in Listing 19 (for the case without server compromise) and Listing 20 (for the case where the server might be compromised, but a YubiHSM is protecting the key) in Section A.2.

The lion's share of the time was spent in searching the right invariants for termination. The running time of tamarin is acceptable: Proving the absence of replay attacks in case of an uncompromised server takes around 35 seconds, proving confidentiality of keys in the case of a compromised server takes around 50 seconds, both on a 2.4GHz Intel Core 2 Duo with 4GB RAM.

4.3.5 *Future work*

We learned that it is possible to obtain formal results on the YubiKey and YubiHSM for an unbounded model using tamarin. However, there are plenty of improvements to be made: In the model presented in this chapter, we treat the session and token counter on the Yubikey as a single value, which is justifiable by the fact that the Yubikey either increases the session counter and resets the token counter, or increases the token counter, thereby implementing a complete lexicographical order on the pair (session counter, token counter). Nevertheless, a more detailed model would not rely on this argument. We furthermore omit the CRC and the time-stamp, since in the symbolic setting, they do not add to the security of the protocol.

Secondly, we simplified the algebraic theory of xor considerably. The treatment of xor is in discussion for future versions of the tamarin tool.

As stated in Section 4.3.4, finding the right lemmas to prove the security properties was a huge part of the analysis. A methodology for deriving the lemmas that tamarin needs to obtain a proof automatically, could permit more automation and hence allow for the analysis of larger models.

5 A PROCESS CALCULUS WITH STATE

The applied pi calculus [2] has been very successful in the domain of protocol verification, as it allows an intuitive modelling of protocols with an emphasis on the most important part: the communication between processes. It is sufficiently abstract to allow for automated analysis, while at the same time it is sufficiently concrete to allow refinement into a working implementation that actually benefits from the protocol analysis – ideally, this refinement can be proven to carry the security results from the analysis to the implementation, as demonstrated by Pironti et al. [79, 80]. Even if the refinement may not be provably correct, a semantically rich model helps minimizing the "degrees of freedom" of such a refinement and thus helps establishing confidence in the implementation.

In our opinion, the formalism of StatVerif [5] has those two qualities. Unfortunately, as shown in Chapter 3, the translation from StatVerif's calculus into Horn clauses suffers some limitations with respect to the application of security APIs. Additionally, there are some limitations in the calculus itself, such as the limited amount of cells, and the limited use of locks. In the last chapter, we have seen how multiset rewriting can be used for the analysis of security APIs, although it requires a certain amount of effort to derive a good modelling. Another disadvantage is that it is quite hard to see whether the set of multiset rewrite rules describing the protocol and an implementation of the protocol do the same thing. The representation of protocols as multiset rewrite rules is very low level and far away from actual protocol implementations, making it difficult to model protocols correctly. Encoding private channels, nested replications and locking mechanisms directly as multiset rewrite rules is a tricky and error prone task. Protocol steps are often represented as a single rule, making them effectively atomic. This obscures eventual issues in concurrent protocol steps, that would otherwise require, e. g., explicit locking, thus increasing the risk of implicitly excluding attacks in the model that are well possible in a real implementation, e. g., race conditions. This chapter aims at combining the advantages of an applied pi-like calculus similar to StatVerif with the constraint solving algorithm employed by the tamarin-prover.

In this chapter, we propose a tool for analyzing protocols that may involve non-monotonic global state. The tool's input language, described in Section 5.2, is a variant of the applied pi calculus with additional constructs for manipulating state. The heart of our tool is a translation from this calculus to a set of multiset rewrite rules

that can then be analyzed by the tamarin-prover (see Section 3.6) as a back-end. We introduce this translation in Section 5.3. We prove the correctness of this translation in Section 5.4 and show that it preserves all properties expressible in a dedicated first order logic for expressing security properties. In Section 5.5, we illustrate the tool on several case studies: a simple security API in the style of PKCS#11, a complex case study of the Yubikey security device, as well as several examples analyzed by other tools that aim at analyzing stateful protocols. In all of these case studies we were able to avoid restrictions that were necessary in previous works.

5.1 RELATED WORK

The most closely related work is the StatVerif tool by Arapinis et al. [5]. They propose a stateful extension of the applied pi calculus, similar to ours, which is translated to Horn clauses and analyzed by the ProVerif tool. Their translation is sound but may report false attacks, which limits the scope of protocols that can be analyzed. Moreover, StatVerif can only handle a finite number of memory cells: When analyzing an optimistic contract signing protocols this appeared to be a limitation and only the status of a single contract is modeled. Arapinis et al. have to provide a manual proof to justify the correctness of this protocol-specific abstraction. We highlight the differences between our calculus and the calculus StatVerif offers in Section 5.2.3. A more extensive discussion of StatVerif, the abstraction-by-set-membership approach by Mödersheim [74], the state synchronisation extension to Guttman's strand space model [47] and work tailored to particular applications [36, 35] can be found in Chapter 3.

In the goal of relating different approaches for protocol analysis, Bistarelli et al. [11] also proposed a translation from a process algebra to multiset rewriting, which is very similar to our goal. They do however not consider private channels or global state and assume that processes have a particular structure. Because of those limitations, their translation and its correctness proof differ significantly from ours. Moreover their work does not include any tool support for automated verification, where we designed our translation with regard to tamarin's constraint solving algorithm.

5.2 A CRYPTOGRAPHIC PI CALCULUS WITH EXPLICIT STATE

Our specification language includes support for private channels, for an explicit global state and for locking mechanisms (which are crucial to write meaningful programs in which concurrent threads manipulate a common memory). The underlying tamarin prover is able to

$\langle M,N \rangle$ ::= $x, y, z \in \mathcal{V}$
| $p \in PN$
| $n \in FN$
| $f(M_1,\ldots,M_n)$ ($f \in \Sigma$ of arity n)

$\langle P,Q \rangle$ ::= 0
| $P \mid Q$
| $! P$
| $\nu n; P$
| $\mathsf{out}(M, N); P$
| $\mathsf{in}(M, N); P$
| if M=N then P [else Q]
| event F ; P ($F \in \mathcal{F}$)
| insert M,N; P
| delete M; P
| lookup M as x in P [else Q]
| lock M; P
| unlock M; P
| $[L] -[A]\rightarrow [R]; P$ ($L, R, A \in \mathcal{F}^*$)

Figure 9: Syntax.

analyze protocols without bounding the number of sessions, nor making any abstractions. Moreover it allows for modelling a wide range of cryptographic primitives by the means of equational theories. As the underlying verification problem is undecidable, tamarin may not terminate. However, it offers an interactive mode with a graphical user interface which helps the user to manually guide the tool in its proof. The translation has been carefully engineered in order to favor termination by tamarin.

5.2.1 *Syntax and informal semantics*

Our calculus is a variant of the applied pi calculus [2]. In addition to the usual operators for concurrency, replication, communication and name creation, it offers several constructs for reading and updating an explicit global state. The grammar for processes is described in Figure 9.

0 denotes the terminal process. For readability we sometimes omit trailing 0 processes. $P \mid Q$ is the parallel execution of processes P and Q and $!P$ the replication of P, allowing an unbounded number of sessions in protocol executions. The construct $\nu n; P$ binds the name n in P and models the generation of a fresh, random value. Processes $\mathsf{out}(M, N); P$ and $\mathsf{in}(M, N); P$ represent the output, respectively input, of message N on channel M. Readers familiar with the applied

pi calculus [2] may note that we opted for the possibility of pattern matching in the input construct, rather than merely binding the input to a variable x. The process if M = N then P else Q will execute P if $M =_E N$ and Q otherwise. As usual we sometimes omit to write else Q when Q is 0. The event construct is merely used for annotating processes and will be useful for stating security properties.

The remaining constructs are used for manipulating state and are new compared to the applied pi calculus. We offer two different mechanisms for state. The first construct is *functional* and associates a value to a key. The construct insert M,N binds the value N to a key M. Successive inserts change this binding. The delete M operation simply "undefines" the mapping for the key M. The lookup M as x in P else Q construct retrieves the value associated to M, and binds it to the variable x in P. If the mapping is undefined for M, the process behaves as Q. The lock and unlock constructs can be used to gain exclusive access to a resource M. This is essential for writing protocols where parallel processes may read and update a common memory. We additionally offer another kind of global state in form of a second *store*, modelled as a multiset of ground facts.

This second store allows access to the underlying notion of state in tamarin, but it is distinct from the previously introduced functional state. It is similar in style to the state extension in the strand space model [47] and the underlying specification language of the tamarin tool [89, 90]: It is accessed via the construct $[L] -[A]\rightarrow [R]; P$, which matches each fact in the sequence L to facts in the current store and, if successful, adds the corresponding instance of facts R to the store. The facts A are used as annotations in a similar way to events.

The following example illustrates a problem that occurs in the analysis of security APIs such as PKCS#11 [36, 20, 44], and will serve as a running example throughout this chapter.

Example 14: The following API allows the creation of keys in some secure memory. The user can access the device via an API. If she creates a key, she obtains a handle, which she can use to let the device perform operations on her behalf. The goal is that the user can never gain knowledge of the key, as the user's machine might be compromised. Consider the following process modelling the device (we use out(m) as a shortcut for out('c', m) for a public channel 'c'):

$$P_{ex} := !P_{new} \mid !P_{set} \mid !P_{dec} \mid !P_{wrap},$$

where

$$\begin{aligned} P_{new} := {} & \nu h; \nu k; \text{event NewKey(h,k);} \\ & \text{insert <'key',h>,k;} \\ & \text{insert <'att',h>,'dec'; out(h)} \end{aligned} \tag{1}$$

In the first line, the device creates a new handle h and a key k, and logs the creation of this key. It then stores the key that belongs to

the handle by associating the pair $\langle$'key', h$\rangle$ to the value of the key k. In the next line, $\langle$'att', h$\rangle$ is associated to a public constant 'dec'. Intuitively, we use the public constants 'key' and 'att' to distinguish two databases. The process

$$P_{set} := \mathsf{in}(h);\ \mathsf{insert}\ \langle\text{'att'},h\rangle,\ \text{'wrap'}$$

provides an interface for changing the attribute of a key from the initial value 'dec' to another value 'wrap' to the user. If a handle has the 'dec' attribute set, it can be used for decryption:

$$\begin{aligned} P_{dec} := \mathsf{in}(\langle\ h,c\rangle);\ \mathsf{lookup}\ \langle\text{'att'},h\rangle\ \mathsf{as}\ a\ \mathsf{in} \\ \mathsf{if}\ a=\text{'dec'}\ \mathsf{then}\ \mathsf{lookup}\ \langle\text{'key'},h\rangle\ \mathsf{as}\ k\ \mathsf{in} \\ \mathsf{if}\ \mathsf{encSucc}(c,k)=\mathsf{true}\ \mathsf{then} \\ \mathsf{event}\ \mathsf{DecUsing}(k,\mathsf{sdec}(c,k));\ \mathsf{out}(\mathsf{sdec}(c,k)) \end{aligned} \tag{2}$$

The first lookup stores the value associated to $\langle$'att', h$\rangle$ in a. The value is compared against 'dec'. If the comparison and another lookup for the associated key value k succeeds, we check whether decryption succeeds and, if so, output the plaintext.

If a key has the 'wrap' attribute set, it might be used to encrypt the value of a second key:

$$\begin{aligned} P_{wrap} := \mathsf{in}(\langle h_1,h_2\rangle);\ \mathsf{lookup}\ \langle\text{'att'},h_1\rangle\ \mathsf{as}\ a_1\ \mathsf{in} \\ \mathsf{if}\ a_1=\text{'wrap'}\ \mathsf{then}\ \mathsf{lookup}\ \langle\text{'key'},h_1\rangle\ \mathsf{as}\ k_1\ \mathsf{in} \\ \mathsf{lookup}\ \langle\text{'key'},\ h_2\rangle\ \mathsf{as}\ k_2\ \mathsf{in} \\ \mathsf{event}\ \mathsf{Wrap}(k_1,k_2);\ \mathsf{out}(\mathsf{senc}(k_2,k_1)) \end{aligned}$$

The bound names of a process are those that are bound by νn. We suppose that all names of sort *fresh* appearing in the process are under the scope of such a binder. Free names must be of sort *pub*. A variable x can be bound in three ways:

1. by the construct lookup M as x, or
2. $x \in vars(N)$ in the construct in(M, N) and x is not under the scope of a previous binder,
3. $x \in vars(L)$ in the construct $[L] -[A]\rightarrow [R]$ and x is not under the scope of a previous binder.

While the construct lookup M as x always acts as a binder, the input construct and the $[L] -[A]\rightarrow [R]$ constructs do not rebind an already bound variable but perform pattern matching. For instance in the process

$$P = \mathsf{in}(c,f(x));\ \mathsf{in}(c,g(x))$$

x is bound by the first input and pattern matched in the second. We discuss this choice in Section 5.2.3.3.

$$\frac{a \in FN \cup PN \quad a \notin \tilde{n}}{\nu\tilde{n}.\sigma \vdash a}\ \textsc{Dname} \qquad \frac{\nu\tilde{n}.\sigma \vdash t \quad t =_E t'}{\nu\tilde{n}.\sigma \vdash t'}\ \textsc{DEq}$$

$$\frac{x \in \mathbf{D}(\sigma)}{\nu\tilde{n}.\sigma \vdash x\sigma}\ \textsc{DFrame} \qquad \frac{\nu\tilde{n}.\sigma \vdash t_1 \cdots \nu\tilde{n}.\sigma \vdash t_n \quad f \in \Sigma^k}{\nu\tilde{n}.\sigma \vdash f(t_1, \ldots, t_n)}\ \textsc{DAppl}$$

Figure 10: Deduction rules.

A process is ground if it does not contain any free variable. We denote by $P\sigma$ the application of the homomorphic extension of the substitution σ to P. As usual we suppose that the substitution only applies to free variables. We sometimes interpret the syntax tree of a process as a term and write $P|_p$ to refer to the subprocess of P at position p (where $|$, if and lookup are interpreted as binary symbols, all other constructs as unary).

5.2.2 *Semantics*

5.2.2.1 *Frames and deduction*

Before giving the formal semantics of our calculus we introduce the notions of frame and deduction. A *frame* consists of a set of fresh names $\tilde{n}$ and a substitution σ and is written $\nu\tilde{n}.\sigma$. Intuitively a frame represents the sequence of messages that have been observed by an adversary during a protocol execution, σ and the secrets generated by the protocol, $\tilde{n}$, a priori unknown to the adversary. Deduction models the capacity of the adversary to compute new messages from the observed ones.

Definition 9 (Deduction)*:* We define the deduction relation $\nu\tilde{n}.\sigma \vdash t$ as the smallest relation between frames and terms defined by the deduction rules in Figure 10.

Example 15*:* If one key is used to wrap a second key, then, given the intruder learns the first key, he can deduce the second. For $\tilde{n} = k_1, k_2$ and $\sigma = \{^{senc(k_2,k_1)}/_{x_1}, ^{k_1}/_{x_2}\}$, $\nu\tilde{n}.\sigma \vdash k_2$, since:

$$\dfrac{\dfrac{\dfrac{x_1 \in \mathbf{D}(\sigma)}{\nu\tilde{n}.\sigma \vdash senc(k_2,k_1)} \quad \dfrac{x_2 \in \mathbf{D}(\sigma)}{\nu\tilde{n}.\sigma \vdash k_1}}{\nu\tilde{n}.\sigma \vdash sdec(senc(k_2,k_1),k_1)} \quad sdec(senc(k_2,k_1),k_1) =_E k_2}{\nu\tilde{n}.\sigma \vdash k_2}$$

5.2.2.2 *Operational semantics*

We can now define the operational semantics of our calculus. The semantics is defined by a labelled transition relation between process configurations. A *process configuration* is a 6-tuple $(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L})$ where

- $\mathcal{E} \subseteq FN$ is the set of fresh names generated by the processes,
- $\mathcal{S}\colon \mathcal{M}_\Sigma \to \mathcal{M}_\Sigma$ is a partial function modeling the functional store.
- $\mathcal{S}^{MS} \subseteq \mathcal{G}^\#$ is a multiset of ground facts and models the multiset of stored facts
- $\mathcal{P}$ is a multiset of ground processes representing the processes executed in parallel
- σ is a ground substitution modeling the messages output to the environment
- $\mathcal{L} \subseteq \mathcal{M}_\Sigma$ is the set of currently acquired locks.

The transition relation is defined by the rules described in Figure 11. Transitions are labelled by sets of ground facts. For readability we omit empty sets and brackets around singletons, i.e., we write $\to$ for $\xrightarrow{\emptyset}$ and $\xrightarrow{f}$ for $\xrightarrow{\{f\}}$. We write $\to^*$ for the reflexive, transitive closure of $\to$ (the transitions that are labelled by the empty sets) and write $\stackrel{f}{\Rightarrow}$ for $\to^* \xrightarrow{f} \to^*$. We can now define the set of traces, i.e., possible executions, that a process admits.

Definition 10 (Traces of P)*:* Given a ground process P we define the *set of traces of* P as

$$traces^{pi}(P) = \Big\{ [F_1, \ldots, F_n] \mid (\emptyset,\emptyset,\emptyset,\{P\},\emptyset,\emptyset) \stackrel{F_1}{\Longrightarrow} (\mathcal{E}_1, \mathcal{S}_1, \mathcal{S}_1^{MS}, \mathcal{P}_1, \sigma_1, \mathcal{L}_1) \stackrel{F_2}{\Longrightarrow} \ldots \stackrel{F_n}{\Longrightarrow} (\mathcal{E}_n, \mathcal{S}_n, \mathcal{S}_n^{MS}, \mathcal{P}_n, \sigma_n, \mathcal{L}_n) \Big\}$$

Example 16*:* The following examples shows how the first key is created on the security device in our running example.

$$\begin{aligned}
&(\emptyset,\emptyset,\emptyset,\{\,!P_{new} \mid !P_{set} \mid !P_{dec} \mid !P_{wrap}\,\}^\#,\emptyset,\emptyset)\\
\to&(\emptyset,\emptyset,\emptyset,\{\underbrace{\,!P_{new}, !P_{set} \mid !P_{dec} \mid !P_{wrap}\,}_{=:\mathcal{P}'}\}^\#,\emptyset,\emptyset)\\
\to&(\emptyset,\emptyset,\emptyset,\{\,P_{new}\,\}^\# \cup^\# \mathcal{P}',\emptyset,\emptyset)\\
\to&(\emptyset,\emptyset,\emptyset,\{\ \texttt{new h; new k; event NewKey(h,k); }\ldots\}^\# \cup^\# \mathcal{P}',\emptyset,\emptyset)\\
\to^*&(\{h', k'\},\emptyset,\emptyset,\{\ \texttt{event NewKey(h',k'); }\ldots\}^\# \cup^\# \mathcal{P}',\emptyset,\emptyset)
\end{aligned}$$

$$\xrightarrow{\text{NewKey}(h',k')}$$

$$\begin{aligned}
&(\{h', k'\},\emptyset,\emptyset,\{\ \texttt{insert } \langle \text{'key'}, h\rangle, k; \ldots\}^\# \cup^\# \mathcal{P}',\emptyset,\emptyset)\\
\to^*&(\{h', k'\},\mathcal{S},\emptyset,\{\,\texttt{out(h); 0}\,\}^\# \cup^\# \mathcal{P}',\emptyset,\emptyset)\\
\to^*&(\{h', k'\},\mathcal{S},\emptyset,\mathcal{P}',\{\,{}^{h'}/_{x_1}\,\},\emptyset), \text{ where}
\end{aligned}$$

$\mathcal{S}(\langle \text{'key'}, h'\rangle) = k'$ and $\mathcal{S}(\langle \text{'att'}, h'\rangle) = \text{'dec'}$. Therefore,

$$[\text{NewKey}(h', k')] \in traces^{pi}(P).$$

Standard operations :

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{0\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L})$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P|Q\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P, Q\}, \sigma, \mathcal{L})$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{!P\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{!P, P\}, \sigma, \mathcal{L})$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\nu a; P\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E} \cup \{a'\}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\{a'/a\}\}, \sigma, \mathcal{L})$$
if a' is fresh

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L}) \xrightarrow{K(M)} (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P}, \sigma, \mathcal{L}) \quad \text{if } \nu\mathcal{E}.\sigma \vdash M$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{out}(M, N); P\}, \sigma, \mathcal{L}) \xrightarrow{K(M)} (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma \cup \{{}^{N}/_{x}\}, \mathcal{L})$$
if x is fresh and $\nu\mathcal{E}.\sigma \vdash M$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{in}(M, N); P\}, \sigma, \mathcal{L}) \xrightarrow{K(\langle M,N \rangle)} (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\tau\}, \sigma, \mathcal{L})$$
if $\exists\tau.\ \tau$ is grounding for N, $\nu\mathcal{E}.\sigma \vdash M$, $\nu\mathcal{E}.\sigma \vdash N\tau$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{out}\ (M,N);P,\ \mathsf{in}\ (M',N');Q\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{P, Q\tau\}, \sigma, \mathcal{L})$$
if $M =_E M'$ and $\exists\tau.\ N =_E N'\tau$ and τ grounding for N'

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{\text{if } M = N \text{ then } P \text{ else } Q\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{P\}, \sigma, \mathcal{L}) \quad \text{if } M =_E N$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{\text{if } M = N \text{ then } P \text{ else } Q\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{Q\}, \sigma, \mathcal{L}) \quad \text{if } M \neq_E N$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{\mathsf{event}(F);P\}, \sigma, \mathcal{L}) \xrightarrow{F} (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{P\}, \sigma, \mathcal{L})$$

Operations on global state:

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{insert}\ M, N;\ P\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}[M \mapsto N], \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L})$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{delete}\ M;P\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}', \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L})$$
where $$\mathcal{S}'(x) = \begin{cases} \mathcal{S}(x) & \text{if } x \neq_E M \\ \bot & \text{otherwise} \end{cases}$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{lookup}\ M \text{ as } x \text{ in } P \text{ else } Q\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\{V/x\}\}, \sigma, \mathcal{L})$$
if $\mathcal{S}(N) =_E V$ is defined and $N =_E M$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{lookup}\ M \text{ as } x \text{ in } P \text{ else } Q\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{Q\}, \sigma, \mathcal{L})$$
if $\mathcal{S}(N)$ is undefined for all $N =_E M$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{lock}\ M;\ P\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \cup \{M\})$$
if $M \not\in_E \mathcal{L}$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{\mathsf{unlock}\ M;\ P\}, \sigma, \mathcal{L}) \longrightarrow (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \setminus \{M' \mid M' =_E M\})$$

$$(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup^{\#} \{[l \dashv[a]\rightarrow r];\ P\}, \sigma, \mathcal{L}) \xrightarrow{a'} (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS} \setminus \mathit{lfacts}(l') \cup^{\#} r', \mathcal{P} \cup^{\#} \{P\tau\}^{\#}, \sigma, \mathcal{L})$$
if $\exists\tau, l', a', r'.$
with τ grounding for $l \dashv[a]\rightarrow r$
and $l' \dashv[a']\rightarrow r' \in_E (l \dashv[a]\rightarrow r)\tau$
and $\mathit{lfacts}(l') \subseteq^{\#} \mathcal{S}^{MS}, \mathit{pfacts}(l') \subset \mathcal{S}^{MS}$

Figure 11: Operational semantics.

5.2.3 *Discussion*

In the following, we will discuss various design decision we have made and compare our calculus with the StatVerif calculus.

5.2.3.1 *Private Channels and Store*

The careful reader might have noticed that the applied pi calculus actually offers a way to store a value, such that it can be accessed from within a different process. If a private channel s is only known to the part of the process allowed to access the store, a fact could be stored by sending it on the channel s, and, similarly, be read by reading from s. So, one might ask why we chose to introduce new constructs for reading and writing state, instead of just providing a sound and complete translation for the communication on private channels and reusing existing, well-accepted syntax. Besides the fact that a value read from the channel "disappears", hence every lookup to the store would need to put the value back on the channel, which would be cumbersome to write, there is another, more important reason: Communication on a private channel is synchronous in our calculus. This means that a process sending a message cannot proceed until the message has been received by a second process. A quick look on the definition of P_{new} (see (1) on page 64) should convince the reader that this is impractical in many cases. In general, databases constitute asynchronous ways of communication and should be modelled as such.

On the other hand, we decided for synchronous message passing in favour of asynchronous message passing, first, because synchronous message passing is more powerful (Palamidessi showed that the synchronous applied pi calculus is strictly more expressive than the asynchronous applied pi calculus [77]) and second, because by using the same mechanism as in the popular protocol verifier ProVerif, we keep our calculus more accessible for users familiar with this tool.

5.2.3.2 *Distributed Databases*

There is only one store for the whole protocol. It is, of course, unrealistic to assume all protocol parties have access to the same database. To model an unbounded number of machines, each with a distinct store, each database access might be prefixed with an identifier to the machine, like in the following template:

$$!(\nu\ \mathsf{id};\ \ldots;\ \mathsf{insert}\ \langle \mathsf{id},\mathsf{x}\rangle,\ \mathsf{y};\ \ldots)$$

If the adversary shall have (partial) access to the store, this needs to be made explicit, by having a process providing an interface to the store. See the following example, where P′ is the actual protocol:

```
!(in(y). insert 'AdversaryCell', y)
| !(lookup 'AdversaryCell' as y in out(y)) | P′
```

5.2.3.3 *Lookup is binding*

It might seem odd that lookup acts as a binder, while input does not. We justify this decision as follows: As P_{dec} and P_{wrap} in the previous example show, lookups appear often after input was received. If lookups were to use pattern matching, the following process

```
P = in(c,x); lookup 'store' as x in P′
```

might unexpectedly perform a check if 'store' contains the message given by the adversary, instead of binding the content of 'store' to x, due to an undetected clash in the naming of variables.

5.2.3.4 *Inline multiset rewriting*

In addition to the access to the functional store via lookup, insert and delete, we support a "low-level" form of state manipulation in form of the construct [L] −[A]→ [R]; P. This style of state manipulation is similar to the state extension in the strand space model [47] and the underlying specification language of the tamarin tool [89, 90].

This low-level store is distinct from the functional store, which is a restriction imposed by our translation. We decided to nevertheless include this construct as a way of accessing the functionality of the underlying tool, tamarin. This offers a great flexibility and has shown useful in our case studies. However, we recommend its use only to users familiar with how our translation works. The discussion on the translation (see Section 5.3.3) gives further insight on why the functional store is distinct from this low-level store.

Note further that data can be moved from the functional store to the low-level store, and vice versa, for example as follows: lookup 'store1' as x in [] −[]→ [store2(x)].

5.2.3.5 *Comparison with StatVerif*

Our calculus is very close to the StatVerif calculus by Arapinis et al. [5]. Notable differences are the following:

1. *Destructor application:* Since tamarin handles the equational theory without making a distinction between constructors and destructors, our calculus does not have a construct for this. The definition of P_{dec} (see (2) on page 65) gives an example on how to handle destructors that might fail, in this case decryption.

2. *Initial state:* In contrast to StatVerif, we have no special operator for state initialisation. Values are undefined from the start, and can be initialised during the protocol run. The modeller can use axioms (c. f. Section 5.3.2, in particular the definition of α_{init}) to enforce the initialisation of the store before the protocol is executed.

3. *State cells:* StatVerif's state cells roughly correspond to keys in the sense of our functional store. While state cells are names and there is only an arbitrary, but a-priori fixed amount of them, keys to the store can be arbitrary terms and there is no a-priori bound on them.

4. *Locking:* Both StatVerif and our calculus use locks(binary semaphores) for process synchronisation. The locking mechanism described in the paper introducing StatVerif [5] can only lock one global resource, but the current version of their translation tool handles the locking of individual cells, as well. The mechanism in our tool can lock arbitrary terms, so similar to the previous point, we have no a-priori bound on the number of lockable resources. There is a syntactic restriction in StatVerif that "the part of the process P that occurs before the next unlock, if any, may not include parallel, replication, or lock." which corresponds to the condition on *well-formed* processes for our calculus(c. f. Definition 13). Well-formedness is a requirement for the translation to multiset rewrite rules, which means that the same restriction applies to our calculus.

   The $L\ -[A]\rightarrow\ R$ construct gives the user a way to implement locking in a different, less restricted manner, see Section 5.3.3.4 for further discussion.

5.3 A TRANSLATION FROM PROCESSES INTO MULTISET REWRITE RULES

In this section we define a translation from a process P into a set of multiset rewrite rules $[\![P]\!]$ and a translation on trace formulas such that $P \models^{\forall} \varphi$ if and only if $[\![P]\!] \models^{\forall} [\![\varphi]\!]$. Note that the result also holds for satisfiability, as an immediate consequence. For a rather expressive subset of trace formulas (see [89] for the exact definition of the fragment), checking whether $[\![P]\!] \models^{\forall} [\![\varphi]\!]$ can then be discharged to the tamarin prover that we use as a backend.

5.3.1 *Translation of processes*

We recall the adversary's message deduction capabilities from Section 3.6.2, encoded in the following set of rules MD:

$$\mathsf{Out}(x) \multimap[\quad]\to !\mathsf{K}(x) \qquad \text{(MDOut)}$$

$$!\mathsf{K}(x) \multimap[\mathsf{K}(x)]\to \mathsf{In}(x) \qquad \text{(MDIn)}$$

$$\multimap[\quad]\to !\mathsf{K}(x : \mathit{pub}) \qquad \text{(MDPub)}$$

$$\mathsf{Fr}(x : \mathit{fresh}) \multimap[\quad]\to !\mathsf{K}(x : \mathit{fresh}) \qquad \text{(MDFresh)}$$

$$!\mathsf{K}(x_1), \ldots, !\mathsf{K}(x_k) \multimap[\quad]\to !\mathsf{K}(f(x_1, \ldots, x_k)) \quad \text{for } f \in \Sigma^k \qquad \text{(MDAppl)}$$

In order for our translation to be correct, we need to make some assumptions on the set of processes we allow. These assumptions are however, as we will see, rather mild and most of them without loss of generality. First we define a set of variables that will be used in our translation and is therefore reserved.

Definition 11 (Reserved variables and facts)*:* The set of reserved variables is defined as the set containing the elements n_a for any $a \in FN$, $lock_l$ for any l.

The set of reserved facts $\mathcal{F}_{res}$ is defined as the set containing facts $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n \in \mathcal{T}$ and $f \in \{$ Init, Insert, Delete, IsIn, IsNotSet, state, Lock, Unlock, Out, Fr, In, Msg, ProtoNonce, Eq, NotEq, Event, InEvent $\}$.

Similar to [5], we require for our translation that any unlock t in a process can be assigned to a lock t preceding it in the process' syntax tree, and that this assignment is injective. Moreover, given a process lock t; P the corresponding unlock in P may not be under a parallel or replication. These conditions allow us to annotate each corresponding lock t, unlock t pair with a unique label l. The annotated version of a process P is denoted $\overline{P}$. In case the annotation fails, i.e., P violates one of the above conditions, the process $\overline{P}$ contains $\bot$. The formal definition of $\overline{P}$ is as follows:

Definition 12 (Process annotation)*:* Given a ground process P we define the annotated ground process $\overline{P}$ as follows:

$$
\begin{aligned}
\overline{0} &:= 0 \\
\overline{P|Q} &:= \overline{P}|\overline{Q} \\
\overline{!P} &:= !\overline{P} \\
\overline{\text{if } t_1 = t_2 \text{ then } P \text{ else } Q} &:= \text{if } t_1 = t_2 \text{ then } \overline{P} \text{ else } \overline{Q} \\
\overline{\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q} &:= \text{lookup } M \text{ as } x \text{ in } \overline{P} \text{ else } \overline{Q} \\
\overline{\alpha; P} &:= \alpha; \overline{P} \\
&\quad \text{where } \alpha \notin \{\text{lock } t, \text{unlock } t \colon t \in \mathcal{T}\} \\
\overline{\text{lock } t; P} &:= \text{lock}^l\ t; \overline{au(P, t, l)} \\
&\quad \text{where } l \text{ is a fresh label} \\
\overline{\text{unlock}^l\ t; P} &:= \text{unlock}^l\ t; \overline{P} \\
\overline{\text{unlock } t; P} &:= \bot
\end{aligned}
$$

where $au(P, t, l)$ annotates the first unlock that has parameter t with the label l, i. e.:

$$
\begin{aligned}
au(P|Q, t, l) &:= \bot \\
au(!P, t, l) &:= \bot \\
au(\text{if } t_1 = t_2 \text{ then } P \text{ else } Q, t, l) &:= \text{if } t_1 = t_2 \text{ then } au(p, t, l) \text{ else } au(Q, t, l) \\
au(\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q, t, l) &:= \text{lookup } M \text{ as } x \text{ in } au(p, t, l) \text{ else } au(Q, t, l) \\
au(\alpha; Pt, l) &:= \alpha; au(P, t, l) \quad \text{where } \alpha \neq \text{unlock } t \\
au(\text{unlock } t; Pt, l) &:= \text{unlock}^l\ t; P \\
au(0, t, l) &:= 0
\end{aligned}
$$

Definition 13 (well-formed)*:* A ground process P is well-formed if

- no reserved variable nor a reserved fact appear in P,
- any name and variable in P is bound at most once and
- $\overline{P}$ does not contain $\bot$.
- For each action $(l -[a]\rightarrow r)$ that appears in the process, the following holds: For each $(l' -[a']\rightarrow r') \in_E \mathit{ginsts}(l -[a]\rightarrow r)$ we have that $\cap_{r''=_E r'} \mathit{names}(r'') \cap \mathit{FN} \subseteq \cap_{l''=_E l'} \mathit{names}(l'') \cap \mathit{FN}$.

A trace formula φ is well-formed if no reserved variable nor a reserved fact appear in φ.

The two first restrictions of well-formed processes are not a loss of generality as processes and formulas can be consistently renamed to avoid reserved variables and α-converted to avoid binding names or variables several times. Also note that the second condition is not necessarily preserved during an execution, e.g. when unfolding a replication $!P$ and P may bind the same names. We only require this condition to hold on the initial process for our translation to be correct.

The annotation of locks restricts the set of protocols we can translate, but allows us to obtain better verification results, since we can predict which unlock is "supposed" to close a given lock. This additional information is helpful for tamarin's backward reasoning. During our case studies, this restriction never formed an obstacle. We nevertheless discuss how the modeller can overcome this restriction in Section 5.3.3.4.

The fourth condition is due to condition three in Definition 2 and ensures that the embedded multiset rewrite rules cannot create fresh names.

Definition 14*:* Given a well-formed ground process P we define the multiset rewrite system $[\![P]\!]$ as

$$MD \cup \{\textsc{Init}\} \cup [\![\overline{P}, [], []]\!]$$

- where the rule Init is defined as

$$\textsc{Init} : [] \dashv[\mathsf{Init}()]\rightarrow [\mathsf{state}_{[]}()]$$

- $[\![P, p, \tilde{x}]\!]$ is defined inductively for process P, position $p \in \mathbb{N}^*$ and sequence of variables $\tilde{x}$ in Figure 12.

- For a position p of P we define state_p to be persistent if $P|_p = !Q$ for some process Q; otherwise state_p is linear.

In the definition of $[\![P, p, \tilde{x}]\!]$ we intuitively use the family of facts state_p to indicate that the process is currently at position p in its syntax tree. A fact state_p will indeed be true in an execution of these rules whenever some instance of P_p (i.e. the process defined by the subtree at position p of the syntax tree of P) is in the multiset $\mathcal{P}$ of the process configuration. Note in particular that the translation of $!P$ results in a persistent fact as $!P$ always remains in $\mathcal{P}$. The translation of the zero-process, parallelisation and replication do nothing but handle state_p-facts.

The translation of the construct $\nu\, a$ translates the name a into a variable n_a, as multiset rewrite rules cannot contain fresh names. Any instantiation of this rule will substitute n_a by a fresh name, which the Fr-fact in the premise guarantees to be new. This step is annotated with a (reserved) action *ProtoNonce*, which is used in the proof of correctness to distinguish adversary and protocol nonces. Note that

$$
\begin{aligned}
[\![0, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x})] \to []\} \\
[\![P \mid Q, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x})] \to [\mathsf{state}_{p\cdot 1}(\tilde{x}), \mathsf{state}_{p\cdot 2}(\tilde{x})]\} \\
&\qquad \cup [\![P, p\cdot 1, \tilde{x}]\!] \cup [\![Q, p\cdot 2, \tilde{x}]\!] \\
[\![!P, p, \tilde{x}]\!] &= \{[!\mathsf{state}_p(\tilde{x})] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup [\![P, p\cdot 1, \tilde{x}]\!] \\
[\![\nu a; P, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x}), \mathsf{Fr}(n_a : \mathit{fresh})] \multimap [\mathit{ProtoNonce}(n_a : \mathit{fresh})] \to \\
&\qquad [\mathsf{state}_{p\cdot 1}(\tilde{x}, n_a : \mathit{fresh})]\} \cup [\![P, p\cdot 1, (\tilde{x}, n_a : \mathit{fresh})]\!] \\
[\![\mathsf{out}(M, N); P, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x}), \mathsf{In}(M)] \multimap [\mathrm{InEvent}(M)] \to [\mathsf{Out}(N), \mathsf{state}_{p\cdot 1}(\tilde{x})], \\
&\qquad [\mathsf{state}_p(\tilde{x})] \to [\mathsf{Msg}(M, N), \mathsf{state}_p^{\mathsf{semi}}(\tilde{x})], \\
&\qquad [\mathsf{state}_p^{\mathsf{semi}}(\tilde{x}), \mathsf{Ack}(M, N)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \\
&\qquad \cup [\![P, p\cdot 1, \tilde{x}]\!] \\
[\![\mathsf{in}(M, N); P, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x}), \mathsf{In}(\langle M, N\rangle)] \multimap [\mathrm{InEvent}(\langle M, N\rangle)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(N))], \\
&\qquad [\mathsf{state}_p(\tilde{x}), \mathsf{Msg}(M, N)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(N)), \mathsf{Ack}(M, N)]\} \\
&\qquad \cup [\![P, p\cdot 1, \tilde{x} \cup \mathit{vars}(N)]\!] \\
[\![\text{if } M = N \text{ then } P &= \{[\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Eq}(M, N)] \to \quad [\mathsf{state}_{p\cdot 1}(\tilde{x})], \\
\text{else } Q, p, \tilde{x}]\!] &\quad [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{NotEq}(M, N)] \to \quad [\mathsf{state}_{p\cdot 2}(\tilde{x})]\} \\
&\qquad \cup [\![P, p\cdot 1, \tilde{x}]\!] \cup [\![Q, p\cdot 2, \tilde{x}]\!] \\
[\![\text{event } F; P, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Event}(), F] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup [\![P, p\cdot 1, \tilde{x}]\!] \\
[\![\text{insert } s, t; P, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Insert}(s, t)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup [\![P, p\cdot 1, \tilde{x}]\!] \\
[\![\text{delete } s; P, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Delete}(s)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup [\![P, p\cdot 1, \tilde{x}]\!] \\
[\![\text{lookup } M \text{ as } v &= \{[\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{IsIn}(M, v)] \to [\mathsf{state}_{p\cdot 1}(\tilde{M}, v)], \\
\text{in } P \text{ else } Q, p, \tilde{x}]\!] &\quad [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{IsNotSet}(M)] \to [\mathsf{state}_{p\cdot 2}(\tilde{x})]\} \\
&\qquad \cup [\![P, p\cdot 1, (\tilde{x}, v)]\!] \cup [\![Q, p\cdot 2, \tilde{x}]\!] \\
[\![\mathrm{lock}^l\ s; P, p, \tilde{x}]\!] &= \{[\mathsf{Fr}(\mathrm{lock}_l), \mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Lock}(\mathit{lock}_l, s)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x}, \mathit{lock}_l)]\} \\
&\qquad \cup [\![P, p\cdot 1, \tilde{x}]\!] \\
[\![\mathrm{unlock}^l\ s; P, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Unlock}(\mathit{lock}_l, s)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup [\![P, p\cdot 1, \tilde{x}]\!] \\
[\![[l \multimap [a] \to r]; P, p, \tilde{x}]\!] &= \{[\mathsf{state}_p(\tilde{x}), l] \multimap [\mathrm{Event}(), a] \to [r, \mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(l))]\} \\
&\qquad \cup [\![P, p\cdot 1, \tilde{x} \cup \mathit{vars}(l)]\!]
\end{aligned}
$$

Figure 12: Definition of $[\![P, p, \tilde{x}]\!]$.

the fact $\mathsf{state}_{p.1}$ in the conclusion carries n_a, so that the following protocol steps are bound to the fresh name used to instantiate n_a.

The first rules of the translation of in and out model the communication between the protocol and the adversary, and vice versa. In the first case, the adversary has to prove knowledge of the channel and the message she would like to send, in the second case knowledge of the channel suffices. The action InEvent serves a purpose that we will explain in the next section. The other rules model an internal communication on a synchronous channel, and are a little more complicated. If the adversary, who controls the scheduling, decides not to eavesdrop the communication and allow internal communication to happen, he should not be able to interrupt the communication step and change his mind – since the channel is synchronous, the sending process can only execute the following step if the message was successfully transmitted. For this reason, when the second rule of the translation of out is fired, the state-fact is substituted by a semi-state fact, $\mathsf{state}^{\mathsf{semi}}$. Additionally, the fact $\mathsf{Msg}(M, N)$ signals that a message is present on the synchronous channel. This fact is picked up by the second rule of the translation of in. Only with the resulting acknowledgement fact $\mathsf{Ack}(M, N)$ it is possible to advance the execution of the sending process, using the third role in the translation of out, which transforms the semi-state *and* the acknowledgement of receipt into $\mathsf{state}_{p.1}(\ldots)$. Only now the next step in the execution of the sending process can be executed.

Some of the labels are used to restrict the set of executions that we will consider. For instance the label $\mathsf{Eq}(M, N)$ will be used to indicate that we only consider executions in which $M =_E N$. This is also the case for event, insert, delete, lookup, lock and unlock. As we will see, these restrictions will be encoded in the trace formula. Finally, the construct for embedded multiset rewrite rules is translated by adding the previous state-fact to the left-hand side, and the next one to the right-hand side.

Example 17: $[\![!P_{new}]\!]$ gives the following set of rules:

$$
\begin{aligned}
[] &\dashv[\mathsf{Init}()]\rightarrow [\mathsf{state}_{[]}()]\\
[\mathsf{state}_{[]}()] &\dashv[\quad]\rightarrow [!\mathsf{state}_1()]\\
[!\mathsf{state}_1(), \mathsf{Fr}(h)] &\dashv[\quad]\rightarrow [\mathsf{state}_{11}(h)]\\
[\mathsf{state}_{11}(h), \mathsf{Fr}(k)] &\dashv[\quad]\rightarrow [\mathsf{state}_{111}(k, h)]\\
[\mathsf{state}_{111}(k, h)] &\dashv[\mathsf{Event}(), \mathsf{NewKey}(h, k)]\rightarrow [\mathsf{state}_{1111}(k, h)]\\
[\mathsf{state}_{1111}(k, h)] &\dashv[\mathsf{Insert}(\langle 'key', h\rangle, k)]\rightarrow [\mathsf{state}_{11111}(k, h)]\\
[\mathsf{state}_{11111}(k, h)] &\dashv[\mathsf{Insert}(\langle 'att', h\rangle, 'dec')]\rightarrow\\
&\qquad\qquad [\mathsf{state}_{111111}(k, h)]\\
[\mathsf{state}_{111111}(k, h)] &\dashv[\quad]\rightarrow [\mathsf{Out}(h), \mathsf{state}_{1111111}(k, h)]
\end{aligned}
$$

An example trace is presented in Figure 13. Every box in this picture stands for the application of a multiset rewrite rule, where the

premises are at the top, the conclusions at the bottom, and the action in middle, if there are any. Every premise needs to have a matching conclusion, which is visualized by the arrows, otherwise the graph would not depict a valid MSR execution. (This is a simplification of the dependency graph representation tamarin uses to perform backward-induction [89, 90].)

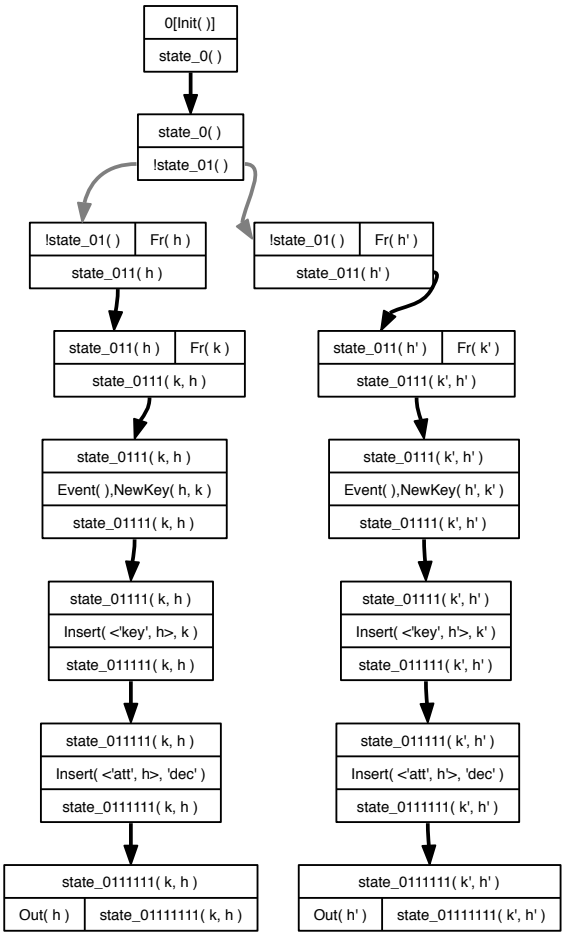


Figure 13: Example trace for translation of $!P_{new}$.

5.3.2 *Translation of trace formulas*

We can now define the translation for formulas.

Definition 15: Given a well-formed trace formula φ we define

$$[\![\varphi]\!]_\forall := \alpha \Rightarrow \varphi \qquad \text{and} \qquad [\![\varphi]\!]_\exists := \alpha \wedge \varphi$$

where $\alpha := \alpha_{init} \wedge \alpha_{eq} \wedge \alpha_{noteq} \wedge \alpha_{in} \wedge \alpha_{notin} \wedge \alpha_{lock} \wedge \alpha_{inEv}$ and

$$\begin{aligned}
\alpha_{init} &:= \forall x, y, i, j.\, \mathrm{Init}()@i \wedge \mathrm{Init}()@j \implies i = j \\
\alpha_{eq} &:= \forall x, y, i.\, \mathrm{Eq}(x, y)@i \implies x \approx y \\
\alpha_{noteq} &:= \forall x, y, i.\, \mathrm{NotEq}(x, y)@i \implies \neg(x \approx y)
\end{aligned}$$

$$
\begin{aligned}
\alpha_{in} :=& \forall x, y, t_3.\mathrm{IsIn}(x,y)@t_3 \implies \\
& \exists t_2.\mathrm{Insert}(x,y)@t_2 \wedge t_2 \lessdot t_3 \\
& \wedge (\forall t_1.\mathrm{Delete}(x)@t_1 \Rightarrow t_1 \lessdot t_2 \vee t_3 \lessdot t_1) \\
& \wedge (\forall t_1, y.\mathrm{Insert}(x,y)@t_1 \implies t_1 \lessdot t_2 \vee t_3 \lessdot t_1)) \\
\alpha_{notin} :=& \forall x, y, t_3.\mathrm{IsNotSet}(x)@t_3 \implies \\
& (\forall t_1, y.\mathrm{Insert}(x,y)@t_1 \implies t_3 \lessdot t_1) \\
& \vee\ \exists t_1.\mathrm{Delete}(x)@t_1 \wedge t_1 \lessdot t_3 \\
& \quad \wedge \forall t_2, y.\mathrm{Insert}(x,y)@t_2 \wedge t_2 \lessdot t_3 \implies t_2 \lessdot t_1 \\
\alpha_{lock} :=& \forall x, l, l', i, j.\mathrm{Lock}(l,x)@i \wedge \mathrm{Lock}(l',x)@j \wedge i \lessdot j \\
& \Rightarrow \exists k.\mathrm{Unlock}(l,x)@k \wedge i \lessdot k \wedge k \lessdot j \wedge \\
& (\forall l', m.\mathrm{Lock}(l',x)@m \Rightarrow \neg(i \lessdot m \wedge m \lessdot k)) \\
& \wedge (\forall l', m.\mathrm{Unlock}(l',x)@m \Rightarrow \neg(i \lessdot m \wedge m \lessdot k)) \\
\alpha_{inEv} :=& \forall t, i.\mathrm{InEvent}(t)@i \Rightarrow \exists j.K(t)@j \wedge \\
& (\forall k.\mathrm{Event}()@k \Rightarrow (k \lessdot j \vee i \lessdot k)) \wedge \\
& (\forall k, t'.K(t')@k \Rightarrow (k \lessdot j \vee i \lessdot k \vee k \approx j))
\end{aligned}
$$

The hypotheses of $[\![\varphi]\!]$ use the labels of the generated rules to filter out executions that we wish to discard:

- α_{init} ensures that the init rule is only fired once.
- α_{eq} and α_{noteq} ensure that we only consider traces where all (dis)equalities hold.
- α_{in} and α_{notin} ensure that all successful lookups were preceded by an insert that was neither revoked nor overwritten while all unsuccessful lookups where either never inserted, or at one point deleted and never re-inserted.
- α_{lock} checks that between each two matching locks there must be an unlock. Furthermore, between the first of these locks and the corresponding unlock, there is neither a lock nor an unlock.
- α_{inEv} ensures that, whenever an instance of MDIN is required to generate an In-fact, it is generated as late as possible, i. e., there is no visible Event between the action $!K(t)$ produced by MDIN, and a rule that requires $\mathsf{In}(t)$. This is needed to be correct with respect to the operational semantics of in and out, see Figure 11.

We also note that $Tr \vDash^{\forall} [\![\varphi]\!]_{\forall}$ iff $Tr \nvDash^{\exists} [\![\neg\varphi]\!]_{\exists}$.

5.3.3 *Discussion*

In the following we will discuss the advantages of using our translation as opposed to a direct modelling in form of multiset rewrite rules. Then we will give some insight into why the state manipulation and

locking constructs are modelled the way they are. Finally, we discuss the axioms we have chosen and the well-formedness conditions we impose on the locks.

5.3.3.1 *The need for such a translation*

Obviously any protocol that we are able to analyze can be directly analyzed by the tamarin prover [89, 90] as we use it as a backend. Indeed, tamarin has already been used for analyzing a model of the Yubikey device in Chapter 4, the case studies presented with Mödersheim's abstraction, as well as those presented with StatVerif. It is furthermore able to reproduce the aforementioned results on PKCS#11 [36] and the TPM [35] – moreover, it does so without bounding the number of keys, security devices, reboots, etc.

An important disadvantage is that the representation of protocols as multiset rewrite rules is very low level and far away from actual protocol implementations, making it very difficult to model a protocol correctly. Our calculus is semantically richer, in particular, it gives information on which part of the protocol runs on which party and which kind of requests are treated in which thread, respectively. This kind of information is fundamental in the development of protocols, as well as in the implementation of protocols starting from a protocol description. There has been work on the automated translation from protocol descriptions in the spi calculus (another variant of the applied pi calculus) into Java programs [79, 80], an approach that appears to be adaptable to our calculus, but not to multiset rewriting. This supports the claim that our calculus contains enough information to serve as a template for an implementation. For protocols that are first designed, then analysed for security, and later implemented on the basis of the analysed model, this translation from a model to an implementation needs to be performed, whether it is done automatically or by hand. One way or another, it should be as straight-forward as possible.

For the other direction, where a protocol description exists in one form or another, possibly in form of an implementation, and is to be modelled with the aim of analysing its security, we face similar problems: Encoding private channels, nested replications and locking mechanisms directly as multiset rewrite rules is a tricky and error-prone task. During our experiments with tamarin, we noticed that in particular the last point is often swept under the rug: Protocol steps typically consist of a single input, followed by several database lookups, and finally an output. In MSR, they are usually modelled as a single rule, and therefore effectively atomic. Real implementations are different from that: Several entities might be involved, database lookups could be slow, etc. In this case, such models could obscure eventual issues in concurrent protocol steps, that would otherwise require, e. g., explicit locking. Ignoring the fact that protocol steps

are not atomic in the model increases the risk of implicitly excluding attacks in the model that are well possible in a real implementation, e. g., race conditions. In the following, we will try to illustrate that the question of where to put locks is *a*) difficult to answer and *b*) a matter of security. Recall the running example, Example 14 on page 7. Using our translation and the tamarin-prover, we are able to show key-secrecy for this example, i. e., $traces^{pi}(P_{ex}) \models^{\forall} \phi$ for

$$\phi := \neg(\exists h, k\colon msg, i, j\colon temp.\ \mathrm{NewKey}(h,k)@i \wedge \mathrm{K}(k)@j).$$

(See Listing 21 for the full model including the necessary typing lemma.) Let us change the example, so it implements a slightly different policy: The initial attribute to a key is "init" instead of "dec", and there are two processes similar to P_{set}, one which allow the attribute to be changed from "init" to "dec", and one which allows the attribute to be changed from "init" to "enc":

$$P'_{ex} := !P'_{new} \mid !P_{set-w} \mid !P_{set-d} \mid !P_{dec} \mid !P_{wrap},$$

where

$$\begin{aligned}
P'_{new} &:= \nu\ h;\ \nu\ k;\ \mathsf{event\ NewKey(h,k);}\\
&\qquad \mathsf{insert\ <'key',h>,k;\ insert\ <'att',h>,'new';\ out(h)}\\
P_{set-d} &:= \mathsf{in(h);\ lookup\ <'att',h>\ as\ a\ in}\\
&\qquad \mathsf{if\ a='init'\ then\ insert\ <'att',h>,\ 'dec'}\\
P_{set-w} &:= \mathsf{in(h);\ lookup\ <'att',h>\ as\ a\ in}\\
&\qquad \mathsf{if\ a='init'\ then\ insert\ <'att',h>,\ 'wrap'}
\end{aligned}$$

P_{dec} and P_{wrap} are defined as before. It may come as a surprise that P'_{ex} allows for an attack, i. e., $traces^{pi}(P'_{ex}) \not\models^{\forall} \phi$, even though Example 14, as well as this altered variant seem to implement similar policies. We will only sketch this attack, which is a variation of the attack described in Example 1, and can be found using our translation: First, the attacker executes P'_{new} to create a key with a handle h. Then, she executes P_{set-d} with handle h until just before the insert command. She will continue this execution in a moment, but first she executes P_{set-w} with h, so that $\langle 'att', h\rangle$ points to $'wrap'$ in the store. This allows her to obtain $senc(k,k)$ by calling P_{wrap} with $\langle h, h\rangle$. Now, by continuing the previous execution of P_{set-d} she can alter the store so that $\langle 'att', h\rangle$ points to $'dec'$. Hence, she is allowed to run P_{dec} on $\langle h, senc(k,k)\rangle$, which reveals k.

This attack can be mitigated, by enclosing everything behind in(h) in the processes P_{set-w} and P_{set-d} between lock h and unlock h. This shows that, for the analysis of security APIs, state synchronisation is an essential part of the modelling and it is far from trivial to see when locking is needed (e. g., in P'_{ex}), and when it is not (e. g., P_{ex}). A modelling that is too coarse to capture such attacks (for example due

the "implicit" global lock during a multiset rewriting step) should be viewed critically, as it could give a false sense of security.

The modelling of the Yubikey in the form of multiset rewrite rules in Chapter 4, as well as in our calculus (in the following Section 5.5) confirms that our translation is needed to derive the better model, i. e., the model including the necessary locking etc. The translation of the protocol from our calculus produces a total of 49 rules (as opposed to the four rules in the manual encoding). While in this case the security results could be confirmed (given that the locks were put in the right places), this may not always be the case. Instead of expecting the protocol modeller to describe the protocol in such tedious detail, we suggest a tool that performs this task while preserving the soundness and completeness present in the tamarin-prover.

5.3.3.2 *Modelling the store using actions instead of facts*

The store is presently modelled in the form of Insert and Delete events that appear in the trace. Since in multiset rewriting, the state of a protocol is the multiset of facts at a given point in time, the reader might ask why the store is not translated to facts, for instance, insert x,y could result in a fact $\mathsf{Store}(x, y)$. We faced the following problem: Store needs to be a linear fact, since we want to be able to remove it at some point. A rule in the translation of some part of the protocol that performs a lookup to this value will need to contain $\mathsf{Store}(x, y)$ on the left-hand side (so the database lookup is performed) as well as on the right-hand side (so it is not removed afterwards). Such a translation does not interact well with tamarin's constraint solving algorithm. Without going too much into the details of how tamarin's constraint solving algorithm works, we will sketch the parts relevant for this argument: Tamarin translates the negation of the security property into a constraint system, which it then tries to solve, i. e., it tries to find a counter-example.

It refines this constraint system until it is either solved (and the security property invalidated), or until all possible cases of the refinement are contradictory. Trace formulas resulting from the negated security property, as well as from previously established lemmas are part of the constraint system as well as *graph constraints*. They describe protocol traces in a way similar to Figure 13: A node describes an instantiation of a multiset rewrite rule, including its premises, actions and conclusions. An edge may connect the conclusion of an instantiated multiset rewriting rule with a matching premise in another instantiation of a possibly different multiset rewriting rule. (There are other kinds of edges which are relevant for message deduction, but unimportant for this argument.) The algorithm performs backward search: The negation of the security property typically postulates the existence of one or two actions in the trace (see for example ϕ in the previous paragraph). This will be refined into a graph constraint

consisting of a node that corresponds to an instance of a multiset rewriting rule that carries this action. Often, this node will have open premises, which will result in a case distinction over multiset rewrite rules that can be instantiated to have a matching conclusion. Those nodes may haven open premises themselves, which may result in another case distinction, etc.

Now back to the example. Whenever there is an open premise for a fact $\mathsf{Store}(x, y)$, there is a case distinction over all multiset rewriting rules that have *Store* as a conclusion, including the rule resulting from a translation of a lookup. But, this rule has *Store* in the conclusion as well as in the premise, which is provokes a loop in tamarin's backward search. Our modelling using the actions allows us to establish a connection between a lookup and the (uniquely defined) insert that this value in the store originated from, i. e., an insert with corresponding key and value, which has neither been overwritten nor deleted until the lookup. This is precisely what we encode in the axiom α_{in}– we have not found a way to express this using only multiset rewrite rules.

We think that it is possible to avoid the loop mentioned above despite using linear facts as a store, but it would require extending tamarin's constraint solving algorithm. There could be a syntactic element to mark linear facts that shall only be verified, but not removed by a multiset rewriting rule. This could be reflected in a graph node that has Store as a read-only premise, but not as part of the conclusion. Read-only premises would need to be connected to a matching conclusion, but a conclusion of this linear fact can be connected to more than one read-only premise in the graph. As a side condition, any premise that may consume the fact, i. e., any not-read-only premise, must appear either before the node with the *Store* in a conclusion, or after the last node with *Store* in a read-only premise. Adding this extension to the constraint solving algorithm and comparing this modelling of state with our current modelling appears to be an interesting line of future research.

5.3.3.3 *Axioms*

The axioms in the translation of the formula are designed to work hand in hand with the translation of the process into rules. They express the correctness of traces with respect to our calculus' semantics, but are also meant to guide tamarin's constraint solving algorithm. In the following, we will discuss how the axioms achieve those goals. The axiom α_{init} assures that the initial rule, which is the only rule annotated with the action $\mathsf{Init}()$, appears once in a trace. Unless the top-level construct of the process is a replication, $\mathsf{state}_{[]}()$ is a linear fact, and can hence only be consumed once. The axioms α_{eq} and α_{noteq} are straight-forward. On the other hand, α_{in} and α_{notin} illustrate what kind of axioms work well. In the case of α_{in}, when a node with the ac-

tion IsIn is created (by definition of the translation, this corresponds to a lookup command), the existential translates into a graph constraint that postulates an insert node for the value fetched by the lookup, and two formulas that assure that *a*) this insert node appears before the lookup, *b*) this insert node is uniquely defined, that is, it is the last insert to the corresponding key. Further *c*) there is not delete in between. Due to this side conditions, α_{notin} only adds one Insert node per IsIn node – the case where an axiom postulates a node, which itself allows for postulating yet another node needs to be avoided, as tamarin runs into loops otherwise.

Similarly, α_{notin}, produces two cases, if it can be applied due to the existence of an IsNotSet node. The first one is coupled with a very strong condition, namely, that no previous insert to this key exists. This case can usually be refuted quickly. In the second case, it must be assumed that a Delete node exists, but that there is no subsequent Insert. In the majority of our case studies, the second case was refuted immediately, since no delete command appeared during the process. Thus, we were left with a very strong assumption (no insert prior to the lookup), which often led to quick termination.

A naïve way of implementing locks using an axiom would be the following (to simplify the presentation, we will assume there to be just one global lock):

> Every lock happens either after an unlock, and there is neither a lock, nor an unlock in between, or it is the first lock, so there is no previous lock and no unlock at all.

This axiom would guide tamarin in the following way: If a lock is postulated, a case distinction is made. In the first case, an unlock is postulated, and a constraint stored which says that there are no locks and unlocks between the two. Then, tamarin tries to resolve the missing premises, in particular the state-facts, which will eventually lead to another lock, since the well-formedness condition imposes every unlock be preceded (in the sense of the depth in the process tree) by a lock. The axiom applies again, etc. ad infinitum.

The axiom α_{lock} avoids this caveat, first and foremost because it only applies to pairs of locks. We will outline how α_{lock} is applied during the constraint solving procedure:

1. If there are two locks for the same term and with possibly different annotations, an unlock for the first of those locks is postulated, more precisely, an unlock with the same term, the same annotation and no lock or unlock for the same term in-between. The axiom itself contains only one case, so the only case distinction that takes place is over which rule produces the matching Unlock-action. However, due to the annotation, all but one are refuted immediately in the next step. Note further that α_{lock} postulates only a single node, namely the node with the action Unlock.

2. Due to the annotation, the fact $\mathsf{state}_p(\ldots)$ contains the fresh name that instantiates the annotation variable. Let $a : \mathit{fresh}$ be this fresh name. Every fact $\mathsf{state}_{p'}(\ldots)$ for some position p' that is a prefix of p and a suffix of the position of the corresponding lock contains this fresh name. Furthermore, every rule instantiation that is an ancestor of a node in the dependency graph corresponds to the execution of a command that is an ancestor in the process tree. Therefore, the backward search eventually reaches the matching lock, including the annotation, which is determined to be a, and hence appears in the Fr-premise.

3. Because of the Fr-premise, any existing subgraph that already contains the first of the two original locks would be merged with the subgraph resulting from the backwards search that we described in the previous step, as otherwise $\mathsf{Fr}(a)$ would be added at two different points in the execution.

4. The result is a sequence of nodes from the first lock to the corresponding unlock, and graph constraints restricting the second lock to not take place between the first lock and the unlock. We note that the axiom α_{lock} is only instantiated once per pair of locks, since it requires that $i \lessdot j$, thereby fixing their order.

In summary, the annotation helps distinguishing which unlock is expected between to locks, vastly improving the speed of the backward search. This optimisation, however, required us to put restrictions on the locks, which are the subject of the next section.

5.3.3.4 *Restrictions on lock/unlock*

The annotation of locks restricts the set of protocols we can translate, but allows us to obtain better verification results, since we can predict which unlock is "supposed" to close a given lock. This additional information is helpful for tamarin's backward reasoning. We think that our locking mechanism captures all practical use cases for locking. However, since our calculus supports inline multiset rewrite rules, the user is free to implement locks herself, e.g., as follows:

$$[\mathrm{NotLocked}()] \to []; \mathit{code}; [] \to [\mathrm{NotLocked}()].$$

Note that in this case the user does not benefit from the optimisation we put into the translation of locks.

Obviously, locks can be modelled in both tamarin's multiset rewriting calculus in tamarin (this is actually what the translation does) and Mödersheim's set rewriting calculus [74]. Hence there are no restrictions on where locks appear, however, they must be implement by hand, for example using a predicat as above. Therefore, there are no restrictions, but no optimisations either. To the best of our knowledge, StatVerif is the only comparable tool that models locks explicitly. As

pointed out in Section 5.2.3, the restriction in StatVerif are strictly stronger than in our calculus.

5.3.3.5 *Well-formedness of translation*

The third condition of Definition 2 does not hold for the translation of the lookup operator. In order to be able to use tamarin as a back-end, we need to show that, under this specific conditions, the solution procedure is still correct. This is formally proven in Appendix B.1. The idea is the following: A modified translation $[\![\cdot]\!]^{\mathsf{D}}$ which is defined like the current translation, but adds a dummy-fact the conclusion when translating an insert, as well as to the premises of a lookup, produces exactly the same set of traces that are correct with respect to the axioms. We use this fact to prove that, if tamarin's constraint solving algorithm would miss a trace for the original, not well-formed translation, it would miss a trace for the dummy-translation too, which would contradict tamarin's correctness as proven by Schmidt, Meier et al. [89, 88, 69].

5.4 CORRECTNESS OF THE TRANSLATION

In this chapter, we will show the correctness of our translation stated by the following theorem.

Theorem 1: Given a well-formed ground process P and a well-formed trace formula φ we have that

$$\mathit{traces}^{pi}(\mathsf{P}) \vDash^{\star} \varphi \text{ iff } \mathit{traces}^{msr}([\![\mathsf{P}]\!]) \vDash^{\star} [\![\varphi]\!]_{\star}$$

where $\star$ is either $\forall$ or $\exists$.

We will first give an overview of the main propositions and lemmas needed to prove Theorem 1. We first introduce two functions on traces. The first one, *filter*, removes all traces that do not satisfy the trace formula α, i. e., our axioms from page 77. The second one, *hide*, removes all reserved actions from each trace in the set. We will show that the set of traces in $\mathit{traces}^{pi}(\mathsf{P})$ is equal to the set of traces that result from the translation $\mathit{traces}^{msr}([\![\mathsf{P}]\!])$, after *filter* and *hide* have been applied. It is *not* the case that $\mathit{traces}^{pi}(\mathsf{P})$ equals $\mathit{traces}^{msr}([\![\mathsf{P}]\!])$, as we will see in the following.

Example 18: Consider the process

$$\mathsf{P} := \text{if 'apple'='orange' then event } A() \text{ else } 0.$$

There is no trace containing $A()$ in $\mathit{traces}^{pi}(\mathsf{P})$, as the semantics of our calculus cannot reduce this process to its then branch. However, there

is a trace containing $A()$ in $traces^{msr}([\![P]\!])$. Observe that $[\![P]\!]$ contains of the following rules:

$$\begin{aligned}[\![P]\!] = \{&[] \dashv[]\rightarrow \mathsf{state}_{[]}(),\\ &\mathsf{state}_{[]}() \dashv[\mathrm{Eq}(\text{'apple'},\text{'orange'})]\rightarrow \mathsf{state}_1(),\\ &\mathsf{state}_1() \dashv[\mathrm{Event}(), A()]\rightarrow \mathsf{state}_{1,1}(),\\ &\mathsf{state}_{1,1}() \dashv[]\rightarrow [],\\ &[] \dashv[\mathrm{NotEq}(\text{'apple'},\text{'orange'}]\rightarrow \mathsf{state}_2(),\\ &\mathsf{state}_2() \dashv[]\rightarrow []\}\end{aligned}$$

There is indeed a trace that contains $A()$, for example the trace

$$\left[\{\,\mathrm{Eq}(\text{'apple'},\text{'orange'})\,\}^{\#},\{\,\mathrm{Event}(),A()\,\}^{\#}\right].$$

However, this trace does not satisfy the axiom α_{eq}, which means that it is not important for the verification of translated security properties, as we will show in the following.

The function *filter* removes traces that are incorrect with respect to our axioms α.

Definition 16: Let α be the trace formula as defined in Definition 15 and Tr a set of traces. We define

$$\mathit{filter}(Tr) := \{tr \in Tr \mid \forall\theta.(tr,\theta) \vDash \alpha\}$$

The following proposition states that if a set of traces satisfies the translated formula, then the filtered traces satisfy the original formula, too.

Proposition 1: Let Tr be a set of traces and φ a trace formula. We have that

$$Tr \vDash^{\star} [\![\varphi]\!]_{\star} \text{ iff } \mathit{filter}(Tr) \vDash^{\star} \varphi$$

where $\star$ is either $\forall$ or $\exists$.

Proof. We first show the two directions for the case $\star = \forall$. We start by showing that $Tr \vDash^{\forall} [\![\varphi]\!]$ implies $\mathit{filter}(Tr) \vDash \varphi$.

$$\begin{aligned} Tr \vDash^{\forall} [\![\varphi]\!]_{\forall} &\Rightarrow \mathit{filter}(Tr) \vDash^{\forall} [\![\varphi]\!]_{\forall} && \text{(since } \mathit{filter}(Tr) \subseteq Tr\text{)}\\ &\Leftrightarrow \mathit{filter}(Tr) \vDash^{\forall} \alpha \Rightarrow \varphi && \text{(by definition of } [\![\varphi]\!]_{\forall}\text{)}\\ &\Leftrightarrow \mathit{filter}(Tr) \vDash^{\forall} \varphi && \text{(since } \mathit{filter}(Tr) \vDash^{\forall} \alpha\text{)}\end{aligned}$$

We next show that $\mathit{filter}(Tr) \vDash^{\forall} \varphi$ implies $Tr \vDash^{\forall} [\![\varphi]\!]_{\forall}$.

$$\begin{aligned} \mathit{filter}(Tr) \vDash^{\forall} \varphi &\Rightarrow \mathit{filter}(Tr) \vDash^{\forall} \alpha \wedge \varphi && \text{(since } \mathit{filter}(Tr) \vDash^{\forall} \alpha\text{)}\\ &\Leftrightarrow Tr \vDash^{\forall} \neg\alpha \vee (\alpha \wedge \varphi)\\ &\qquad \text{(since } \mathit{filter}(Tr) \subseteq Tr \text{ and } (Tr \setminus \mathit{filter}(Tr)) \nvDash^{\forall} \alpha\text{)}\\ &\Leftrightarrow Tr \vDash^{\forall} \alpha \Rightarrow \varphi\\ &\Leftrightarrow Tr \vDash^{\forall} [\![\varphi]\!]_{\forall} && \text{(by definition of } [\![\varphi]\!]_{\forall}\text{)}\end{aligned}$$

The case of $\star = \exists$ now easily follows:

$$Tr \vDash^{\exists} [\![\varphi]\!]_{\exists} \text{ iff } Tr \nvDash^{\forall} [\![\neg\varphi]\!]_{\forall} \text{ iff } \mathit{filter}(Tr) \nvDash^{\forall} \neg\varphi \text{ iff } \mathit{filter}(Tr) \vDash^{\exists} \varphi.$$

□

If we take a second look at Example 18, we see that the traces in $\mathit{traces}^{msr}([\![P]\!])$ contain some actions that never appear in any trace in $\mathit{traces}^{pi}(P)$, namely Eq, NotEq, and Event(). Observe that all these actions are in $\mathcal{F}_{res}$. The trace [NotEq('apple', 'orange')] is such a case (NotEq $\in \mathcal{F}_{res}$), although it satisfies the axioms α, including α_{noteq}, and hence belongs to $\mathit{filter}(\mathit{traces}^{msr}([\![P]\!]))$. We need to "hide" all those actions that are reserved in order to have equivalent sets of traces. Since reserved actions do not appear in well-formed trace formulas, it is safe to hide them. We define the *hiding* operation which filters out all reserved facts from a trace.

Definition 17 (*hide*)*:* Given a trace tr and a set of facts F we inductively define $\mathit{hide}([]) = []$ and

$$\mathit{hide}(F \cdot tr) := \begin{cases} \mathit{hide}(tr) & \text{if } F \subseteq \mathcal{F}_{res} \\ (F \setminus \mathcal{F}_{res}) \cdot \mathit{hide}(tr) & \text{otherwise} \end{cases}$$

Given a set of traces Tr we define $\mathit{hide}(Tr) = \{\mathit{hide}(t) \mid t \in Tr\}$.

As expected, well-formed formulas that do not contain reserved facts evaluate the same whether reserved facts are hidden or not.

Proposition 2*:* Let Tr be a set of traces and φ a well-formed trace formula. We have that

$$Tr \vDash^{\star} \varphi \text{ iff } \mathit{hide}(Tr) \vDash^{\star} \varphi$$

where $\star$ is either $\forall$ or $\exists$.

Proof. We start with the case $\star = \exists$ and show the stronger statement that for a trace tr

$$\forall\theta.\exists\theta'.\text{ if } (tr, \theta) \vDash \varphi \text{ then } (\mathit{hide}(tr), \theta') \vDash \varphi$$

and

$$\forall\theta.\exists\theta'.\text{ if } (\mathit{hide}(tr), \theta) \vDash \varphi \text{ then } (tr, \theta') \vDash \varphi.$$

We will show both statements by nested induction on $|tr|$ and the structure of the formula. (The underlying well-founded order is the lexicographic ordering of the pairs consisting of the length of the trace and the size of the formula.)

If $|tr| = 0$ then $tr = []$ and $tr = \mathit{hide}(tr)$ which allows us to directly conclude letting $\theta' := \theta$.

If $|tr| = n$, we define $\overline{tr}$ and F such that $tr = \overline{tr} \cdot F$. By induction hypothesis we have that

$$\forall\overline{\theta}.\exists\overline{\theta}'.\text{ if } (\overline{tr}, \overline{\theta}) \vDash \varphi \text{ then } (\mathit{hide}(\overline{tr}), \overline{\theta}') \vDash \varphi$$

and

$$\forall \overline{\theta}.\exists \overline{\theta}'.\ \text{if } (\mathit{hide}(\overline{tr}), \overline{\theta}) \vDash \varphi \text{ then } (\overline{tr}, \overline{\theta}') \vDash \varphi$$

We proceed by structural induction on φ.

- $\varphi = \bot$, $\varphi = i \lessdot j$, $\varphi = i \doteq j$ or $t_1 \approx t_2$. In these cases we trivially conclude as the truth value of these formulas does not depend on the trace and for both statements we simply let $\theta' := \theta$.
- $\varphi = f@i$. We start with the first statement. Suppose that $(tr, \theta) \vDash f@i$. If $\theta(i) < n$ then we have also that $\overline{tr}, \theta \vDash f@i$. By induction hypothesis, there exists $\overline{\theta}'$ such that $(\overline{tr}, \overline{\theta}') \vDash f@i$. Hence we also have that $(tr, \overline{\theta}') \vDash f@i$ and letting $\theta' := \overline{\theta}'$ allows us to conclude. If $\theta(i) = n$ we know that $f \in tr_n$. As φ is well-formed $f \notin \mathcal{F}_{res}$ and hence $f \in \mathit{hide}(tr)_{n'}$ where $n' = |\mathit{hide}(tr)|$. The proof of the other statement is similar.
- $\varphi = \neg\varphi'$, $\varphi = \varphi_1 \wedge \varphi_2$, or $\varphi = \exists x : s.\varphi'$. We directly conclude by induction hypotheses (on the structure of φ).

From the above statements we easily have that $Tr \vDash^{\exists} \varphi$ iff $\mathit{hide}(Tr) \vDash^{\exists} \varphi$.

The case of $\star = \forall$ now easily follows:

$$Tr \vDash^{\forall} \varphi \text{ iff } Tr \nvDash^{\exists} \neg\varphi \text{ iff } \mathit{hide}(Tr) \nvDash^{\exists} \neg\varphi \text{ iff } \mathit{hide}(Tr) \vDash^{\forall} \varphi$$

□

We can now state our main lemma which is relating the set of traces of a process P and the set of traces of its translation into multiset rewrite rules, using the functions *hide* and *filter* to ignore reserved actions and discard traces that contradict the our axioms.

Lemma 1: Let P be a well-formed ground process. We have that

$$\mathit{traces}^{pi}(P) = \mathit{hide}(\mathit{filter}(\mathit{traces}^{msr}([\![P]\!]))).$$

We will prove this lemma in the following sections, however, for now we proceed to conclude the proof of Theorem 1 from the bird's-eye view.

Given the above propositions and Lemma 1, we can now easily proof our main theorem.

Proof of Theorem 1.

$$\begin{aligned}
&\mathit{traces}^{pi}(P) \vDash^{\star} \varphi \\
&\quad \Leftrightarrow \mathit{hide}(\mathit{filter}(\mathit{traces}^{msr}([\![P]\!]))) \vDash^{\star} \varphi && \text{(by Lemma 1)} \\
&\quad \Leftrightarrow \mathit{filter}(\mathit{traces}^{msr}([\![P]\!])) \vDash^{\star} \varphi && \text{(by Proposition 2)} \\
&\quad \Leftrightarrow \mathit{traces}^{msr}([\![P]\!]) \vDash^{\star} [\![\varphi]\!]_{\star} && \text{(by Proposition 1)}
\end{aligned}$$

□

In the following subsections, we will show that Lemma 1 holds, by showing inclusion of $\mathit{traces}^{pi}(R)$ in $\mathit{hide}(\mathit{filter}(\mathit{traces}^{msr}([\![P]\!])))$ (Lemma 3 in Section 5.4.2), and by showing the reverse inclusion (Lemma 4 and Lemma 5 in Section 5.4.3). But before we are able to show this, we need to establish a lemma about message deduction and fresh values. This is the purpose of the next subsection.

5.4.1 *Lemmas about message deduction*

The following lemma relates the message deduction in our calculus to the message deduction in multiset rewriting using the rules $\{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl}, \text{Fresh}\}$. It will be useful for both directions of the proof of Lemma 1, i. e., for Lemma 3 and Lemma 5.

First, every message m such that $!K(m)$ is part of the current state can be derived from the terms that appear in Out facts. Second, if a message can be derived in our calculus, there is a chain of application using the above mentioned rules, such that $!K(m)$ is part of the state.

Lemma 2: Let P be a ground process and $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_n} S_n \in \mathit{filter}(\mathit{exec}^{msr}([\![P]\!]))$. Let

$$\{t_1, \ldots, t_m\} = \{t \mid \mathsf{Out}(t) \in_{1 \leqslant j \leqslant n} S_j\},$$
$$\sigma = \{^{t_1}/_{x_1}, \ldots, ^{t_m}/_{x_m}\}, \text{ and}$$
$$\tilde{n} = \{a : \mathit{fresh} \mid \mathit{ProtoNonce}(a) \in_E \bigcup_{1 \leqslant j \leqslant n} E_j\}.$$

We have that

1. if $!K(t) \in S_n$ then $\nu\tilde{n}.\sigma \vdash t$;
2. if $\nu\tilde{n}.\sigma \vdash t$ then there exists S such that
   - $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_n} S_n \longrightarrow^* S \in \mathit{filter}(\mathit{exec}^{msr}_E([\![P]\!]))$,
   - $!K(t) \in_E S$ and
   - $S_n \to^*_R S$ for $R = \{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl}, \text{Fresh}\}$.

To prove this lemma, we show the first statement by induction on the number of steps in the execution. For every rule producing a !K-fact, one can show that a corresponding deduction rule (see Definition 9) can be used to derive this term. For the second statement, we perform induction over the deduction tree witnessing $\nu\tilde{n}.\sigma \vdash t$ which we use to construct a sequence of multiset rewriting steps producing $!K(t)$. See Appendix B for the full proof, including the helping lemmas 13 and 14.

5.4.2 *Inclusion I*

To state the next lemma describing the inclusion of $\mathit{traces}^{pi}(P)$ in $\mathit{hide}(\mathit{filter}(\mathit{traces}^{msr}([\![P]\!])))$ we need two additional definitions. The first

one formally defines which set of rules results from the translation of a particular position in the process tree. Note that the rules in Figure 14 are the same as in Figure 12.

Definition 18: Let P be a well-formed ground process and p_t a position in P. We define the set of multiset rewrite rules generated for position p_t of P, denoted $[\![P]\!]_{=p_t}$ as follows:

$$[\![P]\!]_{=p_t} := [\![P, [], []]\!]_{=p_t}$$

where $[\![\cdot,\cdot,\cdot]\!]_{=p_t}$ is defined in Figure 14.

The next definition will be useful to state that for a process P every fact of the form $\mathsf{state}_p(\tilde{t})$ in a multiset rewrite execution of $[\![P]\!]$ corresponds to an active process in the execution of P which is an instance of the subprocess $P|_p$.

Definition 19: Let P be a ground process, $\mathcal{P}$ be a multiset of processes and S a multiset of multiset rewrite rules. We write $\mathcal{P} \leftrightarrow_P S$ if there exists a bijection between $\mathcal{P}$ and the multiset $\{\mathsf{state}_p(\tilde{t}) \mid \exists p, \tilde{t}.\ \mathsf{state}_p(\tilde{t}) \in^\# S\}^\#$ such that whenever $Q \in^\# \mathcal{P}$ is mapped to $\mathsf{state}_p(\tilde{t}) \in^\# S$ we have that

1. $P|_p\tau = Q\rho$, for some substitution τ and some bijective renaming ρ of fresh, but not bound names in Q, and
2. $\exists ri \in_E \mathit{ginsts}([\![P]\!]_{=p}).\ \mathsf{state}_p(\tilde{t}) \in \mathit{prems}(ri)$.

When $\mathcal{P} \leftrightarrow_P S$, $Q \in^\# \mathcal{P}$ and $\mathsf{state}_p(\tilde{t}) \in^\# S$ we also write $Q \leftrightarrow_P \mathsf{state}_p(\tilde{t})$ if this bijection maps Q to $\mathsf{state}_p(\tilde{t})$.

We are now ready to formulate an invariant that implies the inclusion of traces generated using the semantics of our calculus in the fragment of multiset rewriting traces of the translation of process that fulfills the axioms. We show that this invariant holds for every corresponding pair of a state in our calculus, and a state of the multiset rewriting execution. The correspondence is defined by the function f.

Lemma 3: Let P be a well-formed ground process and

$$(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) \xrightarrow{E_1} (\mathcal{E}_1, \mathcal{S}_1, \mathcal{S}_1^{MS}, \mathcal{P}_1, \sigma_1, \mathcal{L}_1) \xrightarrow{E_2} \ldots \xrightarrow{E_n} (\mathcal{E}_n, \mathcal{S}_n, \mathcal{S}_n^{MS}, \mathcal{P}_n, \sigma_n, \mathcal{L}_n)$$

where $(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) = (\emptyset, \emptyset, \emptyset, \{P\}, \emptyset, \emptyset)$. Then, there are $(F_1, S_1), \ldots, (F_{n'}, S_{n'})$ such that

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \ldots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \in \mathit{exec}^{msr}([\![P]\!])$$

and there exists a monotonic, strictly increasing function $f\colon \mathbb{N}_n \to \mathbb{N}_{n'}$ such that $f(n) = n'$ and for all $i \in \mathbb{N}_n$

1. $\mathcal{E}_i = \{a \mid \mathit{ProtoNonce}(a) \in \bigcup_{1 \leqslant j \leqslant f(i)} F_j\}$

$$
\begin{aligned}
[\![0, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x})] \to [] \,\}_{p \stackrel{?}{=} p_t} \\
[\![P \mid Q, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x})] \to [\mathsf{state}_{p\cdot 1}(\tilde{x}), \mathsf{state}_{p\cdot 2}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \\
&\qquad \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \cup [\![Q, p \cdot 2, \tilde{x}]\!]_{=p_t} \\
[\![!P, p, \tilde{x}]\!]_{=p_t} &= \{\, [!\mathsf{state}_p(\tilde{x})] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \\
[\![\nu a; P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x}), \mathsf{Fr}(n_a : \mathit{fresh})] \multimap [\mathit{ProtoNonce}(n_a : \mathit{fresh})] \to \\
&\qquad [\mathsf{state}_{p\cdot 1}(\tilde{x}, n_a : \mathit{fresh})] \,\}_{p \stackrel{?}{=} p_t} \cup [\![P, p \cdot 1, (\tilde{x}, n_a : \mathit{fresh})]\!]_{=p_t} \\
[\![\mathsf{out}(M, N); P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x}), \mathsf{In}(M)] \multimap [\mathrm{InEvent}(M)] \to [\mathsf{Out}(N), \mathsf{state}_{p\cdot 1}(\tilde{x})], \\
&\qquad [\mathsf{state}_p(\tilde{x})] \to [\mathsf{Msg}(M, N), \mathsf{state}_p^{\mathsf{semi}}(\tilde{x})], \\
&\qquad [\mathsf{state}_p^{\mathsf{semi}}(\tilde{x}), \mathsf{Ack}(M, N)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \\
&\qquad \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \\
[\![\mathsf{in}(M, N); P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x}), \mathsf{In}(\langle M, N\rangle)] \multimap [\mathrm{InEvent}(\langle M, N\rangle)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(N))], \\
&\qquad [\mathsf{state}_p(\tilde{x}), \mathsf{Msg}(M, N)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(N)), \mathsf{Ack}(M, N)] \,\}_{p \stackrel{?}{=} p_t} \\
&\qquad \cup [\![P, p \cdot 1, \tilde{x} \cup \mathit{vars}(N)]\!]_{=p_t} \\
[\![\text{if } M = N \text{ then } P &= \{\, [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Eq}(M, N)] \to \quad [\mathsf{state}_{p\cdot 1}(\tilde{x})], \\
\text{else } Q, p, \tilde{x}]\!]_{=p_t} &\qquad [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{NotEq}(M, N)] \to \quad [\mathsf{state}_{p\cdot 2}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \\
&\qquad \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \cup [\![Q, p \cdot 2, \tilde{x}]\!]_{=p_t} \\
[\![\text{event } F; P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Event}(), F] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \\
[\![\text{insert } s, t; P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Insert}(s, t)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \\
[\![\text{delete } s; P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Delete}(s)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \\
[\![\text{lookup } M \text{ as } \nu &= \{\, [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{IsIn}(M, \nu)] \to [\mathsf{state}_{p\cdot 1}(\tilde{M}, \nu)], \\
\text{in } P \text{ else } Q, p, \tilde{x}]\!]_{=p_t} &\qquad [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{IsNotSet}(M)] \to [\mathsf{state}_{p\cdot 2}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \\
&\qquad \cup [\![P, p \cdot 1, (\tilde{x}, \nu)]\!]_{=p_t} \cup [\![Q, p \cdot 2, \tilde{x}]\!]_{=p_t} \\
[\![\mathrm{lock}^l\ s; P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathrm{Fr}(\mathrm{lock}_l), \mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Lock}(\mathit{lock}_l, s)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x}, \mathit{lock}_l)] \,\}_{p \stackrel{?}{=} p_t} \\
&\qquad \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \\
[\![\mathrm{unlock}^l\ s; P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x})] \multimap [\mathrm{Unlock}(\mathit{lock}_l, s)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})] \,\}_{p \stackrel{?}{=} p_t} \cup [\![P, p \cdot 1, \tilde{x}]\!]_{=p_t} \\
[\![[l \multimap [a] \to r]; P, p, \tilde{x}]\!]_{=p_t} &= \{\, [\mathsf{state}_p(\tilde{x}), l] \multimap [\mathrm{Event}(), a] \to [r, \mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(l))] \,\}_{p \stackrel{?}{=} p_t} \\
&\qquad \cup [\![P, p \cdot 1, \tilde{x} \cup \mathit{vars}(l)]\!]
\end{aligned}
$$

Figure 14: Definition of $[\![P, p, \tilde{x}]\!]_{=p_t}$ where $\{\cdot\}_{a \stackrel{?}{=} b} = \{\,\cdot\,\}$ if $a = b$ and $\emptyset$ otherwise.

2. $\forall t \in \mathcal{M}.\, \mathcal{S}_i(t) = \begin{cases} u & \text{if } \exists j \leqslant f(i).\mathsf{Insert}(t,u) \in F_j \\ & \wedge \forall j', u'. \\ & \quad j < j' \leqslant f(i) \Rightarrow \mathsf{Insert}(t,u') \notin_E F_{j'} \\ & \quad \wedge \mathsf{Delete}(t) \notin_E F_{j'} \\ \bot & \text{otherwise} \end{cases}$

3. $\mathcal{S}_i^{MS} = S_{f(i)} \setminus^{\#} \mathcal{F}_{res}$

4. $\mathcal{P}_i \leftrightarrow_P S_{f(i)}$

5. $\{ x\sigma_i \mid x \in \mathbf{D}(\sigma_i) \}^{\#} = \{\mathsf{Out}(t) \in \cup_{k \leqslant f(i)} S_k\}^{\#}$

6. $\mathcal{L}_i =_E \{ t \mid \exists j \leqslant f(i), u.\ \mathsf{Lock}(u,t) \in_E F_j \wedge \forall j < k \leqslant f(i).\ \mathsf{Unlock}(u,t) \notin_E F_k \}$

7. $[F_1, \ldots, F_{n'}] \vDash \alpha$ where α is defined as in Definition 15.

8. $\exists k.\ f(i-1) < k \leqslant f(i)$ and $E_i = F_k$ and $\cup_{f(i-1)<j\neq k\leqslant f(i)} F_j \subseteq \mathcal{F}_{res}$

The first condition expresses that the set of restricted values of the process is the set of nonces that the translation of ν has produced. The second condition expresses that the store corresponds to the Insert and Delete-actions in the translation. The third condition expresses that every non-reserved fact in the state of the translation is a fact in the secondary store of the calculus. The fourth condition makes sure that the set of current processes and the state-facts in the translation correspond, correspond according to Definition 19, i. e., for every state fact that is in the premise of some ground instance of a multiset rewrite rule there is a substitution and a bijective renaming of fresh but unbound names that relates this rule instance to a running process. The fifth condition states that the frame corresponds to the set of terms that have appeared in Out-facts during the protocol execution. The sixth condition expresses that the set of locks corresponds to the Lock and Unlock-actions in the translation. The seventh condition makes sure that the resulting trace respects the axioms α. Finally, the eighth condition expresses that the actions between two corresponding traces are the same, except for reserved actions.

The proof of this lemma is an induction over the number of transitions, following a case distinction over all transitions possible in the semantics of our calculus. In the majority of cases, only a few conditions have to be proven, while the others follow directly from the induction hypothesis. In particular the cases for insert, delete, lookup, lock and unlock are difficult, since the induction hypothesis expresses a semantical interpretation of the actions in the trace, while the axioms α_{in}, α_{notin} and α_{lock} are formulated with the goal of facilitating analysis using tamarin. This semantical correctness of the

axioms and the annotation procedure are therefore part of those cases in the proof. The complete proof can be found in Section B.2.1 in the Appendix.

5.4.3 *Inclusion II*

To simplify the proof for the opposite inclusion, we first define a normal form of a multiset rewriting execution with respect to our translation, see Definition 33 on page 220 in Section B.2.2. A normal execution, e. g., removes semi-states as soon as possible, keeps the rules corresponding to an internal communication together and produces In-facts as late as possible.

We can show that we can bring every execution into this form, which allows us to only reason about "normalized" executions in the proof of Lemma 5.

Lemma 4 (Normalisation)*:* Let P be a well-formed ground process. If

$$S_0 = \emptyset \xrightarrow{E_1}_{[\![P]\!]} S_1 \xrightarrow{E_2}_{[\![P]\!]} \dots \xrightarrow{E_n}_{[\![P]\!]} S_n \in \mathit{exec}^{msr}([\![P]\!])$$

and $[E_1, \dots, E_n] \vDash \alpha$, then there exists a normal MSR execution

$$T_0 = \emptyset \xrightarrow{F_1}_{[\![P]\!]} T_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} T_{n'} \in \mathit{exec}^{msr}([\![P]\!])$$

such that $\mathit{hide}([E_1, \dots, E_n]) = \mathit{hide}(F_1, \dots, F_{n'})$ and $[F_1, \dots, F_{n'}] \vDash \alpha$.

This lemma is proven in Section B.2.2. Like in the previous section, we first define what it means when a set of processes and a multiset correspond to each other:

Definition 20*:* Let P be a ground process, $\mathcal{P}$ be a multiset of processes and S a multiset of multiset rewrite rules. We write $\mathcal{P} \leftrightsquigarrow_P S$ if there exists a bijection between $\mathcal{P}$ and the multiset $\{\mathsf{state}_p(\tilde{t}) \mid \exists p, \tilde{t}.\ \mathsf{state}_p(\tilde{t}) \in^{\#} S\}^{\#}$ such that whenever $Q \in^{\#} \mathcal{P}$ is mapped to $\mathsf{state}_p(\tilde{t}) \in^{\#} S$, then:

1. $\mathsf{state}_p(\tilde{t}) \in_E \mathit{prems}(R)$ for $R \in \mathit{ginsts}([\![P]\!]_{=p})$.
2. Let θ be a grounding substitution for $\mathtt{state}(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then

   $$(P|_p\tau)\rho =_E Q$$

   for a substitution τ, and a bijective renaming ρ of fresh, but not bound names in Q, defined as follows:

   $$\begin{aligned} \tau(x) &:= \theta(x) && \text{if } x \text{ not a reserved variable} \\ \rho(a) &:= a' && \text{if } \theta(n_a) = a' \end{aligned}$$

When $\mathcal{P} \leftrightsquigarrow_P S$, $Q \in^{\#} \mathcal{P}$ and $\mathsf{state}_p(\tilde{t}) \in^{\#} S$ we also write $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$ if this bijection maps Q to $\mathsf{state}_p(\tilde{t})$. Lemma 5 is similar to Lemma 3 and is proven by induction over the number of transitions, see Section B.2.2 in the Appendix.

Lemma 5: Le P be a well-formed ground process. If

$$S_0 = \emptyset \xrightarrow{E_1}_{[\![P]\!]} S_1 \xrightarrow{E_2}_{[\![P]\!]} \ldots \xrightarrow{E_n}_{[\![P]\!]} S_n \in \mathit{exec}^{msr}([\![P]\!])$$

is normal and $[E_1, \ldots, E_n] \vDash \alpha$ (see Definition 33 and 15), then there are $(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}^{MS}_0, \mathcal{P}_0, \sigma_0, \mathcal{L}_0), \ldots, (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'})$ and $F_1, \ldots, F_{n'}$ such that:

$$(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}^{MS}_0, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) \xrightarrow{F_1} (\mathcal{E}_1, \mathcal{S}_1, \mathcal{S}^{MS}_1, \mathcal{P}_1, \sigma_1, \mathcal{L}_1) \xrightarrow{F_2} \ldots \xrightarrow{F_{n'}} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'})$$

where $(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}^{MS}_0, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) = (\emptyset, \emptyset, \emptyset, \{P\}, \emptyset, \emptyset)$ and there exists a monotonically increasing, surjective function $f\colon \mathbb{N}_n \setminus \{0\} \to \mathbb{N}_{n'}$ such that $f(n) = n'$ and for all $i \in \mathbb{N}_n$

1. $\mathcal{E}_{f(i)} = \{ a \in \mathit{FN} \mid \mathit{ProtoNonce}(a) \in_E \bigcup_{1 \leqslant j \leqslant i} E_j \}$
2. $\forall\, t \in \mathcal{M}.\, \mathcal{S}_{f(i)}(t) =$
$$\begin{cases} u & \text{if } \exists j \leqslant i. \mathrm{Insert}(t,u) \in_E E_j \\ & \quad \wedge \forall j', u'. \\ & \qquad j < j' \leqslant i \Rightarrow \mathrm{Insert}(t,u') \notin_E E_{j'} \\ & \qquad \wedge \mathrm{Delete}(t) \notin_E E_{j'} \\ \bot & \text{otherwise} \end{cases}$$
3. $\mathcal{S}^{MS}_{f(i)} =_E S_i \setminus^{\#} \mathcal{F}_{res}$
4. $\mathcal{P}_{f(i)} \leftrightsquigarrow_P S_i$
5. $\{ x\sigma_{f(i)} \mid x \in \mathbf{D}(\sigma_{f(i)}) \}^{\#} =_E \{ \mathrm{Out}(t) \in \cup_{k \leqslant i} S_k \}^{\#}$
6. $\mathcal{L}_{f(i)} =_E \{ t \mid \exists j \leqslant i, u.\, \mathrm{Lock}(u,t) \in_E E_j$ and $\forall j < k \leqslant i.\ \mathrm{Unlock}(u,t) \notin_E E_k \}$.

Furthermore,

7. $\mathit{hide}([E_1, \ldots, E_n]) =_E [F_1, \ldots, F'_n]$.

Lemma 1 follows now from Lemma 3, Lemma 4 and Lemma 5.

5.5 CASE STUDIES

This section gives an overview of the case studies we performed, including a simple security API similar to PKCS#11 [81], the Yubikey security token (see Chapter 4), the optimistic contract signing protocol by Garay, Jakobsson and MacKenzie (GJM) [46], the left-right encryption example discussed by Arapinis et al. [5] and the key-server example discussed by Mödersheim [74]. The results are summarized in Table 1. For each of the case studies we also provide the number of typing lemmas that were needed by the tamarin prover, and whether manual guidance of the tool was required.

EXAMPLE	LEMMAS	MANUAL
Security API à la PKCS#11	1	no
Yubikey Protocol [60, 93]	3	yes
GJM Contract-Signing protocol [5, 46]	0	no
Mödersheim's Example (lock/insert) [74]	0	yes*
Mödersheim's Example (emb. MSRs) [74]	0	no
Security Device Example [5]	1	no
Needham-Schroeder-Lowe [66]	1	no

* only a small amount of guidance was necessary (7 mouse clicks).

Table 1: Case studies.

5.5.1 *Security API à la PKCS#11*

This example illustrates how our modelling might be useful for the analysis of security APIs in the style of the PKCS#11 standard [81]. We expect studying a complete model of PKCS#11, such as in [36], to be a straightforward extension of our running example in Section 5.2. The actual case study models two additional operations: First, given a handle and a plaintext, the user can request an encryption under the key the handle points to. Second, given an encryption of some value k_2 under a key k_1, and a handle h_1 pointing to k_1, the user can request the encryption to be *unwrapped*, which means it is decrypted under k_1. The result is stored on the device, and she receives a handle h_2 pointing to k_2. Furthermore, new keys are assigned an initial attribute, from which they can move to either 'wrap', or 'unwrap', see the following snippet:

```
in(<'set_dec',h>); lock <'att',h>;
        lookup <'att',h> as a in
               if a='init' then
                       insert <'att',h>, 'dec'; unlock <'att',h>
```

Note that, in contrast to the running example of the previous sections, in this modelling it is necessary to encapsulate the state changes between lock and unlock, as otherwise an adversary could stop the execution after line 3, set the attribute to 'wrap' in another subprocess and therefore be able to produce a wrapping, which he can, after resuming operation at line 4, decrypt and therefore learn a key. Such subtleties can produce attacks which can be detected and mitigated using our modeling. If the locking is handled correctly, we can show secrecy of keys produced on the device. See Listing 22 in Section A.3 for the complete model.

5.5.2 *Needham-Schoeder-Lowe*

We can show secrecy for a session-key established between two honest parties running the Needham-Schroeder-Lowe protocol [65]. The modelling does not require tags on the messages. See Listing 23 in Section A.3 for the complete model.

5.5.3 *Yubikey*

The Yubikey [93] is a small hardware device designed to authenticate a user against network-based services. When the user presses a button on the device, it outputs a one-time-password consisting of a counter, a timestamp and other data encrypted using a key that is on the device. This one-time-password can be used to authenticate against a server that shares the same secret key.

We introduced the Yubikey in Chapter 4 and showed by the means of an injective correspondence property and a formalisation of the protocol in tamarin's multiset rewriting calculus, that an attacker that controls the network cannot perform replay attacks. The modelling in this chapter is more fine-grained due to the use of our calculus, which makes security-relevant operations like locking and tests on state explicit. The resulting model is closer to the real-life operation of such a device and the server component. The modeling of the Yubikey takes approximately 38 lines in our calculus, which translates to 49 multiset rewrite rules. The modelling from Chapter 4 contains only four rules, which are quite complicated in themselves, resulting in 23 lines of code. From the new modelling, we learn that the server can treat multiple authentication requests in parallel, as long as they do not claim to stem from the same Yubikey. An implementation on the basis of the model from Chapter 4 would need to implement a global lock accessible to the authentication server and all Yubikeys to be on the safe side, since in the MSR model, each of the four rules is atomic. This is of course impossible: First, the Yubikeys are likely to be used at different places around the world, so it is unlikely that there exist means of direct communication between them (the Yubikey "types" in an OTP in a web-form for the user, i. e., it does not send anything on the network itself), second, there are typically many Yubikeys deployed at once, so locking the authentication between the moment the button on the Yubikey is pressed and the moment the OTP is send on the network is not only unrealistic, but highly inefficient, too. Thirdly, and most importantly, the Yubikey does not have support for such a mechanism, so while a server-side global lock might be conceivable (albeit impractical for the reason previously mentioned), a real global lock could not be implemented for the Yubikey as deployed. In Section 5.3.3, we argue that such locking issues are far from trivial to detect. We refer to Listing 24 in Section A.3 for the complete model.

5.5.4 *The GJM contract signing protocol*

A contract signing protocol allows two parties to sign a contract in a fair way: None of the participants should be bound to the contract without the other participant being bound as well. A straightforward solution is to use a trusted party that collects both signatures on the contract and then sends the signed contracts to each of the participants. Optimistic protocols have been designed to avoid the use of a trusted party whenever possible (optimizing efficiency, and avoiding the potential cost of a trusted party). In these protocols the parties first try to simply exchange the signed contracts; in case of failure or cheating behavior of one of the parties, the trusted party can be contacted. Depending on the situation, the trusted party may either *abort* the contract, or *resolve* it. In case of an abort decision the protocol ensures that none of the parties obtains a signed contract, while in case of a resolve the protocol ensures that both participants obtain the signed contract. For this the trusted party needs to maintain a database with the current status of all contracts (aborted, resolved, or no decision has been taken). In our calculus, the status information is naturally modelled using our insert and lookup constructs. The use of locks is also crucial here to avoid the status to be changed between a lookup and an insert.

The contract signing protocol by Garay, Jakobsson and MacKenzie was also studied by Arapinis et al. [46, 5]. They showed the crucial property that the same contract may never be both aborted and resolved. However, due to the fact that StatVerif only allows for a finite number of memory cells, they have shown this property for a single contract and provide a manual proof to lift the result to an unbounded number of contracts. We directly prove this property for an unbounded number of contracts in the model described in Listing 25 in Section A.3.

5.5.5 *Further Case Studies*

We investigated the case study presented by Mödersheim [74] and Arapinis et al. [5], in order to be able to compare our approach to each of them. For Mödersheim's key-server example, we encoded two models of this example, one using the insert construct, the other manipulating state using the embedded multiset rewrite rules. For this example the second model turned out to be more natural and more convenient, allowing for a direct automated proof without any additional typing lemma (Listing 26 in Section A.3, Listing 27 for the modelling using insert and lookup).

The modelling of the left-right encryption example by Arapinis et al., also described in Example 4, was straight-forward (Listing 28 in

Section A.3). In both cases we were able to re-use the typing lemmas from the tamarin models of those protocols by Simon Meier [71, 68].

Part II

WHEN IS A SECURITY API "SECURE"?

We introduce and discuss a definition of security for security API implementations. This definition is presented in form of the first universally composable key-management functionality, formalized in Hofheinz' and Shoup's GNUC framework. GNUC and similar frameworks define the security of protocols by comparison to special network entities accessed by secure channels which express the "ideal" of whatever computation the protocol should define. This approach allows for composability, which means that a "secure" protocol can be substituted by the functionality that defines its security in any context. The key-management functionality enforces a range of security policies and can be extended by key usage operations with no need to repeat the security proof. We illustrate its use by proving an implementation of a security token secure with respect to arbitrary key-usage operations and explore a proof technique that allows the storage of cryptographic keys externally.

6

WHEN IS A SECURITY API SECURE?

In this chapter, we discuss the characteristics of security APIs and criteria we require in order to call a security API "secure". Deciding whether a given definition is good is difficult, therefore we discuss what makes a good definition before we try to characterize the nature of security APIs in a cryptographic context. This helps establishing a basis for the design decisions we make and provides for criteria for the evaluation of our definition.

6.1 CRITERIA FOR PERSUASIVE DEFINITIONS

The following five rules are traditionally used criteria for a certain kind of definition [28, 54, 53]. With the aim of deriving a definition that is commonly acceptable, we will regard those rules as desirable properties of a good definition of security.

A. It should set out the essential attributes of what it defines.

B. It should not be too wide, e. g., it should not include security APIs that are insecure.

C. It should not be too narrow.

D. It should not be obscure.

E. It should not be negative where it could be positive.

Of course, security definitions have slightly different constraints although we think that those properties provide a good starting point. We would like to add an additional constraint, which is very important for a security definition: It should be possible to work with the definition, by which we means that it should both be possible to show whether an object is secure with respect to the definition, and it should be possible to use the fact that an object is secure with respect to the definition for the analysis of other objects.

F. Security definitions should be applicable.

Furthermore, security definitions usually put a lot of emphasis on the second property. Since security definitions that are too wide might give a false sense of security and thus lead to attacks, which are costly, the general consensus is to try to be "on the safe side". In practice, this means that often clarity, and sometimes generality, are sacrificed to that end – the definition we propose is not different in this regard. The fifth property, a definition should not be negative, where it could

be positive, is usually easy to fulfill: A secure cryptographic primitive is typically an algorithm, or a set of algorithms, that has a number of properties.

6.2 CHARACTERISTICS OF A "SECURE" SECURITY API

The first property brings us back to one of the questions we have posed in the introduction: What characterises cryptographic security APIs? First, they provide an interface to cryptographic keys that are stored in some secure memory. Second, this interface allows to indirectly use these keys to compute cryptographic functions possibly depending on one or more secrets, randomness, and the user's input. Third, the access to those secrets can be restricted, for example by the configuration of the device.

What characterises the security of such devices? First, it should not reveal secrets to the user. Second, the cryptographic functions should be secure – what this means depends on the cryptographic function. Third, the *intended* access restrictions to the secrets should be respected. Fourth, the previous properties should hold in a network where the user might be malicious, and where several security APIs might be present and might contain the same secrets.

6.3 COMPOSABILITY

The fourth point suggests the use of a framework that supports universal composability, i. e., a framework that preserves the security of a component in a network even under (parallel) composition with a second components. The GNUC ("GNUC is Not UC") framework [51], which we will introduce in Chapter 7, is such a framework. Composability is helpful for the analysis of higher-level protocols, which is an appealing feature for security definitions in particular. Consider the following perspective on security APIs: Hosts in a network can easily be compromised, since it is very difficult to secure general-purpose machines against attacks on the implementation level. Taking scenarios where parties are possibly under adversary control into account, it is often impossible to show protocols secure. Let us modify the protocol as follows: Instead of performing the cryptographic operations in the protocol themselves, the parties have security APIs which perform those computations for them. In the case of GNUC, the security API would be an incorruptible entity in the network that the parties – corrupted or not – have access to. Maybe, assuming trust on the security APIs but not the parties, we can show interesting security properties for this protocol. Through the use of a framework that allows for composability, we can substitute the security APIs by a far more abstract object, a functionality (cf. Chapter 7), which helps performing this kind of proof.

6.4 OVERVIEW

In Chapter 7, we will introduce the GNUC framework. In Chapter 8, we introduce our security definition, which tries to incorporate the characterisation of a security API and its security we sketched above, with the aim of deriving a definition that "sets out the essential attributes of what it defines". In Chapter 9, we analyse this security definition. We will prove a lemma which states that many guarantees one should expect from our definitions are indeed provided to support the claim that this definition is not too wide. In the same chapter, we discuss the limitations of our approach, and evaluate it with respect to the properties mentioned before, discussing in particular whether the definition too narrow or too obscure.

In Chapter 8, we furthermore introduce generic architecture for security APIs. It can serve as a guideline to the design of security APIs and is independent of how cryptographic functions are implemented, as long as they are known upfront and the implementation is secure in itself. In order to achieve this, we make use of the composability of the underlying framework. We show in Chapter 10, that this generic implementation is secure with respect to our definition. In Chapter 11, we will provide an example of how to use the secure implementation in a higher-level protocol, supporting the claim that our definition is applicable in the sense of being useful for the analysis of other protocols.

The contributions that we will present in the following chapters are joint work with Steve Kremer and Graham Steel and were published at ESORICS 2013 [57].

6.5 RELATED WORK

There is recent work that aims at trying to define appropriate security notions for security APIs [21, 30, 58]. Both the proposal by Cachin and Chandran and the proposal by Kremer et al. define security in terms of a cryptographic game, i. e., in both cases, there is a game with a challenger that, depending on a coin toss, either acts like the security API, or acts like an idealized version of the security API, e. g., encrypting random values instead of the actual plaintexts etc. This approach has two major disadvantages: First, it is not clear how the security notion will compose with other protocols implemented by the API. How does the security of a single security API translate to the security of 100 security APIs that may eventually share the same keys? We have previously argued that this is inappropriate since security APIs are first and foremost used as building blocks for larger protocols – therefore, composability is crucial.

Second, it is difficult to see whether a definition covers the attack model completely, since the game may be tailored to a specific API.

For example, the security definition by Cachin and Chandran [21] specifies a list of commands that a security API must support – for many applications this definition is too narrow. Furthermore, it requires that one single central key server is used for a group of users, since the security depends on a complete and current log of all operations. The definition of Cortier and Steel [30] also defines the set of commands that the API must support, albeit it is intended for distributed tokens, as opposed to the definition by Cachin and Chandran. This definition allows only for security proofs in the symbolic model and has a more limited functionality in comparison to Cachin and Chandran's definition. Recent work by Daubignard, Lubicz and Steel extends the interface to public-key cryptography [32]. While public key cryptography is supported by Chachin and Chandran's definition, as well as our definiton, neither supports the use of public-key encryption of key export and import. Daubignard, Lubicz and Steel use signature keys to sign encryptions to guarantee the integrity of keys that are imported onto a device.

The functionality defined in these works is fixed, which means their approach is limited to APIs that provide no more operations than "allowed". This can be done better; to this end it is worthwhile to find out which operations are essential to key-management. In present thesis, we adapt the more general approach to API security of Kremer et al. [58] to a framework that allows for composition. The idea is to separate the task of key-management from the task of key-usage, in order to distinguish the functions that are relevant for security (and therefore have to be defined) from the functions that are not relevant for the security (and can therefore be left open). Kremer et al. give a game-based definition in the computational model, as well as a definition in the symbolic model. We extend the idea of separating key-management and key-usage and transfer it into the framework of GNUC. In fact, the composability of this framework allows us to go one step further and define the security of the key-usage operations without fixing the set of key-usage operations in our definition.

Some aspects of the functionality $\mathcal{F}_{\text{crypto}}$ by Küsters et al. [61] are similar to our key-management functionality in that they both provide cryptographic primitives to a number of users and enjoy composability. However, the $\mathcal{F}_{\text{crypto}}$ approach aims at abstracting a specified set of cryptographic operations on client machines to make the analysis of protocols in the simulation-based security models easier, and addresses neither key-management nor policies.

7 INTRODUCTION TO GNUC

The GNUC framework, recently proposed by Hofheinz and Shoup [51], is a variant of the UC framework [23] and as such a framework for simulation-based security. The requirements on a protocol are formalized by abstraction: An *ideal functionality* computes the protocol's inputs and outputs securely, while a 'secure' protocol is one that emulates the ideal functionality. Simulation-based security naturally models the composition of the API with other protocols, so that proofs of security can be performed in a modular fashion. We decided to use the GNUC model because it avoids shortcomings of the original Universal Composability (UC) framework which have been pointed out over the years. Moreover, the GNUC framework is well structured and well documented resulting in more rigorous and readable security proofs.

Hofheinz and Shoup proposed GNUC to address several known shortcomings in UC. In particular, in UC the notion of a poly-time protocol implies that the interface of a protocol has to contain enough input padding to give sub-protocols of the implementation enough running time, hence the definition of an interface that is supposed to be abstract depends on the complexity of its implementation. Moreover, the proof of the composition theorem is flawed due to an inadequate formulation of the composition operation [51], though here the authors remark that, "none of the objections we raise point to gaps in security proofs of existing protocols. Rather, they seem artifacts of the concrete technical formulation of the underlying framework". These shortcomings are also addressed to a greater or lesser extent by other altenative frameworks [84, 67]: We chose GNUC because it is similar in spirit to the original UC yet rigorous and well documented. We now give a short introduction to GNUC and refer the reader to [51] for additional details.

7.1 PRELIMINARIES

Let Σ be some fixed, finite alphabet of symbols. We note η the security parameter.

Definition 21 (probabilistic polynomial-time (PPT))*:* We say that a probabilistic program A runs in polynomial-time if the probability that A's runtime on an input of length n is bounded by a polynomial in n is 1. If so, we say such a program is PPT.

Definition 22 (Computationally indistinguishable)*:* Let $X := \{X_\eta\}_\eta$ and $Y := \{Y_\eta\}_\eta$ be two families of random variables, where each

random variable takes values in a set $\Sigma^* \cup \{\perp\}$. We say that X and Y are *computationally indistinguishable*, written $X \approx Y$, if for every PPT program D that takes as input a string over Σ we have that

$$|\Pr[D(x) = 1 | x \leftarrow X_\eta] - \Pr[D(y) = 1 | y \leftarrow Y_\eta]|$$

is negligible in η.

7.2 MACHINES AND INTERACTION

In GNUC a protocol π is modeled as a library of programs, e. g., a function from protocol names to code. This code will be executed by interactive Turing machines. There are two distinguished machines, the environment and the adversary, that π does not define code for. All other machines are called *protocol machines*. Protocol machines can be divided into two subclasses: *regular* and *ideal* protocol machines. They come to life when they are called by the environment and are addressed using machine ids. A machine ID `<pid,sid>` contains two parts: the party ID `pid`, which is of the form `<reg,basePID>` for regular protocol machines and `<ideal>` for ideal protocol machines, and the session ID `sid`. Session ids are structured as pathnames of the form $\langle \alpha_1, \ldots, \alpha_k \rangle$. The last component α_k must be of the particular form *protName*, *sp*. When the environment sends the first message to a protocol machine, a machine running the code defined by the protocol name *protName* is created. The code will often make decisions based on the session parameter *sp* and the party ID. A machine M, identified by its machine ID $\langle pid, \langle \alpha_1, \ldots, \alpha_k \rangle \rangle$, can call a subroutine, i.e., a machine with the machine ID $\langle pid, \langle \alpha_1, \ldots, \alpha_k, \alpha_{k+1} \rangle \rangle$. We then say that M is the *caller* with respect to this machine. Two protocol machines, regular or ideal, are *peers* if they have the same session ID. Programs have to declare which other programs they will call as subroutines, defining a static call graph which must be acyclic and thus have a program r with in-degree 0 – then we say that the protocol is rooted at r.

GNUC imposes the following communication constraints on a regular protocol machine M: It can only send messages to the adversary, to its ideal peer (i. e., a machine with party ID `<ideal>` and the same session ID), its subroutines and its caller. If the caller is the environment, a sandbox mechanism translates its machine ID, which is simply `<env>`, to the machine ID of the caller of M (which is uniquely defined). As a consequence, regular protocol machines cannot talk directly to regular peers. They can communicate via the adversary, which models an insecure network, or via the ideal peer. This ideal peer is a party that can communicate directly with all regular protocol parties and the adversary.

The code of the machines is described by a sequence of steps similarly to [51, § 12]. Each step is defined by a block of the form

```
name [condition]:  P
```

The label `name` identifies the step. The logical expression `[condition]` is a *guard* that must be satisfied to trigger a step. We omit the guard in case it is true. A step name in the guard expression evaluates to true if the corresponding step was triggered at some previous point. `P` is the code (whose semantics we expect to be clear) to be executed whenever the guard evaluates to true. In particular `P` may contain *accept-clauses* that describe the form of the message that can be input. The accept clause, too, might have logical conditions that must be satisfied in order to continue the execution of the step. Any message not triggering any step is processed by sending an error message to A. Listing 1 in the next section gives an example of a functionality written in this notation.

7.3 DEFINING SECURITY VIA IDEAL FUNCTIONALITIES

As in other universal composability frameworks, the security of a protocol is specified by a so-called *ideal functionality*, which acts as a third party and is trusted by all participants. Formally, an ideal functionality is a protocol that defines just one protocol name, say r. The behavior defined for this protocol name depends on the type of machine: All regular protocol machines act as "dummy parties" and forward messages received by their caller (which might be the environment) to their ideal peer. The ideal peer (the machine with the same session ID and party ID `<ideal>`, see Section 7.2) interacts with the regular parties and the adversary: Using the inputs of the parties, the ideal functionality defines a secure way of computing anything the protocol shall compute, explicitly computing the data that is allowed to leak to the attacker.

Example 19: For instance, an authenticated channel is specified as a functionality that takes a message from Alice and sends it to the attacker, exposing its content to the network, but only accepting a message from the attacker (the network) if it is the same message Alice sent in the first place.

We present the following formulation of this functionality, which is very similar to the one presented in [51, §12.1.1] and which we will later come back to:

Listing 1: $\mathcal{F}_{\text{ach}}$ with session parameters $\langle \mathrm{P}_{pid}, \mathrm{Q}_{pid}, label \rangle$. Note that in this example, every step can only be executed once.

```
ready-sender: accept <ready> from P;
   send <sender-ready> to A
ready-receiver[¬ready-sender]: accept <ready> from Q;
   send <ready-receiver-early> to A
ready-receiver[ready-sender]: accept <ready> from Q;
   send <receiver-ready> to A;
```

```
send [ready-receiver]: accept <send,x> from P;
   x̄ ← x; send <send,x> to A
done [send]: accept <done> from A;
   send <done> to P
deliver[send]: accept <deliver,x> from A where x = x;
   send <deliver,x̄> to Q
```

We see that a functionality is completely defined by the code run on the ideal protocol machine.

Now we can define a second protocol, which is rooted at r, and does not necessarily define any behaviour for the ideal party, but only for the regular protocol machines. The role of the environment Z is to distinguish whether it is interacting with the ideal system (dummy users interacting with an ideal functionality) or the real system (users executing a protocol). We say that a protocol π *emulates* a functionality $\mathcal{F}$ if for all attackers interacting with π, there exists an attacker, the simulator *Sim*, interacting with $\mathcal{F}$, such that no environment can distinguish between interacting with the attacker and the real protocol π, or the simulation of this attack (generated by *Sim*) and $\mathcal{F}$. It is actually not necessary to quantify over all possible adversaries: The most powerful adversary is the so-called dummy attacker A_D that merely acts as a relay forwarding all messages between the environment and the protocol [51, Theorem 5].

Let Z be a program defining an environment, i. e., a program that satisfies the communication constraints that apply to the environment (e. g., it sends messages only to regular protocol machines or to the adversary). Let A be a program that satisfies the constraints that apply to the adversary (e. g., it sends messages only to protocol machines (ideal or regular) it previously received a message from). The protocol π together with A and Z defines a structured system of interactive Turing machines (formally defined in [51, § 4]) denoted $[\pi, A, Z]$. The execution of the system on external input 1^η is a randomized process that terminates if Z decides to stop running the protocol and output a string in Σ^*. The random variable $\mathrm{Exec}[\pi, A, Z](\eta)$ describes the output of Z at the end of this process (or $\mathrm{Exec}[\pi, A, Z](\eta) = \bot$ if it does not terminate). Let $\mathrm{Exec}[\pi, A, Z]$ denote the family of random variables $\{\mathrm{Exec}[\pi, A, Z](\eta)\}_{\eta=1}^{\infty}$. An environment Z is well-behaved if the data-flow from Z to the regular protocol participants and the adversary is limited by a polynomial in the security parameter η. We say that Z *is rooted at* r, if it only invokes machines with the same session identifier referring to the protocol name r. We do not define the notion of a *poly-time protocol* and a bounded adversary here due to space constraints and refer the reader to the definition in [51, § 6].

Definition 23 (emulation w.r.t. the dummy adversary)*:* Let π and π' be poly-time protocols rooted at r. We say that π' emulates π if there

exists an adversary *Sim* that is bounded for π, such that for every well-behaved environment Z rooted at r, we have

$$\mathrm{Exec}[\pi, \mathit{Sim}, Z] \approx \mathrm{Exec}[\pi', A_D, Z],$$

where $\approx$ denotes computational indistinguishability (Definition 22).

8

KEY-MANAGEMENT FUNCTIONALITY AND REFERENCE IMPLEMENTATION

In the following, we will develop and discuss a composable notion of secure key-management in the form of a functionality with the name $\mathcal{F}_{\text{KM}}$. To the best of our knowledge, it is the first composable definition of secure key-management.

An ideal functionality should provide the required operations in a way that makes security obvious. This means its design must be as simple as possible in order for this security to be clear. However, there are subtle issues in such designs: Obtaining a satisfactory formulation of digital signature took years because of repeated revisions caused by subtle flaws making the functionality unrealizable. The functionality we define will at some point need to preserve authenticity in a similar way to this signature functionality, but in a multi-session setting. So we must expect a key-management functionality to be *at least* as complex. Nonetheless we aim to keep it as simple as possible, and so justify the inclusion of each feature by discussing what minimum functionality we expect from a key-management system. It will be necessary to compromise at certain points, for example, we decided to only support symmetric encryption as a key-transportation mechanism. The following Chapter 9 provides an analysis of $\mathcal{F}_{\text{KM}}$: In Section 9.1, we will give some properties that any security API that implements $\mathcal{F}_{\text{KM}}$ enjoys. We list limitations in Section 9.2. A discussion of $\mathcal{F}_{\text{KM}}$ takes place in Section 9.3.

The functionality $\mathcal{F}_{\text{KM}}$ formalises the requirements for a security API: It assures that keys are transferred correctly from one security token to another, that the global security policy is respected (even though the keys are distributed on several tokens) and that operations which use keys are computed correctly. The latter is achieved by describing operations unrelated to key-management by so-called key-usage functionalities. $\mathcal{F}_{\text{KM}}$ is parametric in the policy and the set of key-usage functionalities, which can be arbitrary. This facilitates revision of API designs, because changes to operations that are not part of the key-management or the addition of new functions do not affect the emulation proof. This allows the aim of deriving a definition that is wide enough to cover many different forms of security APIs. To achieve this extensibility, we investigate what exactly a "key" means in the context of simulation-based security in Section 8.2.

Common functionalities in such settings do not allow two parties to share the same key. In fact, they do not have a concept of keys, but a concept of "the owner of a functionality" instead. The actual

key is kept in the internal state of a functionality, used for computation, but never output. Dealing with key-management, we need the capability to export and import keys and we propose an abstraction of the concept of keys, that we call *credentials*. The owner of a credential can not only compute a cryptographic operation, but he can also delegate this capacity by transmitting the credential. We think this concept is of independent interest. We subsequently introduce a general proof method that allows the substitution of credentials by actual keys when instantiating a functionality.

We will now proceed to explain the architecture of $\mathcal{F}_{\text{KM}}$, then, in Section 8.2, introduce our concept of key-usage functionalities which covers the usual cryptographic operations we might want to perform with our managed keys. In Section 8.3, we describe our notion of security policies for key-management. In Section 8.4, we formally define $\mathcal{F}_{\text{KM}}$, as well as a generic implementation of $\mathcal{F}_{\text{KM}}$.

8.1 DESIGN RATIONALS

The network we want to show secure has the following structure: A set of users takes input from the environment, each of them is connected to a security token. Each security token is a network entity, just like the users, and has a secure channel to the user it belongs to. Cryptographic keys are stored on the token, but are not given directly to the user – instead, at creation of a key, the user (and thus the environment) receives a handle to the key.

We consider such a network secure if it emulates a network in which the users are communicating with a single entity, the key-management functionality $\mathcal{F}_{\text{KM}}$, instead of their respective security token. It gives the users access to its operations via handles, too, and is designed to model the "ideal" way of performing key-management. To show the security of the operations that have nothing to do with key-management, $\mathcal{F}_{\text{KM}}$ accesses several other functionalities which model the security of the respective operations. This allows us to have a definition that is applicable to many different cases.

8.1.1 *Policies*

As mentioned in Chapter 6, a security API we want to consider secure should respect the intended access restrictions that apply to keys. In order to formalise this intention, we assume there to be a *policy*, which is to be enforced on a global level. Our definition allows the expression of two kinds of requirements: usage policies of the form "key A can only be used for tasks X and Y", and dependency policies of the form "the security of key A may depend on the security of keys B and C". The need for the first is obvious. The need for the second arises because almost all non-trivial key-management systems

allow keys to encrypt other keys, or derive keys by, e. g., encrypting an identifier with a master key. Typically, the policy defines *roles* for keys, i. e., groups of tasks that can be performed by a key, and *security levels*, which define a hierarchy between keys. The difficulty lies in enforcing this policy globally when key-management involves a number of distributed security tokens that can communicate only via an untrusted network. Recall Example 2 from the introduction which showed that a global policy can be violated even if each security token enforces the policy locally. Our ideal key-management functionality considers a distributed set of security tokens as a single trusted third party. It makes sure that every use of a key is compliant with the (global) policy. Therefore, if a set of well-designed security tokens with a sound local policy emulates the ideal key-management functionality, they can never reach a state where a key is used for an operation that is contrary to the global policy. This implies that, in general, the key should be kept secret from the user, as the user cannot be forced to comply with any policy. Thus, keys are only accessed via an interface that executes only the operations on the key permitted by the policy. The functionality associates some meta-data, an *attribute*, to each key. This attribute defines the key's role, and thus its uses. Existing industrial standards [81] and recent academic proposals [21, 58] are similar in this respect.

8.1.2 *Sharing Secrets*

A key created on one security token is *a priori* only available to users that have access to this token (since it is hidden from the user). Many cryptographic protocols require that the participants share some key, so, in order to be able to run a protocol between two users of different security tokens, we need to be able to "transfer" keys between devices without revealing them. There are several ways to do this, e. g., using semantically secure symmetric or asymmetric encryption, but we will opt for the simplest, key-wrapping (the encryption of one key by another). While it is possible to define key-management with a more conceptual view of "transferring keys" and allow the implementation to decide for an option, we think that since key-wrapping is relevant in practice (it is defined in RFC 3394), the choice for this option allows us to define the key-management in a more comprehensible way. We leave the definition of a notion more general in this regard for future work.

8.1.3 *Secure Setup*

The use of key-wrapping requires some initial shared secret values to be available before keys can be transferred. We model the setup in the following way: A subset of users, *Room*, is assumed to be in

a secure environment during a limited setup-phase. Afterwards, the only secure channel is between a user U_i, and his security token ST_i. The intruder can access all other channels, and corrupt any party at any time, as well as corrupt keys, i. e., learn the value of the key stored inside the security token. This models the real world situation where tokens can be initialised securely but then may be lost or subject to, e. g., side channel attacks once deployed in the field.

8.1.4 *Operations required*

These requirements give a set of operations that key-management demands: `new` (create keys), `wrap` and `unwrap` (our chosen method of transferring keys), `corrupt` (corruption of keys) and `share/finish_setup` (modelling a setup phase in a secure environment). For policies that allow a key's attribute to be changed, the command `attr_change` is provided. We argue that a reasonable definition of secure key-management has to provide at least those operations. Furthermore, the users need a way to access the keys stored in the security tokens, so there is a set of operations for each type of key. A signature key, for example, allows the operations *sign* and *verify*. This allows the following classification: The first group of operations defines *key-management*, the second *key-usage*. While key-management operations, for example `wrap`, might operate on two keys of possibly different types, key-usage operations are restricted to calling an operation on a single key and user-supplied data. This is coherent with global policies as mentioned above: The form "key A can be used for task X" expresses key-usage, the form "the security of key A depends on keys B and C" expresses a constraint on the key-management.

8.2 KEY-USAGE (KU) FUNCTIONALITIES

We now define an abstract notion of a functionality making use of a key which we call a ku functionality. This will allow us to define our ideal key-management functionality $\mathcal{F}_{\text{KM}}$ in a way that is general with respect to a large class of cryptographic functionalities. For every ku operation, $\mathcal{F}_{\text{KM}}$ calls the corresponding ku functionality, receives the response and outputs it to the user. We define $\mathcal{F}_{\text{KM}}$ for arbitrary ku operations, and consider a security token secure, with respect to the implemented ku functionalities, if it emulates the ideal functionality $\mathcal{F}_{\text{KM}}$ parametrized by those ku functionalities. This allows us to provide an implementation for secure key-management independent of which ku functionalities are used.

8.2.1 *Credentials*

Many existing functionalities, e. g., [23], bind the roles of the parties, e. g., signer and verifier, to a machine ID encoded in the session parameters. In implementations, however, the privilege to perform an operation is linked to the knowledge of a key rather than a machine ID. While for most applications this is not really a restriction, it is for *key*-management. The privilege to perform an operation of a ku functionality must be transferable as some piece of information, which however cannot be the actual key: A signature functionality, for example, that exposes its keys to the environment is not realizable, since the environment could then generate dishonest signatures itself. Our solution is to generate a key, but only send out a *credential*, which is a hard-to-guess pointer that refers to this key. We actually use the key generation algorithm to generate credentials. As opposed to the real world, where security tokens map handles to keys, and compute the results based on the keys, in the ideal world, $\mathcal{F}_{\text{KM}}$ maps handles to credentials, and uses those credentials to address ku functionalities, which compute the results. The implementation of a ku functionality maps credentials to cryptographic keys (see Definition 24). While credentials are part of the key-management functionality $\mathcal{F}_{\text{KM}}$ and the ku functionality, they are merely devices used for abstracting keys. They are used in the proofs, but disappear in the reference implementation presented in Section 8.4.

In summary, credentials serve as an abstraction of keys in the ideal world. Whoever knows the credential is allowed to sign. Keys not only allow a distinguished party, the owner of the key, to perform operation but also allow *delegation* of this capacity. An abstraction of 'keys' by 'credentials' is thus more powerful than abstraction of the 'owner of a key' by a 'fixed identity of party that is allowed to sign'. In our scenario, where keys are exported, it is necessary to abstract keys.

8.2.2 *Key-manageable functionalities*

Our approach imposes assumptions on the ku functionalities, as they need to be implementable in a key-manageable way.

Definition 24 (key-manageable implementation)*:* A key-manageable implementation $\hat{\mathrm{I}}$ is defined by *(i)* a set of commands Cmds that can be partitioned into private and public commands, as well as key-(and credential-)generation, i. e., $\mathcal{C} = \mathcal{C}^{priv} \uplus \mathcal{C}^{pub} \uplus \{\,\mathsf{new}\,\}$, and *(ii)* a set of PPT algorithms implementing those commands, $\{\,\mathit{impl}_{\mathrm{C}}\,\}_{\mathrm{C}\in\mathcal{C}}$, such that for the key-generation algorithm $\mathit{impl}_{\mathsf{new}}$ it holds that

- for all k, $\Pr\left[\,\mathrm{k}' = \mathrm{k} \mid (\mathrm{k}', \mathit{public}) \leftarrow \mathit{impl}_{\mathsf{new}}(1^{\eta})\right]$ is negligible in η, and,

- $\Pr\left[|k_1| \neq |k_2| \mid (k_1, p_1) \leftarrow \mathit{impl}_{\mathsf{new}}(1^\eta); (k_2, p_2) \leftarrow \mathit{impl}_{\mathsf{new}}(1^\eta)\right]$ is negligible in η.

$\hat{I}$ is a protocol in the sense of [51, §5], i. e., a run-time library that defines only one protocol name. The session parameter encodes a machine ID P. When called on this machine, the code below is executed. If called on any other machine no message is accepted. From now on in our code we follow the convention that the response to a query $(\text{Command}, \mathit{sid}, \ldots)$ is always of the form $(\text{Command}^\bullet, \mathit{sid}, \ldots)$, or $\perp$. The variable L holds a set of pairs and is initially empty.

```
new: accept <new> from parentId;
        (key, public) ← impl_new(1^η);
        (credential,_) ← impl_new(1^η);
        L ← L ∪ {(credential, key)};
        send <new•, credential, public> to parentId
command: accept <C, credential, m> from parentId;
        if (credential, key) ∈ L for some key
                send <C•, impl_C(key, m)> to parentId
public_command: accept <C, public, m> from parentId;
        send <C•, impl_C(public, m)> to parentId
corrupt: accept <corrupt,credential> from parentId;
        if (credential, key) ∈ L for some key
                send <corrupt•, key> to parentId
inject: accept <inject,k> from parentId;
        (c,_) ← impl_new(1^η)
        L ← L ∪ {(c, k)}; send <inject•,c> to parentId
```

The definition requires that each command C can be implemented by an algorithm impl_C. If C is private, impl_C takes the key as an argument. Otherwise it only takes public data (typically the public part of some key, and some user data) as arguments. In other words, an implementation $\hat{I}$ emulating $\mathcal{F}$ is, once a key is created, stateless with respect to queries concerning this key. The calls $\langle$`corrupt`$\rangle$ and $\langle$`inject`$\rangle$ are necessary for cases where the adversary learns a key, or is able to insert dishonestly generated key-material.

Definition 25 (key-manageable functionality)*:* A poly-time functionality $\mathcal{F}$ (to be precise, an ideal protocol [51, § 8.2]) is *key-manageable* iff it is poly-time, and there is a set of commands $\mathcal{C}$ and implementations, i. e., PPT algorithms $\mathtt{Impl}_\mathcal{F} = \{\mathit{impl}_C\}_{C \in \mathcal{C}}$, defining a key-manageable implementation $\hat{I}$ (also poly-time) which emulates $\mathcal{F}$.

8.3 POLICIES

Since all credentials on different security tokens in the network are abstracted to a central storage, $\mathcal{F}_{\text{KM}}$ can implement a global policy. Every credential in $\mathcal{F}_{\text{KM}}$ is associated with an attribute from a set of attributes A and to the ku functionality it belongs to (which we will

call its *type*). Keys that are used for key-wrapping are marked with the type `KW`.

Definition 26 (Policy)*:* Given the ku functionalities $\mathcal{F}_i$, $i \in \{1, \ldots, l\}$ and corresponding sets of commands $\mathcal{C}_i$, a *policy* is a quaternary relation $\Pi \subset \{\mathcal{F}_1, \ldots, \mathcal{F}_l, \texttt{KW}\} \times \cup_{i \in \{1,\ldots,l\}} \mathcal{C}_i^{priv} \cup \{\texttt{new}, \texttt{wrap}, \texttt{unwrap}, \texttt{attr\_change}\} \times A \times A$.

$\mathcal{F}_{KM}$ is parametrized by a policy Π. If $(\mathcal{F}, C, a, a') \in \Pi$ and if

- $C = \texttt{new}$, then $\mathcal{F}_{KM}$ allows the creation of a new key for the functionality $\mathcal{F}$ with attribute a.
- $\mathcal{F} = \mathcal{F}_i$ and $C \in \mathcal{C}_i^{priv}$, then $\mathcal{F}_{KM}$ allows sending the command C to $\mathcal{F}$, if the key is of type $\mathcal{F}$ and has the attribute a.
- $\mathcal{F} = \texttt{KW}$ and $C = \texttt{wrap}$, then $\mathcal{F}_{KM}$ allows wrapping a key with attribute a' using a wrapping key with attribute a.
- $\mathcal{F} = \texttt{KW}$ and $C = \texttt{unwrap}$, then $\mathcal{F}_{KM}$ allows unwrapping a wrapping with attribute a' using a wrapping key with attribute a.
- if $C = \texttt{attr\_change}$, then $\mathcal{F}_{KM}$ allows changing a key's attribute from a to a'.

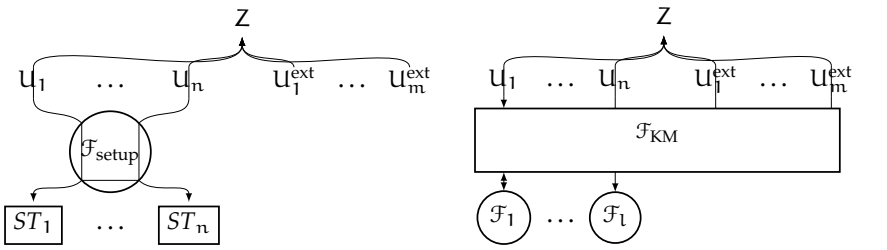
Note that a' is only relevant for the commands `wrap`, `unwrap` and `attr_change`. Because of the command `attr_change`, a key can have different attributes set for different users of $\mathcal{F}_{KM}$, corresponding to different security tokens in the real word.

Example 20*:* To illustrate Definition 26, consider the case of a single ku functionality for encryption $\mathcal{F}_{enc}$. The set of attributes A is $\{0, 1\}$: Intuitively a key with attribute 1 can be used for wrapping and a key with attribute 0 for encryption. The following table describes a policy that allows wrapping keys to wrap encryption keys, but not other wrapping keys, and allows encryption keys to perform encryption on user-data, but nothing else – even decryption is disallowed. The policy Π consists of the following 4-tuples ($\mathcal{F}$,Cmd,attr$_1$,attr$_2$):

$\mathcal{F}$	Cmd	attr$_1$	attr$_2$
KW	new	1	*
$\mathcal{F}_{enc}$	new	0	*
KW	wrap	1	0
KW	unwrap	1	0
$\mathcal{F}_{enc}$	enc	0	*

8.4 THE KEY-MANAGEMENT FUNCTIONALITY AND A GENERIC AND SECURE REFERENCE IMPLEMENTATION

We are now in a position to give a full definition of $\mathcal{F}_{KM}$, together with a generic, secure architecture for security APIs. We give the de-



(a) Distributed security tokens in the network

(b) An idealized functionality $\mathcal{F}_{KM}$ in the same network

Figure 15: Distributed security tokens in the network (left-hand side) and idealized functionality $\mathcal{F}_{KM}$ in the same network (right-hand side).

scription of $\mathcal{F}_{KM}$ in the Listings 3 to 8. At the same time, to illustrate our definition and demonstrate its use, we present the implementation of a security API showing that it is possible to implement a security API for key-management that is independent of the ku functions it provides. By presenting the functionality $\mathcal{F}_{KM}$ and the model implementation side-by-side we hope to be able to show how, on the one hand, how $\mathcal{F}_{KM}$ provides the security guarantees and deals with corner cases, while on the other hand, the implementation illustrates a straight-forward way of implementing $\mathcal{F}_{KM}$ using a deterministic, symmetric key-wrapping scheme.

The implementation *ST* is inspired by Kremer et al.'s proposal for a secure implementation of key-management [58] and is parametric with respect to the ku parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and the implementation functions $\overline{\texttt{Impl}} := \{\texttt{Impl}_{\mathcal{F}}\}_{\mathcal{F}\in\overline{\mathcal{F}}}$. It is therefore composable in the following sense: If a device performs the key-management according to our implementation, it does not matter how many, and which functionalities it enables access to, as long as those functionalities provide the amount of security the designer aims to achieve (cf. Corollary 1). In Chapter 10, we show that this implementation is indeed a realization of $\mathcal{F}_{KM}$. We emphasize that extending $\mathcal{F}_{KM}$ and the implementation by a new ku functionality does not require a new proof.

8.4.1 *Structure and Network setup*

For book-keeping purposes $\mathcal{F}_{KM}$ maintains a set $\mathcal{K}_{cor}$ of corrupted keys and a wrapping graph $\mathcal{W}$ whose vertices are the credentials. An edge (c_1, c_2) is created whenever (the key corresponding to) c_1 is used to wrap (the key corresponding to) c_2.

$\mathcal{F}_{KM}$ acts as a proxy service to the ku functionalities. It is possible to create keys, which means that $\mathcal{F}_{KM}$ asks the ku functionality for the credentials and stores them, but outputs only a *handle* referring to the credential, which represents the key. A handle can be the position of

the key in memory, or a running number – we just assume that there is a way to draw them such that they are unique. When a command $C \in \mathcal{C}_i^{priv}$ is called with a handle and a message, $\mathcal{F}_{\text{KM}}$ substitutes the handle with the associated credential, and forwards the output to $\mathcal{F}_i$. The response from $\mathcal{F}_i$ is forwarded unaltered. All queries are checked against the policy. The environment may corrupt parties connected to security tokens, as well as individual keys.

Definition 27 (Parameters to a security token network)*:* We summarize the parameters of a security token network as two tuples, $(\mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room})$ and $(\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi)$. The first tuple defines *network parameters*:

- $\mathcal{U} = \{U_1, \ldots, U_n\}$ are the party IDs of the users connected to a security token
- $\mathcal{U}^{\text{ext}} = \{U_1^{\text{ext}}, \ldots, U_m^{\text{ext}}\}$ are the party IDs of external users, i. e., users that do not have access to a security token.
- $\mathcal{ST} = \{ST_1, \ldots, ST_n\}$ are the party IDs of the security tokens accessed by $U_1, \ldots, U_n$.
- $\mathit{Room} \subset \mathcal{U}$.

The second tuple defines *key-usage parameters*:

- $\overline{\mathcal{F}} = \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$, and
- $\overline{\mathcal{C}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_l\}$ are key-manageable functionalities with corresponding sets of commands. Note that $\texttt{KW} \notin \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$, and that each $\mathcal{C}_i \in \overline{\mathcal{C}}$ is partitioned into the private $\mathcal{C}_i^{priv}$ and public commands $\mathcal{C}_i^{pub}$, as well as the singleton set consisting of `new`.
- Π is a policy for $\overline{\mathcal{F}}$ and $\overline{\mathcal{C}}$ (cf. Definition 26) and a membership test on Π can be performed efficiently.

Figure 15 shows the network of distributed users and security tokens on the left, and their abstraction $\mathcal{F}_{\text{KM}}$ on the right. There are two kinds of users: $U_1, \ldots, U_n =: \mathcal{U}$, each of whom has access to exactly one security token ST_i, and external users $U_1^{\text{ext}}, \ldots, U_m^{\text{ext}} =: \mathcal{U}^{\text{ext}}$, who cannot access any security token but may access the public parts of keys that are stored on a security token. The security token ST_i can only be controlled via the user U_i. The functionality $\mathcal{F}_{\text{setup}}$ in the real world captures our setup assumptions, which need to be achieved using physical means. Among other things, $\mathcal{F}_{\text{setup}}$ assures a secure channel between each pair (U_i, ST_i). The necessity of this channel follows from the fact that *a*) GNUC forbids direct communication between two regular protocol machines (indirect communication via A is used to model an insecure channel) and *b*) $U_1, \ldots, U_n$ can be corrupted by the environment, while $ST_1, \ldots, ST_n$ are incorruptible, since security tokens are designed to be better protected against physical attacks, as well as worms, viruses etc. Although we assume that

the attacker cannot gain full control of the device (party corruption), he might obtain or inject keys in our model (key corruption).

The session ID *sid* is of the form $\langle \alpha_1, \ldots, \alpha_{k-1}, \langle \texttt{prot} - \texttt{fkm}, sp \rangle \rangle$, where the session parameter *sp* is some encoding of the network parameters $\mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room}$. The code itself is parametric in the ku parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$. When we refer to $\mathcal{F}_{\text{KM}}$ as a network identity, we mean the machine ID $\langle \texttt{ideal}, \mathit{sid} \rangle$.

Similarly, each security token $ST_i \in \{ST_1, \ldots, ST_n\}$ is addressed via the machine ID $\langle ST_i, \mathit{sid} \rangle$. We will abuse notation by identifying the machine ID with ST_i, whenever the session ID is clear from the context. The session parameter within *sid* encodes the network parameters $\mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room}$. ST_i makes subroutine calls to the functionality $\mathcal{F}_{\text{setup}}$ which subsumes our setup assumptions. $\mathcal{F}_{\text{setup}}$ provides two things: 1. a secure channel between each pair U_i and ST_i, 2. a secure channel between all pairs ST_i and ST_j for which $U_i, U_j \in \mathit{Room}$ during the *setup phase* (see below). ST_i receives commands from a machine $U_i \in \mathcal{U}$, which is defined in Definition 29. U_i relays arbitrary commands sent by the environment to ST_i (via $\mathcal{F}_{\text{setup}}$). The environment cannot talk directly to ST_i, but the attacker can send queries on behalf of any corrupted user, given that the user has been corrupted previously (by the environment).

8.4.2 *Setup phase*

The setup assumptions are implemented by the functionality $\mathcal{F}_{\text{setup}}$, which Listing C.3 in Section C.3 in the appendix describes in full detail. All users in *Room* are allowed to share keys during the setup phase, i. e., the implementation is allowed to use secure channels to transport keys during this phase, but not later. This secure channel *between each two security tokens* $ST_i, ST_j \in \mathit{Room}$ is only used during the setup phase. Once the setup phase is finished, the expression `setup_finished` evaluates to true and the functionality enters the run phase. During the run phase, $\mathcal{F}_{\text{setup}}$ provides only a secure channel between a user U_i, which takes commands from the environment, and his security token ST_i. When we say that ST_i calls $\mathcal{F}_{\text{setup}}$, we mean that it sends a message to the machine ID $\langle \texttt{ideal}, \langle \mathit{sid}, \langle \texttt{prot} - \texttt{fsetup}, \langle \mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room} \rangle \rangle \rangle \rangle$.

8.4.3 *Executing commands in* $\mathcal{C}^{priv}$

We will now describe $\mathcal{F}_{\text{KM}}$ and the reference implementation *ST* command by command. Note that, from now on, when we say that $\mathcal{F}_{\text{KM}}$ *calls* $\mathcal{F}$, we mean that it sends a message to a regular peer that calls $\mathcal{F}$ as a sub-protocol and relays the answers. Formally, $\mathcal{F}_{\text{KM}}$ sends a message to the machine ID $\mathsf{F} = \langle \langle \texttt{reg}, \mathcal{F} \rangle, \mathit{sid} \rangle$, who in turn addresses $\langle \langle \texttt{reg}, \mathcal{F} \rangle, \langle \mathit{sid}, \langle \mathcal{F}, \mathsf{F} \rangle \rangle \rangle$ as a dummy party. This is necessary

since Condition C6 in [51, §4.5] disallows ideal parties from making sub-routine calls. Note first that, for unambiguity, we use the code $\mathcal{F}$ as the party ID for this user. Note secondly that $\mathcal{F}$ uses the session parameter F to identify F as the only machine ID it accepts messages from.

If the policy Π permits execution of a command $C \in \mathcal{C}^{priv}$, $\mathcal{F}_{KM}$ calls the corresponding functionality as a sub-protocol (via F), substituting the handle by the corresponding credential. Similarly, ST_i uses the corresponding key to compute the output of the implementation function $impl_C$ of the command C (see Listings 2). Note that the security token communicates with its respective user via $\mathcal{F}_{setup}$, which forwards messages between ST_i and U_i, serving as a secret channel.

Listing 2: Executing command C on a handle h with data m ($\mathcal{F}_{KM}$ above, ST_i below).

```
command[finish_setup]: accept <C ∈ C_i^priv, h, m> from U ∈ 𝒰;
if Store[U, h]=<F_i, a, c> and <F_i, C, a, *>∈ Π and F_i ≠ KW
        call F_i with <C, c, m>; accept <C•, r> from F_i;
        send <C•, r> to U
```

```
command[finish_setup]:  accept <C ∈ C_i'^priv, h, m> from F_setup;
if Store[U_i, h] =<F_i', a, k> and <F_i', C, a, *>∈ Π and F_i' ≠ KW
        send <C•, impl_C(k, m)> to F_setup
```

8.4.4 *Creating keys*

A user can create keys of type $\mathcal{F}$ and attribute a using the command $\langle \mathsf{new}, \mathcal{F}, a \rangle$. In $\mathcal{F}_{KM}$, the functionality $\mathcal{F}$ is asked for a new credential and some public information. The credential is stored with the meta-data at a freshly chosen position h in the store. *ST* proceeds similarly, but stores an actual key, instead of a credential. Both $\mathcal{F}_{KM}$ and *ST* output the handle h and the public information given by $\mathcal{F}$, or produced by the key-generation algorithm. $\mathcal{F}_{KM}$ treats wrapping keys differently: It calls the key-generation function for KW. It is possible to change the attributes of a key in future, if the policy permits (Listing 6).

Listing 3: Creating keys of type $\mathcal{F}$, and attribute a ($\mathcal{F}_{KM}$ above, ST_i below).

```
new[ready]: accept <new, F, a> from U ∈ 𝒰;
        if ⟨F, new, a, *⟩ ∈ Π
                if F =KW then
                        (c, public) ← impl_new^KW(1^η)
                else
                        call F with <new>;
                        accept <new•, c, public> from F
                if c ∈ 𝒦 ∪ 𝒦_cor then
                        send <error> to A
```

```
                else
                        create h;
                        Store[U, h] ← <F,a,c>;
                        K := K ∪ {c}; send <new•,h,public> to U
```

```
new[ready]: accept <new,F,a> from F_setup;
        if ⟨F, new, a, ∗⟩ ∈ Π
                (k, public) ← impl^F_new(1^η);
                create h;
                Store[U_i, h] ← <F,a,k>;
                send <new•,h,public> to F_setup
```

8.4.5 *Wrapping and Unwrapping*

The commands that are important for key-management are handled by $\mathcal{F}_{\text{KM}}$ itself. To transfer a key from one security token to another in the real world, the environment instructs, for instance, U_1 to ask for a key to be *wrapped* (see Listing 4). A wrapping of a key is the encryption of a key with another key, the wrapping key. The wrapping key must of course be on both security tokens prior to that. U_1 will receive the wrap from ST_1 and forward it to the environment, which in turn instructs U_2 to unwrap the data it just received from U_1. The implementation ST_i just verifies if the wrapping confirms the policy, and then produces a wrapping of c_2 under c_1, with additionally authenticated information: the type and the attribute of the key, plus a user-chosen identifier that is bound to a wrapping in order to identify which key was wrapped. This could, e. g., be a key-digest provided by the ku functionality the key belongs to. The definition of $\mathcal{F}_{\text{KM}}$ is parametric in the algorithms wrap, unwrap and $impl_{new}$ used to produce the wrapping. When a handle to a credential c is corrupted, the variable $key[c]$ stores the corresponding key, c.f. Listing 7. We use $\$^l$ to denote a bitstring drawn from a uniform distribution of bitstrings of length l .

Listing 4: Wrapping key h_2 under key h_1 with additional information *id* ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

```
wrap[finish_setup]: accept <wrap,h_1,h_2,id> from U ∈ 𝒰;
if Store[U, h_1]=<KW,a_1,c_1> and Store[U, h_2]=<F_2,a_2,c_2>
and <KW,wrap,a_1,a_2>∈ Π then
        if ∃w.<c_2,<F_2,a_2,id>,w>∈encs[c_1]
                send <wrap•,w> to U
        else
                W ← W ∪{(c_1, c_2)};
                if c_1 ∈ K_cor
                        forall c_3 reachable from c_2 in W;
                                corrupt c_3;
                                w ← wrap^<F_2,a_2,id>(c_1, key[c_2])
                else
```

```
                    w ← wrap^<F2,a2,id>(c1,$^|c2|)
                encs[c1] ← encs[c1] ∪{ <c2,<F2,a2,id>,w>};
                send <wrap•,w> to U
```

```
wrap[finish_setup]: accept <wrap,h1,h2,id> from F_setup;
        if Store[Ui,h1]=<KW,a1,k1> and Store[Ui,h2]=<F2,a2,k2>
                    and <KW,wrap,a1,a2>∈ Π
                w ← wrap^<F2,a2,id>(k1,k2);
                send <wrap•,w> to F_setup
```

When a wrapped key is unwrapped using an uncorrupted key, $\mathcal{F}_{\text{KM}}$ checks if the wrapping was produced before using the same identifier. Furthermore, $\mathcal{F}_{\text{KM}}$ checks if the given attribute and types are correct. If this is the case, it creates another entry in `Store`, i. e., a new handle h' for the user U pointing to the *correct* credentials, type and attribute type of the key. This way, $\mathcal{F}_{\text{KM}}$ can guarantee the consistency of its database for uncorrupted keys, see Theorem 2 in Chapter 9. If the key used to unwrap is corrupted, this guarantee cannot be given, but the resulting entry in the store is marked corrupted. It is possible to inject keys by unwrapping a wrapping created *outside the device*. Such keys could be generated dishonestly by the adversary, that is, not using their respective key-generation function. In this case, the ⟨ `inject` ⟩ call imports the cryptographic value of the key onto the ku functionality, which generates a new credential for this value.

Listing 5: Unwrapping w created with attribute a_2, F_2 and *id* using the key h_1. $\exists!x.p(x)$ holds if there exists exactly one x such that $p(x)$ holds ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

```
unwrap[finish_setup]: accept <unwrap,h1,w,a2,F2,id> from U ∈ 𝒰;
if Store[U,h1]=<KW,a1,c1> and <KW,unwrap,a1,a2>∈ Π,F2 ∈ F̄
        if c1 ∈ K_cor
                c2 ← unwrap^<F2,a2,id>(c1,w);
                if c2 ≠ ⊥ and c2 ∉ K
                        if F2 =KW
                            create h2;
                            Store[U,h2] ← <F2,a2,c2>;
                            key[c2] ← c2;
                            K_cor ← K_cor ∪{c2}
                        else
                            call F2 with <inject,c2>;
                            accept <inject•,c'>;
                            if c' ∉ K ∪ K_cor
                                        create h2;
                                        Store[U,h2] ← <F2,a2,c'>;
                                        key[c'] ← c2;
                                        K_cor ← K_cor ∪{c'};
                                        send <unwrap•,h> to U
                else if c2 ≠ ⊥ ∧ c2 ∈ K ∧ c2 ∈ K_cor
                    create h2;
                    Store[U,h2] ← <F2,a2,c2>;
                    send <unwrap•,h> to U
```

```
            else // (c2 = ⊥ ∨ c2 ∈ K \ Kcor)
                send <error> to A
        else if ( c1 ∉ Kcor and ∃!c2.<c2,<F2,a2,id>,w>∈encs[c1
            ])
                create h2;
                Store[U, h2] ← <F2,a2,c2>;
                send <unwrap•,h2> to U
```

```
unwrap[finish_setup]: accept <unwrap,h1,w,a2,F2,id> from Fsetup
if Store[Ui, h1]=<KW,a1,k1> and F2 ∈ F̄ and <KW,unwrap,a1,a2>∈ Π
                and k2 = unwrap^<F2,a2,id>(k1, w) ≠ ⊥
        create h2;
        Store[U, h2] ← <F2,a2,k2>;
        send <unwrap•,h2> to Fsetup
```

There is an improvement that became apparent during the emulation proof (see Chapter 10). When unwrapping with a corrupted key, $\mathcal{F}_{\text{KM}}$ checks the attribute to be assigned to the (imported) key against the policy, instead of accepting that a corrupted wrapping-key might import any wrapping the attacker generated. This prevents, e.g., a corrupted wrapping-key of low security from *creating* a high-security wrapping-key by unwrapping a dishonestly produced wrapping. This detail in the definition of $\mathcal{F}_{\text{KM}}$ enforces a stronger implementation than the one in [58]: *ST* validates the attribute given with a wrapping, enforcing that it is sound according to the policy, instead of blindly trusting the authenticity of the wrapping mechanism. Hence our implementation is more robust against key-corruption.

8.4.6 *Changing attributes of keys*

The attributes associated with a key with handle h can be updated using the command $\langle \texttt{attr\_change}, h, a' \rangle$.

Listing 6: Changing the attribute of h to a' ($\mathcal{F}_{\text{KM}}$ above, ST_i below).

```
attr_change[finish_setup]: accept <attr_change,h,a'> from U ∈ U;
if Store[U, h]=<F,a,c> and<F,attr_change,a,a'>∈ Π
        Store[U, h]=<F,a',c>;
        send <attr_change•> to U
```

```
attr_change[finish_setup]: accept <attr_change,h,a'> from Fsetup;
if Store[Ui, h]=<F,a,k> and <F,attr_change,a,a'>∈ Π
        Store[Ui, h]=<F,a',k>;
        send <attr_change•> to Fsetup
```

8.4.7 *Corruption*

Since keys might be used to wrap other keys, we would like to know how the loss of a key to the adversary affects the security of other

keys. When an environment "corrupts a key" in $\mathcal{F}_{KM}$, the adversary learns the credentials to access the functionalities. Since corruption can occur indirectly, via the wrapping command, too, we factored this out into a procedure used both in the corrupt command and the wrap command. The procedure used in $\mathcal{F}_{KM}$ is depicted in Listing 7. Listing 8 shows the actual corruption command in $\mathcal{F}_{KM}$ and in ST. In $\mathcal{F}_{KM}$, the corruption procedure is invoked for all keys that have been wrapped with the newly corrupted key. ST implements this corruption by outputting the actual key to the adversary.

Listing 7: Corruption procedure used in steps `corrupt` and `wrap`.

```
procedure for corrupting a credential c:
─────────────────────────────────────────
K_cor ← K_cor ∪ {c}
for any Store[U, h] =< F, a, c >
        if F = KW
                key[c] ← c; send <corrupt•,h,c> to A
        else
                call F with <corrupt,c>; accept <corrupt•,k> from
                     F
                key[c] ← k; send <corrupt•,h,k> to A
```

Listing 8: Corrupting h ($\mathcal{F}_{KM}$ above, ST_i below).

```
corrupt[finish_setup]:  accept <corrupt,h> from U ∈ 𝒰;
        if Store[U, h] =< F, a, c >
                for all c′ reachable from c in 𝒲 corrupt c′
```

```
corrupt[finish_setup]: accept <corrupt,h> from F_setup;
        if Store[U_i, h] =< F, a, k > send <corrupt•,h,k> to A
```

8.4.8 *Public key operations*

Some cryptographic operations (e. g., digital signatures) allow users without access to a security token to perform certain operations (e. g., signature verification). Those commands do not require knowledge of the credential (in $\mathcal{F}_{KM}$), or the secret part of the key (in ST). They can be computed using publicly available information. In the case where participants in a high-level protocol make use of, e. g., signature verification, but nothing else, the protocol can be implemented without requiring those parties to have their own security tokens. Note that $\mathcal{F}_{KM}$ relays this call to the underlying ku functionality unaltered, and independent of its store and policy (see Figure 9). The implementation ST_i does not implement this step, since U_i, U_i^{ext} compute $impl_C(public, m)$ themselves.

Listing 9: Computing the public commands C using the inputs *public* and m ($\mathcal{F}_{KM}$, note that ST_i does not implement this step).

```
public_command: accept <C,public,m> from U ∈ U ∪ U^ext;
        if C ∈ C_{i,pub}
                call F_i with <C,public,m>;
                accept <C•,r> from F_i;
                send <C•,r> to U
```

8.4.9 *Formal definition of* $\mathcal{F}_{\text{KM}}$

Before we give the formal definition of $\mathcal{F}_{\text{KM}}$, note that $\mathcal{F}_{\text{KM}}$ is not an ideal protocol in the sense of [51, § 8.2], since not every regular protocol machine runs the dummy party protocol – the party $\langle\langle \texttt{reg}, \mathcal{F}_i \rangle, sid \rangle$ relays the communication with the ku functionalities.

Definition 28 ($\mathcal{F}_{\text{KM}}$)*:* Given the ku parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, and polytime algorithms `wrap`, `unwrap` and $impl_{new}$, let the ideal protocols $\mathcal{F}_{p+1}, \ldots, \mathcal{F}_l$ be rooted at `prot-`$\mathcal{F}_{p+1}$,...,`prot-`$\mathcal{F}_l$. In addition to those protocols names, $\mathcal{F}_{\text{KM}}$ defines the protocol name `prot-fkm`. For `prot-fkm`, the protocol defines the following behaviour: a regular protocol machine with machine ID $\langle\langle \texttt{reg}, \mathcal{F}_i \rangle, sid \rangle$ for $\mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$ runs the following code:

```
ready: accept <ready> from parentId
        send <ready> to <ideal,sid> (= F_KM)
relay_to: accept <m> from <ideal,sid> (= F_KM)
        send <m> to <<reg,F_i>,<sid,<prot-F_i,<>>> (= F_i)
relay_from: accept <m> from <<reg,F_i>,<sid,<prot-F_i,<>>>
        send <m> to <ideal,sid> (= F_KM)
```

The ideal party runs the logic for $\mathcal{F}_{\text{KM}}$ *described in Listings 3 to 8.*

REMARK 1: Credentials for different ku functionalities are distinct. It is nonetheless possible to encrypt and decrypt arbitrary credentials using `<wrap>` and `<unwrap>`. Suppose a designer wants to prove a security API secure which uses shared keys for different operations. One way or another, she would need to prove that those roles do not interfere. For this case, we suggest providing a functionality that combines the two desired operations, and proving that the implementation of the two operations combined emulates the combined functionality. It is possible to assign different attributes to keys of the same ku functionality, and thus restrict their use to certain commands, effectively providing different roles for credentials to the same ku functionality. This can be done by specifying two attributes for the two roles and defining a policy that restricts which operation is permitted for a key of each attribute.

REMARK 2: Many commonly used functionalities are not *caller-independent*, often the access to critical functions is restricted to a network party that is encoded in the session identifier. In the imple-

mentation, this party is the party holding the key which is otherwise not represented in the functionality. We think that it is possible to construct caller-independent functionalities for many functionalities, if the implementation relies on keys but is otherwise stateless. A general technique for transforming such functionalities into key-manageable functionalities that preserves existing proofs will be discussed in Section 12.2 in the conclusion of this thesis.

REMARK 3: Constraint C6 in [51, §8.2] requires each regular machine to send a message to $\mathcal{F}_{\text{setup}}$ before it can address it. The initialization procedure and the parts of the definition of $\mathcal{F}_{\text{KM}}$, *ST* and $\mathcal{F}_{\text{setup}}$ that perform this procedure are explained in detail in Appendix C.1.

8.4.10 *Formal definition of the security token network*

We can now define the network of security tokens *ST*, users and external users that implements $\mathcal{F}_{\text{KM}}$:

Definition 29 (security token network)*:* For ku parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and implementation functions $\overline{\texttt{Impl}} := \{\,\texttt{Impl}_F\,\}_{F \in \overline{\mathcal{F}}}$, define the protocol $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ as follows: $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ defines $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot}-\texttt{fkm})$ and $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot}-\texttt{fsetup})$. The session parameter is expected to be an encoding of the network parameters $\mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room}$. The code executed depends on the party running $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}(\texttt{prot}-\texttt{fkm})$: If the party has identity U_i, (which we assume to have the form $\langle\langle \texttt{reg}, \texttt{u-i}\rangle, \mathit{sid}\rangle$) the following code is executed:

```
relay_to: accept <m> from parentId
        call F_setup with <m, ST_i>
relay_from: accept <m, ST_i> or <m = ⊥> from F_setup
        send <m> to parentId
public_command:
        accept <C, public, m> from parentId
        if C ∈ C_{i,pub}
                send <C•, impl_C(public, m)> to parentId
```

If the party has identity U_i^{ext}, (which we assume to have the form $\langle\langle \texttt{reg}, \texttt{ext-u-i}\rangle, \mathit{sid}\rangle$) the following code is executed:

```
public_command: accept <C, public, m> from parentId
        if C ∈ C_{i,pub}
                send <C•, impl_C(public, m)> to parentId
```

A regular protocol machine with machine ID $\texttt{<reg,}\mathcal{F}_i\texttt{>,sid}$ for $\mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$ runs the following code:

```
ready: accept <ready> from parentId
        call F_setup with <ready>
```

All other regular protocol machines run the code of the dummy adversary. If the party has identity ST_i, (which we assume to have the

form $\langle\langle \mathtt{reg}, \mathtt{st\text{-}i}\rangle, \mathit{sid}\rangle$) then the code for ST described in this chapter is executed for i. The code for $ST_{\mathtt{i}}$ is given in Section 8.4 and in, for the setup procedure, in Appendix C.1. For other machines, including ideal machines, it responds to any message with an error message to the adversary, i. e., $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}(\mathtt{prot\text{-}fkm})$ is *totally regular*. $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}(\mathtt{prot{-}fkm})$ declares the use of `prot-fsetup` as a subroutine. $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}(\mathtt{prot{-}fsetup})$ runs $\mathcal{F}_{\text{setup}}$, i. e., $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ is a $\mathcal{F}_{\text{setup}}$-hybrid protocol.

9

ANALYSIS OF THE KEY-MANAGEMENT FUNCTIONALITY

In this chapter, we will give an analysis of $\mathcal{F}_{\text{KM}}$ in three steps: First, we will show that this functionality gives the properties that you would expect it to guarantee. Second, we will discuss the limitations of our approach. Third, we will discuss to what extent we have reached the requirements for a commonly acceptable definition we have outlined in Chapter 6.

9.1 PROPERTIES

In order to identify some properties we get from the design of $\mathcal{F}_{\text{KM}}$, we introduce the notion of an attribute policy graph:

Definition 30: We define a family of *attribute policy graphs* $(\mathcal{A}_{\Pi,\mathcal{F}})$, one for each ku functionality $\mathcal{F}$ and one for key-wrapping (in which case $\mathcal{F} = \texttt{KW}$) as follows:

- a is a node in $\mathcal{A}_{\Pi,\mathcal{F}}$ if $(\mathcal{F}, C, a, a') \in \Pi$ for some C, a'.
- a is additionally marked `new` if $(\mathcal{F}, \texttt{new}, a, a') \in \Pi$.
- An edge (a, a') is in $\mathcal{A}_{\Pi,\mathcal{F}}$ whenever $(\mathcal{F}, \texttt{attr\_change}, a, a') \in \Pi$.

Example 21: For the policy Π described in Example 20, the attribute policy graph $\mathcal{A}_{\Pi,\texttt{KW}}$ contains one node, labelled 1, connected to itself and marked `new`. Similarly, the attribute policy graph $\mathcal{A}_{\Pi,\mathcal{F}_{\text{enc}}}$ contains one node 0 connected to itself and marked `new`.

The following theorem shows that *(1)* the set of attributes an uncorrupted key can have in $\mathcal{F}_{\text{KM}}$ is determined by the attribute policy graph, *(2)* there are exactly three ways to corrupt a key, and *(3)* ku functionalities receive the `corrupt` message only if a key is corrupted.

Theorem 2 (Properties of $\mathcal{F}_{\text{KM}}$)*:* Every instance of $\mathcal{F}_{\text{KM}}$ with parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and session parameters $\mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}$, *Room* has the following properties:

(1) At any step of an execution of $[\mathcal{F}_{\text{KM}}, A_D, Z]$, the following holds for $\mathcal{F}_{\text{KM}}$: For all $\texttt{Store}[U, h] = \langle \mathcal{F}, a, c \rangle$ such that $c \notin \mathcal{K}_{\text{cor}}$, there is a node a' marked `new` in the attribute policy graph $\mathcal{A}_{\Pi,\mathcal{F}}$ such that a is reachable from a' in $\mathcal{A}_{\Pi,\mathcal{F}}$ and there was a step `new` where $\texttt{Store}[U', h'] = \langle \mathcal{F}, a', c \rangle$ was added.

(2) At any step of an execution of $[\mathcal{F}_{\text{KM}}, A_D, Z]$, the following holds for $\mathcal{F}_{\text{KM}}$: All $c \in \mathcal{K}_{\text{cor}}$ were either

a) *directly corrupted:* there was a `corrupt` step triggered by a query $\langle \texttt{corrupt}, h \rangle$ from U while $\texttt{Store}[U, h] = \langle \mathcal{F}, a, c \rangle$, or indirectly, that is,

b) *corrupted via wrapping:* there is $c' \in \mathcal{K}_{\text{cor}}$ such that at some point the `wrap` step was triggered by a message $\langle \texttt{wrap}, h', h, id \rangle$ from U while $\texttt{Store}[U, h'] = \langle \texttt{KW}, a', c' \rangle$, $\texttt{Store}[U, h] = \langle \mathcal{F}, a, c \rangle$, or

c) *corrupted via unwrapping (injected):* there is $c' \in \mathcal{K}_{\text{cor}}$ such that at some point the `unwrap` step was triggered by a message $\langle \texttt{unwrap}, h', w, a, F, id \rangle$ from U while $\mathit{Store}[U, h'] = \langle \texttt{KW}, a', c' \rangle$ and $c = \texttt{unwrap}_{c'}^{\langle F, a, id \rangle}(w)$ for some a, F and id. If this step did not return `<error>`, then, at the point in time this step was triggered, $c \in \mathcal{K}_{\text{cor}}$ (it was corrupted before), or $c \notin \mathcal{K}$ (k was created outside $\mathcal{F}_{\text{KM}}$).

(3) At any step of an execution of $[\mathcal{F}_{\text{KM}}, A_D, Z]$, the following holds: Whenever an ideal machine $\mathcal{F}_i = \langle \texttt{ideal}, \langle \texttt{sid}, \langle \mathcal{F}_i, F \rangle \rangle \rangle$, with $F = \langle \langle \texttt{reg}, \mathcal{F} \rangle, \langle \texttt{sid} \rangle \rangle$ accepts the message $\langle \texttt{corrupt}, c \rangle$ for some c such that $\mathcal{F}_{\text{KM}}$ in session `sid` has an entry $\mathit{Store}[U, h] = \langle \mathcal{F}_i, a, c \rangle$, then $c \in \mathcal{K}_{\text{cor}}$ in $\mathcal{F}_{\text{KM}}$.

Proof. We proof each property separately:

(1) Proof by induction over the number of epochs since $\mathcal{F}_{\text{KM}}$'s first activation, t: If $t = 0$, `Store` is empty. $t \rangle 0$: Since the property was true in the previous step, there are only three steps we need to look at: If a key $\langle \mathcal{F}, a, c \rangle$ is added to `Store` at step `new`, then it is created only if the policy contains an entry $(\mathcal{F}, \texttt{new}, a)$, i. e., a itself is a `new` node. If a key $\langle \mathcal{F}, a, c \rangle$ is added to `Store` at step `unwrap`, let $\langle \texttt{unwrap}, h_1, w, \mathcal{F}, id \rangle$ be the arguments sent by a user U, and $\texttt{Store}[U, h_1] = \langle \texttt{KW}, a_w, c_w \rangle$. If $c \notin \mathcal{K}_{\text{cor}}$, then $c_w \notin \mathcal{K}_{\text{cor}}$, too, and thus there is an entry $\langle c, \langle \mathcal{F}, a, id \rangle, w \rangle \in \texttt{encs}[c_w]$. The array `encs` is only written in `wrap`, therefore there was a position $[U', h']$ in the store, such that $\texttt{Store}[U', h'] = \langle \mathcal{F}, a, c \rangle$. Using the induction hypothesis, we see that a is reachable in the attribute policy graph. The third and last step where the store is written to is `AttributeChange`. If this step alters the attribute from a' to a, there must have been an entry $(\mathcal{F}, \texttt{attr\_change}, a', a) \in \Pi$. By induction hypothesis, a' is reachable from a `new` node, and hence a is, too.

(2) A credential c is only added to the set $\mathcal{K}_{\text{cor}}$ in three steps: If it is added in `corrupt`, then a message $\langle \texttt{corrupt}, h \rangle$ was received from $U \in \mathcal{U}$ and $\texttt{Store}[U, h] = \langle F, a, c \rangle$. If it was added in `wrap`, then a message $\langle \texttt{wrap}, h_1, h_2 \rangle$ must have been received from $U \in \mathcal{U}$ while $\texttt{Store}[U, h_1] = \langle \texttt{KW}, a_1, c_1 \rangle$, and c was reachable from c_1. Let c' be the last node on the path to c. $c' \in \mathcal{K}_{\text{cor}}$ because

it is reachable from c_1, too. Since $(c', c) \in \mathcal{W}$, there was another wrapping query $\langle \texttt{wrap}, h', h, id \rangle$ with $\texttt{Store}[U, h'] = \langle \texttt{KW}, a', c' \rangle$ and $\texttt{Store}[U, h] = \langle \mathcal{F}, a, c \rangle$. Since entries in the store are never deleted (only the attribute can be altered), and credentials are never removed from $\mathcal{K}_{\text{cor}}$, the property holds in this case. If it was added in `unwrap`, then $c \notin \mathcal{K}$ at that point in time, and $c = \texttt{unwrap}_{c'}^{\langle F, a, id \rangle}(w) \neq \bot$. Furthermore, we observe that the conditionals prior to adding c to $\mathcal{K}_{\text{cor}}$ require that $\texttt{Store}[U, h'] = \langle \texttt{KW}, a', c' \rangle$, $c' \in \mathcal{K}_{\text{cor}}$, and that the step was triggered by a message $\langle \texttt{unwrap}, h', w, a, F, id \rangle$. If $c \in \mathcal{K} \setminus \mathcal{K}_{\text{cor}}$, then $\langle \texttt{error} \rangle$ is output, since c is only added to $\mathcal{K}_{\text{cor}}$ if $c \in \mathcal{K}$ (thus the first subbranch is not taken) and $c \notin \mathcal{K} \cup \mathcal{K}_{\text{cor}}$ (thus the first and second subbranch are not taken).

(3) $\hat{I}_i$ accepts only messages coming from the party F, and F in turn only accepts messages coming from $\mathcal{F}_{\text{KM}}$. Therefore, we can conclude from the definition of step `corrupt` in $\mathcal{F}_{\text{KM}}$ that $c \in \mathcal{K}_{\text{cor}}$.

□

9.2 LIMITATIONS

Before we discuss $\mathcal{F}_{\text{KM}}$ and to what extend it achieves our goals from Chapter 6, we discuss the limitations of the key-management functionality we have presented in this chapter.

ARCHITECTURE OF THE NETWORK One of the key features of the key-management functionality $\mathcal{F}_{\text{KM}}$ is that it is able to abstract a network of security APIs as one monolithic block that verifies accordance to a policy before every step it takes. This ensures that the policy is respected on a global level, as the first statement of Theorem 2 and the definition of $\mathcal{F}_{\text{KM}}$ show. Different instantiations of $\mathcal{F}_{\text{KM}}$ with different parameters, e. g., in terms of the policy and the ku functionalities supported, may coexist in the same network, but they may not share the same keys.

In practice, this means that the enforcement of a global policy can only extend to a set of security APIs in the network that are similarly configured, in particular, that offer the same interface to the outside. This is for example the case if each department in an organisation has an identical HSM that manages the department's keys, but also shares some common key used for backup and key-transfer. If one of the HSMs offers a command that the other HSMs do not support then it is (in general) not possible to abstract those HSMs using the same instance of $\mathcal{F}_{\text{KM}}$. Since the key-transfer in the setup phase is limited to the same instance of $\mathcal{F}_{\text{KM}}$, those HSMs cannot share keys with each other. One such example is the Yubikey protocol from Chapter 4:

The Yubikeys in the network and the YubiHSM provide a completely different interface, but need to share the AES keys in order to run the protocol.

In summary, our approach is limited to cases where the security APIs in a network offer the same interface, i. e., where the environment is homogeneous.

KEY-TRANSPORT Currently, $\mathcal{F}_{\text{KM}}$ is tightly coupled with the employment of a deterministic, symmetric and authenticated encryption scheme that is secure against key-dependant messages for key export and import. While practitioners indeed favour deterministic key-encryption in protocol design and standardization efforts (see, e. g., RFC 3394), it restricts the analysis to security devices providing this kind of encryption. We have not yet covered asymmetric encryption of keys in $\mathcal{F}_{\text{KM}}$ (but we cover asymmetric encryption of user-supplied data), although $\mathcal{F}_{\text{KM}}$ could be extended to support this option. An even better alternative than extending $\mathcal{F}_{\text{KM}}$ to support different ways of key-transfer would be to introduce a more conceptual view of transferring keys. A key-management functionality would support an interface that allows the environment to give the instruction to transfer a key from one security API to another, but send the wrapping of the key, or whatever piece of information is transferred, on a public channel, i. e., to the adversary, as opposed to the environment. The adversary is free to relay the message to the security API which is supposed to receive the message. In case he does, the environment only receives a simple acknowledgement from the user connected to the receiving token. The machines in $\mathcal{U}$ would implement this more abstract interface depending on what mechanism the actual security APIs employs. This way, we could formulate $\mathcal{F}_{\text{KM}}$ without specifying how exactly keys are transferred.

KEY-USAGE The access that key-usage functions have on keys is very limited: They can compute an algorithm that uses the value of the key, some untrusted input provided by the user or the adversary, and randomness. This is a realistic assumption for general purpose security APIs such as PKCS#11 but not for security APIs that are designed for a specific protocol, like the Yubikey. The Yubikey can be set up with an AES key. Upon a query, it computes the encryption of a OTP that contains a running counter. Due to the design of $\mathcal{F}_{\text{KM}}$, it is not possible store the last counter value, so $\mathcal{F}_{\text{KM}}$ is not useful for the analysis of the security of a set of Yubikeys in the network. It would only be possible to compute the encrypted OTP given the key and the counter value as an input, but the counter cannot be trusted unless it is stored on the device itself.

On the other hand, it is this restriction, the strict separation between key-management and key-usage, that allows us to achieve the

result in Chapter 10 which shows a generic implementation of the key-management secure with respect to arbitrary ku functionalities. It might be possible to achieve the same result with a redesign of $\mathcal{F}_{\text{KM}}$ that supports an additional store for payload values, similar to the store for credentials, which might be used in each ku functionality. In the case of the Yubikey, this would allow storing the last counter value on the device.

COMMITMENT PROBLEM Adaptive corruption of parties, as well as adaptive corruption of keys that produce an encryption, provokes the well-known commitment problem [50]. This requires us to place limitations on the types of corruptions that the environment may produce.

9.3 DISCUSSION

In the Chapter 6, we gave five qualities which we think a persuasive definition of security should possess. For each of those desiderata, we will now discuss whether is satisfied by the definition of provably secure key-management in the GNUC model presented in Chapter 8,

> It should set out the essential attributes of what it defines.

Our definition quite clearly defines the aspects of the use of a security API that are relevant for key-management. The characteristics of a security API are set out through the interface $\mathcal{F}_{\text{KM}}$ provides: First, that it provides an interface to cryptographic keys that are stored in some secure memory. Second, that this interface allows to indirectly use this keys to compute cryptographic functions possibly depending on one (but for the ku functions not more than one), randomness, and the user's input. Third, the access to those secrets can be restricted, using the policy.

The characteristics of the security of security APIs are set out as well: First, it does not reveal secrets to the user (unless a key is corrupted, in which case the adversary learns those secrets). Second, the cryptographic functions are secure, which is defined by the corresponding ku functionality. Third, the intended access restrictions to the secrets, defined in the policy Π are respected. Fourth, the previous properties hold in a network where the user might be malicious, and where several security APIs might be present and might contain the same secrets, through the use of the GNUC framework and the support for party corruption.

Finally, the concept of credentials sets out quite well which attributes are essential for a key, and how keys can be handled in simulation-based security frameworks.

> It should not be too wide, e. g., it should not include security APIs that are insecure.

Theorem 2 gives some indication that this is achieved, but careful inspection of $\mathcal{F}_{\text{KM}}$ is necessary to validate this claim.

It should not be too narrow.

As pointed out in the previous section, there are a number of limitations to our approach, some of which exclude certain kinds of APIs, in particular security APIs that focus less on key-management and more on providing some protocol-specific functionality. One can argue that security results for the latter kind of security APIs are likely to be protocol dependant as well.

The key-management functionality presented is not as general as our approach would allow; it could be extended to allow for different kinds of key-transportation, for different kinds of ku functionalities, etc., as we pointed out in the previous section. However, the approach of separating key-management and key-usage, combined with composability features of the underlying framework allow us to reason about the security of the key-management and key-usage, while keeping the definition flexible in this regard, which we regard as a promising starting point for future work.

It should not be obscure.

Definitions in simulation-based security frameworks, such as the one we employ in this work, GNUC, are notorious for being long and full of technical details. The definition of $\mathcal{F}_{\text{KM}}$ is no exception in this regard, it might even be worse than most functionalities. The fact that a theorem like Theorem 2 is necessary to convince the reader of the soundness of the definition illustrates the obscurity of $\mathcal{F}_{\text{KM}}$. In the following, we will try justify the obscurity of $\mathcal{F}_{\text{KM}}$, but we acknowledge the fact that it is indeed difficult to read and understand.

First, many functionalities in simulation-based security suffer from this problem, often due to the definition of efficiency in the framework. Most existing frameworks, perhaps with the exception of the abstract cryptography framework by Maurer and Renner [67], provide their composability results on very low level of abstraction, so that technical details, like the scheduling, the runtime of the protocol etc. are necessary to obtain a security result. The initialisation phase Section C.1, to give an example, is the result of such a technical requirement. Fortunately, since the first proposal by Canetti in 2001 [22], simulation-based security frameworks have been, and now, in 2013, still are an active subject of research [63], leading to continuous improvements [23, 51, 84, 67, 63]. We hope that future frameworks allow for presenting functionalities like $\mathcal{F}_{\text{KM}}$ in a more concise manner. In particular the initialisation procedure of the network (see Section C.1), as well as the communication between $\mathcal{F}_{\text{KM}}$ and its ku functionalities (see Definition 28) are obscured due to technical requirements of the framework.

Second, defining the security of key-management in this setting *is* a challenging task. It took year to obtain a satisfactory formulation of digital signature took years and repeated revision due to by subtle flaws making the functionality unrealizable [24, 23, 8]. $\mathcal{F}_{\text{KM}}$ preserves authenticity in a similar way to this signature functionality, but in a multi-session setting. So we must expect a key-management functionality to be at least as complex. Previous definitions of security for security APIs in terms of games are a matter of several pages, too [58, 21](but still more concise).

Third, as $\mathcal{F}_{\text{KM}}$ is the first formulation of a functionality of its kind, it is only natural that there is room for improvement. We hope that this work sparks interest for future work which might enhance the clarity of this work.

Security definitions should be applicable.

A definition in the UC framework can be applicable in two senses. First, it can be used to evaluate the security of a protocol. In Chapter 10 we show that $\mathcal{F}_{\text{KM}}$ can be used to show a generic implementation of key-management secure. Second, it can be used to help showing the security of higher-level protocols that make use,e. g., of one security tokens per participant. Using the composition theorem, the set of security tokens used by the participants can be substituted by an instance of $\mathcal{F}_{\text{KM}}$ which provides for example the properties shown in Theorem 2.

10

PROOF OF EMULATION.

We show that for arbitrary ku parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, the security token network described in Chapter 8, $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$, consisting of the set of users $\mathcal{U}$ connected to security tokens $\mathcal{ST}$, the set of external users $\mathcal{U}^{\text{ext}}$ and the functionality $\mathcal{F}_{\text{setup}}$, emulates the key-management functionality $\mathcal{F}_{\text{KM}}$. We achieve this result in two steps which we describe in two lemmas. The first step shows that the key-management functionality $\mathcal{F}_{\text{KM}}$ can be implemented by a similar functionality, which, instead of calling a ku functionality $\mathcal{F}_i$, calls the functions $\texttt{Impl}_{\mathcal{F}_i}$ that define a key-manageable implementation of $\mathcal{F}_i$. The resulting functionality $\mathcal{F}_{\text{KM}}^{\texttt{impl}} := \mathcal{F}_{\text{KM}}^{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_l/\texttt{Impl}_{\mathcal{F}_l}}$ acts as a "big" machine that all users in $\mathcal{U}$ access, just like $\mathcal{F}_{\text{KM}}$, but instead of substituting keys by credentials and forwarding requests to the ku functionalities, $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$ computes the output itself, using the implementation functions in $\texttt{Impl}_{\mathcal{F}_1} \ldots \texttt{Impl}_{\mathcal{F}_l}$. Now, the translation from credentials to keys (which takes place in the key-manageable implementations) is not necessary anymore, and consequently $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$ stores keys instead of credentials.

In the second step, we show that the computations that $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$ performs can be distributed. The security token network $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ performs the same computations on the key-usage part that $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$ performs, but implements the key-management part, including key-wrapping and unwrapping, using the algorithms $impl_{\text{new}}^{\texttt{KW}}$, wrap and unwrap. Under the assumption that those three algorithms constitute a secure and correct key-wrapping scheme, we can show that $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ implements $\mathcal{F}_{\text{KM}}^{\texttt{impl}}$ and, hence, by transitivity of the emulation relation, $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ emulates $\mathcal{F}_{\text{KM}}$.

The first step is subsumed by the following lemma.

Lemma 6: Let $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ be ku parameters such that all $F \in \overline{\mathcal{F}}$ are key-manageable. Let further $\texttt{Impl}_{\mathcal{F}_i}$ be the set of functions defining the key-manageable implementation $\hat{I}_i$ of $\mathcal{F}_i$. Then $\mathcal{F}_{\text{KM}}^{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_l/\texttt{Impl}_{\mathcal{F}_l}}$ emulates $\mathcal{F}_{\text{KM}}$. Furthermore, it is poly-time.

Proof Sketch (Full proof in Appendix D). Making use of the composition theorem, the last functionality $\mathcal{F}_l$ in $\mathcal{F}_{\text{KM}}$ can be substituted by its key-manageable implementation $\hat{I}_L$. Then, $\mathcal{F}_{\text{KM}}$ can simulate $\hat{I}_l$ instead of calling it. Let $\mathcal{F}_{\text{KM}}^{\{\mathcal{F}_l/\hat{I}_l\}}$ be the resulting functionality. In the next step, calls to this simulation are substituted by calls to the functions used in $\hat{I}_l$, i. e., $impl_C$ for each $C \in \mathcal{C}_l$ – so instead of addressing the simulation of $\hat{I}_l$ with the credentials, the function $impl_C$ is called with the key that the credential would otherwise be mapped to. The re-

sulting, partially implemented functionality $\mathcal{F}_{\text{KM}}^{\{\mathcal{F}_l/\text{Impl}_{\mathcal{F}_l}\}}$ saves keys rather than credentials (for $\mathcal{F}_l$). We repeat the previous steps until $\mathcal{F}_{\text{KM}}$ does not call any ku functionalities anymore, i. e., we have $\mathcal{F}_{\text{KM}}^{\{\mathcal{F}_1/\text{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_n/\text{Impl}_{\mathcal{F}_n}\}}$, i. e., $\mathcal{F}_{\text{KM}}^{\text{impl}}$. □

Before we come to Lemma 7, which describes the last step, we need to define what we understand under a key-wrapping scheme. We took the definition by Kremer et al. [58] as a basis, which is based on the notion of deterministic, authenticated encryption by Rogaway and Shrimpton [86], but additionally supports key-dependant messages. In contrast to the definition by Kremer et al., we allow wrapping the same key with the same wrapping key but under different attributes, which is also permitted in Rogaway and Shrimpton's definition.

Definition 31 (Multi-user setting for key wrapping)*:* We define experiments $\mathbf{Exp}^{\text{wrap},}_{\mathcal{A},\text{KW}}(\eta)$ and $\mathbf{Exp}^{\text{wrap,fake}}_{\mathcal{A},\text{KW}}(\eta)$ for a key-wrapping scheme $KW = (\text{KG}, \text{wrap}, \text{unwrap})$. In both experiments the adversary can access a number of keys $k_1, k_2, \ldots, k_n \ldots$ (which he can ask to be created via a query NEW). In his other queries, the adversary refers to these keys via symbols $K_1, K_2, \ldots, K_n$ (where the implicit mapping should be obvious). By abusing notation we often use K_i as a placeholder for k_i so, for example, $\text{wrap}^a_{K_i}(K_j)$ means $\text{wrap}^a_{k_i}(k_j)$. We now explain the queries that the adversary is allowed to make, and how they are answered in the two experiments.

- $\text{NEW}(K_i)$: a new key k_i is generated via $k_i \leftarrow \text{KG}(\eta)$
- $\text{ENC}(K_i, a, m)$ where $m \in \mathcal{K} \cup \{K_i \mid i \in \mathbb{N}\}$ and $h \in \mathcal{H}$. The experiment returns $\text{wrap}^a_{k_i}(m)$.
- $\text{TENC}(K_i, a, m)$ where $m \in \mathcal{K} \cup \{K_i \mid i \in \mathbb{N}\}$ and $a \in \mathcal{H}$. The real experiment returns $\text{wrap}^a_{k_i}(m)$, whereas the fake experiment returns $\$^{|\text{wrap}^a_{k_i}(m)|}$
- $\text{DEC}(K_i, a, c)$: the real experiment returns $\text{unwrap}^a_{k_i}(c)$, the fake experiment returns $\perp$.
- $\text{CORR}(K_i)$: the experiment returns k_i

Correctness of the wrapping scheme requires that for any $k_1, k_2 \in \mathcal{K}$ and any $a \in \mathcal{H}$, if $c \leftarrow \text{wrap}^a_{k_1}(k_2)$ then $\text{unwrap}^a_{k_1}(c) = k_1$.

Consider the directed graph whose nodes are the symbolic keys K_i and in which there is an edge from K_i to K_j if the adversary issues a query $\text{ENC}(K_i, a, K_j)$. We say that a key K_i is corrupt if either the adversary corrupted the key from the start, or if the key is reachable in the above graph from a corrupt key. If a handle, respectively pointer, points to a corrupted key, we call the pointer corrupted as well.

We make the following assumptions on the behaviour of the adversary.

- For all i the query $\mathsf{NEW}(K_i)$ is issued at most once.
- All the queries issued by the adversary contain keys that have already been generated by the experiment.
- The adversary never makes a test query $\mathsf{TENC}(K_i, a, K_j)$ if K_i is corrupted at the end of the experiment.
- If A issues a test query $\mathsf{TENC}(K_i, a, m)$ then A does not issue $\mathsf{TENC}(K_j, a', m')$ or $\mathsf{ENC}(K_j, a', m')$ for $(K_i, a, m) = (K_j, a', m')$
- The adversary never queries $\mathsf{DEC}(K_i, a, c)$ if c was the result of a query $\mathsf{TENC}(K_i, a, m)$ or of a query $\mathsf{ENC}(K_i, a, m)$ or K_i is corrupted.

At the end of the execution the adversary has to output a bit b which is also the result of the experiment. The advantage of adversary A in breaking the key-wrapping scheme KW is defined by:

$$\begin{aligned} \mathrm{Adv}^{\mathsf{wrap}}_{\mathsf{KW},A}(\eta) = \big|\Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},A}(\eta) \ : b = 1\right] - \\ \Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{wrap,fake}}_{\mathsf{KW},A}(\eta) \ : b = 1\right]\big| \end{aligned}$$

and KW is secure if the advantage of any probabilistic polynomial time algorithm is negligible.

The second and last step assumes a key-wrapping scheme that is secure with respect to this definition. It is subsumed by the following lemma, which shows that the completely implemented, but still monolithic functionality $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ can be emulated by a set of security tokens, each only managing the set of keys that belongs to its associated user. A necessary condition is that a secure and correct key-wrapping scheme is used. In this step, we need to restrict the set of environments to those which guarantee that keys are not corrupted after they have been used to wrap. Otherwise, we would provoke the commitment problem [50]. To formally define this class of environments, we introduce the notion of a *guaranteeing environment*, and a predicate on the inputs and outputs an environment receives and sends in a given execution, which is called *corrupt-before-wrap*. Both can be found in Appendix D.

Lemma 7: For any ku parameter $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and set of sets of PPT algorithms $\overline{\mathtt{Impl}}$, let $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_l/\mathtt{Impl}_{\mathcal{F}_l}}$ be the partial implementation of $\mathcal{F}_{\mathrm{KM}}$ with respect to all ku functionalities in $\overline{\mathcal{F}}$. If $KW = (impl^{\mathsf{KW}}_{\mathsf{new}}, \mathsf{wrap}, \mathsf{unwrap})$ is a secure and correct key-wrapping scheme (Definition 31) then $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ emulates $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ for environments that guarantee *corrupt-before-wrap*.

Proof sketch (Full proof in Appendix D). This lemma is proven using a reduction to the security of the key-wrapping scheme. We show that any environment that is able to distinguish $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ from $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ can

be transformed into an adversary against the fake-or-real game described in Definition 31. Fix such an environment Z. We first have to show that an attacker we construct from Z, which we call $\mathcal{B}_{\mathsf{Z}}$, is valid with respect to the fake-or-real game, i. e., that it does not repeat queries, etc. Then we show that the distinguishing environment Z interacting with $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ and a (carefully crafted) simulator has the same output distribution as $\mathcal{B}_{\mathsf{Z}}$ in the fake experiment and that Z interacting with $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ and a dummy adversary has the same output distribution as $\mathcal{B}_{\mathsf{Z}}$ in the real experiment. This allows us to conclude that a distinguishing environment Z would imply a distinguishing attacker for the key-wrapping game and thus contradict the assumption that *KW* is secure and correct. □

The main result follows from the transitivity of emulation and Lemmas 6 and 7:

Corollary 1: Let $\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi$ be ku parameters such that all $\mathsf{F} \in \overline{\mathcal{F}}$ are key-manageable. Let $\mathtt{Impl}_{\mathcal{F}_i}$ be the set of functions defining the key-manageable implementation $\hat{\Pi}_i$ of $\mathcal{F}_i$. If $KW = (impl_{\mathsf{new}}^{\mathsf{KW}}, \mathsf{wrap}, \mathsf{unwrap})$ is a secure and correct key-wrapping scheme (see Definition 31), then $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ emulates $\mathcal{F}_{\mathrm{KM}}$ for environments that guarantee *corrupt-before-wrap*.

11 A SIMPLE CASE STUDY

To demonstrate the use of our main result, we will instantiate the generic reference implementation of a security API presented in Chapter 8 with a policy and two functionalities. If the functionalities are key-manageable, then, by Corollary 1 from the previous chapter, we have a concrete implementation that is secure with respect to our definition. We furthermore demonstrate how two security token using this implementation can be employed in a higher-level protocol to implement an authenticated channel.

11.1 REALIZING $\mathcal{F}_{\text{KM}}$ FOR A STATIC KEY-HIERARCHY

We equip the security token with the functionalities $\mathcal{F}_1 = \mathcal{F}_{\text{Rand}}$ and $\mathcal{F}_2 = \mathcal{F}_{\text{SIG}}$ described below. The resulting token $ST^{\mathcal{F}_{\text{Rand}},\mathcal{F}_{\text{SIG}}}$ is able to encrypt keys and random values and sign user-supplied data. It is not able to sign keys, as this task is part of the key-management. The first functionality, $\mathcal{F}_{\text{Rand}}$, is unusual, but demonstrates what can be done within the design of $\mathcal{F}_{\text{KM}}$, as well as its limitations. It models how random values can be stored as keys, with equality tests and corruption, which means here that the adversary learns the value of the random value. Since our framework requires a strict division between key-management and usage, they can be transmitted (using wrap) and compared, but not appear elsewhere, since other ku functionalities shall not use them. We define $\mathcal{F}_{\text{Rand}}$ as follows:

```
new: accept <new> from parentId (=:p);
        c ← {0,1}^η; L ← L∪ {(c,0)}; send <new•,c,> to p
command: accept <equal,c,n> from p;
        if (c,k) ∈ L for some k
                if k ∉ K_cor
                        send <equal•,false> to p
                else if n = k
                        send <equal•,true> to p
corrupt: accept <corrupt,c> from p;
        if (c,0) ∈ L
                k ← {0,1}^η;
                L ← (L \ {(c,0)}) ∪ {(c,k)};
                K_cor = K_cor ∪ {k};
                send <corrupt•, k> to A
inject: accept <inject,n> from P;
        (c,_) ← {0,1}^η;
        K_cor ← K_cor ∪ {n};
        L ← L ∪ {(c,n)};
        send <inject•,c> to parentId
```

The two functions $impl_{\texttt{new}}$ and $impl_{\texttt{equal}}$ give the key-manageable implementation: $\texttt{impl}_{\texttt{new}}$ on input 1^η gives output $(n, \_)$ for $n \leftarrow \{0,1\}^\eta$; $impl_{\texttt{equal}}$ on input n, n' gives output $n = n'$.

The digital signature functionality is designed after the one described in [62] and detailed in Listing 10. It is parametrized by three algorithms KG, sign and verify. It expects the session parameter to encode a machine ID P, and implements $\mathcal{C}^{priv} = \{\texttt{sign}\}$ and $\mathcal{C}^{pub} = \{\texttt{verify}\}$.

Listing 10: A signature functionality $\mathcal{F}_{\text{SIG}}$.

```
new: accept <new> from P
        (sk, vk) ← KG(1^η); (credential,_) ← KG(1^η);
        L ← L∪ { (credential,sk,vk) };
    send <new•,credential,vk> to P
sign: accept <sign,credential,m> from P
        if (credential,sk,vk)∈L for some key
                σ ← sign(sk, m)
                if verify(vk, m, σ) ≠ ⊥ ∧ sk ∉ K_cor
                        signs[vk]= signs[vk] ∪{ (m, σ) }
                else σ ← ⊥
                send <sign•,σ> to P
verify: accept <verify,vk,<m, σ>> from P
        b ← verify(vk, m, σ)
        if ∃c, sk. (c,sk,vk)∈L and sk ∉ K_cor and b = 1 and
            ∄σ' : (m, σ') ∈signs[vk] or b ∉ {0, 1}
                b ← ⊥
        send <verify•,b> to P
corrupt: accept <corrupt,credential> from P
        if (credential,sk,vk)∈L for some sk, vk
                K_cor ← K_cor ∪{sk}; send <corrupt•,sk> to A
inject: accept <inject,<sk, vk>> from P
        (c,_) ← KG(1^η); K_cor ← K_cor ∪{sk};
        L ← L ∪{(c, sk, vk)}; send <inject•,c> to parentId
```

In the following, we will consider $\mathcal{F}_{\text{KM}}$ for the parameters $\overline{\mathcal{F}} = \{\mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{SIG}}\}$, $\overline{\mathcal{C}} = \{\{\texttt{equal}\}, \{\texttt{sign}, \texttt{verify}\}\}$ and a static key-hierarchy Π, which is defined as the relation that consists of all 4-tuples $(\mathcal{F}, \text{Cmd}, \text{attr}_1, \text{attr}_2)$ such that the conditions in one of the lines in the following table holds. Note that we omit the "=" sign when we mean equality and "*" denotes that no condition has to hold for the variable.

$\mathcal{F}$	Cmd	attr_1	attr_2
KW	new	> 0	*
$\neq$ KW	new	0	*
*	attr_change	a	a
KW	wrap	> 0	$\text{attr}_1 > \text{attr}_2$
KW	unwrap	> 0	$\text{attr}_1 > \text{attr}_2$
$\mathcal{F}_i$	$C \in \mathcal{C}^{priv}$	o	*

(where $a \in \mathbb{N}$)

Theorem 2 allows immediately to conclude some useful properties on this instantiation of $\mathcal{F}_{\text{KM}}$: From *(1)* we conclude that all keys with $c \notin \mathcal{K}_{\text{cor}}$ have the attribute they were created with. This also means that the same credential always has the same attribute, no matter which user accesses it. From *(2)*, we can see that for each corrupted credential $c \in \mathcal{K}_{\text{cor}}$, there was either a query $\langle \texttt{corrupt}, h \rangle$, where $\texttt{Store}[U, h] = \langle \mathcal{F}, a, c \rangle$, or there exist $\texttt{Store}[U, h'] = \langle \texttt{KW}, a', c' \rangle$ and $\texttt{Store}[U, h] = \langle \mathcal{F}, a, c \rangle$, and a query $\langle \texttt{wrap}, h', h, id \rangle$ was emitted, for $c' \in \mathcal{K}_{\text{cor}}$, or an unwrap query $\langle \texttt{unwrap}, h', w, a, F, id \rangle$ for a $c \in \mathcal{K}_{\text{cor}}$ was emitted. By the definition of the strict key-hierarchy policy, in the latter two cases we have that $a' > a$. It follows that, for any credential c for $\mathcal{F}$, such that $\texttt{Store}[U, h] = \langle \mathcal{F}, a, c \rangle$ for some U, h and a, we have that $c \notin \mathcal{K}_{\text{cor}}$, as long as every corruption query $\langle \texttt{corrupt}, h^* \rangle$ at U was addressed to a different key of lower or equal rank key, i. e., $\texttt{Store}[U, h^*] = \langle \texttt{KW}, a^*, c^* \rangle$, $c^* \neq c$ and $a^* \leqslant a$. By *(3)*, those credentials $c \notin \mathcal{K}_{\text{cor}}$ have not been corrupted in their respective functionality, i. e., it has never received a message $\langle \texttt{corrupt}, c \rangle$.

11.2 AN EXAMPLE IMPLEMENTATION OF THE AUTHENTICATED CHANNEL FUNCTIONALITY

The following implementation of the authenticated channel functionality $\mathcal{F}_{\text{ach}}$ from the introduction may serve as an example on how to use $\mathcal{F}_{\text{KM}}$ in a protocol. We repeat the definition of $\mathcal{F}_{\text{ach}}$ from Chapter 7 here:

Listing 11: $\mathcal{F}_{ach}$ with session parameters $\langle P_{pid}, Q_{pid}, label \rangle$. Note that in this example, every step can only be executed once.

```
ready-sender: accept <ready> from P;
   send <sender-ready> to A
ready-receiver[¬ready-sender]: accept <ready> from Q;
   send <ready-receiver-early> to A
ready-receiver[ready-sender]: accept <ready> from Q;
   send <receiver-ready> to A;
send [ready-receiver]: accept <send,x> from P;
   x̄ ← x; send <send,x̄> to A
done [send]: accept <done> from A;
   send <done> to P
deliver[send]: accept <deliver,x> from A where x = x;
   send <deliver,x̄> to Q
```

The idea is the following: Two parties, the sender and the recipient, use the set-up phase to generate a shared signature key. The recipient creates this signature key, stores the public part, shares the private part, which is hidden inside $\mathcal{F}_{\text{KM}}$, with the sender, and then announces the end of the set-up phase. At some later point, when

the sender is instructed to send a message, it attaches the signature to the message. The recipient accepts only messages that carry a valid signature, which she can verify using the public part of the shared signature key. Obviously, a similar implementation without the key-management functionality $\mathcal{F}_{\text{KM}}$ is possible (although other means of pre-sharing signature keys, or MAC keys for that matter, would be required). However, our aim is to provide a very concise use case.
Formally, the protocol π_{ach} defines three protocol names: `proto-ach`, `proto-fkm` and `proto-sig`. $\pi_{\text{ach}}(\texttt{proto-sig})$ is defined by the signature functionality $\mathcal{F}_{\text{SIG}}$. $\pi_{\text{ach}}(\texttt{proto-fkm})$ is defined by $\mathcal{F}_{\text{KM}}$, for the parameters $\overline{\mathcal{F}} = \{\mathcal{F}_{\text{SIG}}\}$, $\overline{\mathcal{C}} = \{\{\texttt{sign},\texttt{verify}\}\}$ and a static key-hierarchy Π as defined in the previous section. $\pi_{\text{ach}}(\texttt{proto-ach})$ parses the session parameter as a tuple $\langle \mathsf{P}_{pid}, \mathsf{Q}_{pid}, label\rangle$, where *label* is used to distinguish different channels, and $\mathsf{P}_{pid}, \mathsf{Q}_{pid}$ to identify the sender and the recipient. Let `sid` be the session ID. Then we will use P to denote $\langle \mathsf{P}_{pid}, \texttt{sid}\rangle$ and Q to denote $\langle \mathsf{Q}_{pid}, \texttt{sid}\rangle$. As we want to keep the example simple, we do not model party corruption. The following code defines the behaviour of the sender P:

```
ready-sender: accept <ready> from parentId;
        call F_KM with <ready>
import[ready-sender]: accept <share•,h'> from F_KM; h̄' ← h';
        call F_KM with <finish_setup>
send[import]: accept <send,x> from parentId;
        call F_KM with <sign,h̄',x>; accept <sign•,σ> from F_KM;
    send <x, σ> to A
done [send]: accept <done> from A;
        send <done> to parentId
```

When we say that P calls $\mathcal{F}_{\text{KM}}$, we mean that P sends a message to $\langle \mathsf{P}_{pid}, \langle \texttt{sid}, \langle \texttt{prot}-\texttt{fkm}, \langle \mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room}\rangle\rangle\rangle$, where $\mathcal{U} = \{\mathsf{P},\mathsf{Q}\}$, $\mathcal{U}^{\text{ext}} = \{\mathsf{Q}\}$, $\mathit{Room} = \{\mathsf{P},\mathsf{Q}\}$ and $\mathcal{ST}$ any two regular peers of P and Q, but not P and Q themselves. Similar for the receiver Q:

```
ready-receiver:
        accept <ready> from parentId; call F_KM with <ready>;
        accept <proceed> from A; call F_KM with <new,F_SIG,0>;
        accept <new,h,vk> from F_KM;
        h̄ ← h;
        v̄k ← vk;
        call F_KM with <share,h>
deliver[ready-receiver]:
        accept <deliver,x,σ> from A;
        call F_KM with <verify,v̄k,<x, σ>>;
        accept <verify,1> from F_KM;
        send <deliver,x> to parentId
```

The following lemma makes use of the fact that $\mathcal{F}_{\text{KM}}$ provides an authentic way to share keys during the set-up phase, and that $\mathcal{F}_{\text{SIG}}$ outputs $\langle \texttt{verify}, 1\rangle$ only if the corresponding message was "registered" before.

Lemma 8: π_{ach} emulates $\mathcal{F}_{\text{ach}}$.

Proof. We have to show that there exists a simulator *Sim* bounded for $\mathcal{F}_{\text{ach}}$ and that, for every well-behaved environment Z rooted at `prot-fach`,

$$\text{Exec}[\pi_{\text{ach}}, A_D, Z] \approx \text{Exec}[\mathcal{F}_{\text{ach}}, \mathit{Sim}, Z].$$

We define the following simulator *Sim*:

```
ready-sender:
        accept <sender-ready> from <ideal>
        send <F_KM,<ready•,P>> to <env>
faux-ready-receiver[¬ready-sender]:
        accept <ready-receiver-early> from <ideal>
        send <F_KM,<ready•,Q> to <env>
        accept <Q,proceed> from <env>
        send <F_KM,error> to <env>
ready-receiver[ready-sender]:
        accept <receiver-ready> from <ideal>
        send <F_KM, <ready•,Q>> to <env>
        accept <Q,proceed> from <env>
        (sk, vk) ← KG(1^η)
        send <F_KM,<finish_setup•>> to <env>
send [ready-receiver]:
        accept <send,x> from <ideal>
        σ ← sign(sk, x)
        send <P,<x,σ>> to <env>
done [send]:
        accept <P,done> from <env>
        send <done> to P
faux-deliver[ready-receiver∧¬send]:
        accept <Q,<deliver,x,σ>> from <env>
        send <Q,error> to <env>
deliver[send]:
        accept <Q,<deliver,x,σ>> from <env>
        if σ = σ
                send <deliver,x> to <ideal>
error:
        accept <P,error> from <ideal>
        send <P,error> to <env>
```

Let us fix the session ID `sid`, and assume it is of the form $\langle sid', \langle \texttt{prot-ach}, \langle P_{pid}, Q_{pid}, label\rangle\rangle \ldots\rangle$. Let $P = \langle P_{pid}, \texttt{sid}\rangle$ and $Q = \langle Q_{pid}, \texttt{sid}\rangle$. First, we show that *Sim* is bounded for $\mathcal{F}_{\text{ach}}$: It is trivial to verify that *Sim* is time-bounded, since KG and sign are assumed to be. Every flow from *Sim* to $\mathcal{F}_{\text{ach}}$ is provoked by an input from the environment, and the length of the message from *Sim* to $\mathcal{F}_{\text{ach}}$ is polynomially related to the length of the message from the environment to *Sim*.

Next, we show that, for every well-behaved environment Z rooted at `prot-fach`, $\text{Exec}[\pi_{\text{ach}}, A_D, Z] = \text{Exec}[\mathcal{F}_{\text{ach}}, \mathit{Sim}, Z]$, by showing a stronger invariant:

At the end of each epoch, the following conditions hold true:

1. The view of Z is the same in $[\pi_{\text{ach}}, \mathsf{A}_\mathsf{D}, \mathsf{Z}]$ and $[\mathcal{F}_{\text{ach}}, \mathit{Sim}, \mathsf{Z}]$.
2. If $\mathcal{F}_{\text{ach}}$ has finished a step S in $[\mathcal{F}_{\text{ach}}, \mathit{Sim}, \mathsf{Z}]$, *Sim* has finished the same step, and in $[\pi_{\text{ach}}, \mathsf{A}_\mathsf{D}, \mathsf{Z}]$, either P or Q has finished S.

Induction over the number of activations of Z. In the base case, Z has not called any party, so the invariant holds trivially. Now assume the invariant held true at the end of the previous activation of Z, after which Z sends a message m to some party. We have to show the invariant to hold true when this new epoch is over, i. e., Z is activated again. Case distinction over the steps that were completed by $\mathcal{F}_{\text{ach}}$ in $[\mathcal{F}_{\text{ach}}, \mathit{Sim}, \mathsf{Z}]$ before Z emitted m. Note that the guards define a partial order of the steps, therefore it is sufficient to perform the distinction over the last completed step:

1. $\mathcal{F}_{\text{ach}}$ *has not finished any step yet.* If Z sends $m = \langle \texttt{ready} \rangle$ to P, then in $[\mathcal{F}_{\text{ach}}, \mathit{Sim}, \mathsf{Z}]$, *Sim* will translate $\mathcal{F}_{\text{ach}}$'s response into $\langle \mathcal{F}_{\text{KM}}, \langle \texttt{ready}^\bullet, \mathsf{P} \rangle \rangle$, which is what Z would receive in $[\pi_{\text{ach}}, \mathsf{A}_\mathsf{D}, \mathsf{Z}]$ due to the definition of $\mathcal{F}_{\text{KM}}$. If Z sends the message to Q instead, in $[\mathcal{F}_{\text{ach}}, \mathit{Sim}, \mathsf{Z}]$ an error message is sent to *Sim*, as the guard for the step `ready-receiver` would not be fulfilled. By induction hypothesis, *Sim* is in the same state as $\mathcal{F}_{\text{ach}}$, and therefore simulates the error that $\mathcal{F}_{\text{KM}}$ would send out in $[\pi_{\text{ach}}, \mathsf{A}_\mathsf{D}, \mathsf{Z}]$, because it receives a `share` query from Q without having received `ready` from P before – otherwise Z would have sent `ready` to P and then $\mathcal{F}_{\text{ach}}$ would have finished this step. If any other message is sent, $\mathcal{F}_{\text{KM}}$ sends an error message to *Sim*, who forwards it to Z just like A_D does in $[\pi_{\text{ach}}, \mathsf{A}_\mathsf{D}, \mathsf{Z}]$.
2. $\mathcal{F}_{\text{ach}}$ *has finished* `ready-sender`. If Z sends $m = \langle \texttt{ready} \rangle$ to Q, in $[\pi_{\text{ach}}, \mathsf{A}_\mathsf{D}, \mathsf{Z}]$ it will receive $\langle \mathcal{F}_{\text{KM}}, \langle \texttt{ready}^\bullet, \mathsf{Q} \rangle$ from A_D. Similarly in $[\mathcal{F}_{\text{ach}}, \mathit{Sim}, \mathsf{Z}]$, *Sim* would wait for the same response to proceed. If *Sim* and Q in π_{ach} were in this step before Z sends m, and $m = \langle \mathsf{Q}, \texttt{proceed} \rangle$, then, in $[\pi_{\text{ach}}, \mathsf{A}_\mathsf{D}, \mathsf{Z}]$, Q would create a key on $\mathcal{F}_{\text{KM}}$, and save the handle as well as the public part. Before Z's next activation, $\mathcal{F}_{\text{KM}}$ would activate P on step `import` which would store the same handle (by definition of $\mathcal{F}_{\text{KM}}$) and finish the setup phase, producing as only observable output to Z the message $\langle \mathcal{F}_{\text{KM}}, \texttt{finish\_setup}^\bullet \rangle$. In $[\mathcal{F}_{\text{ach}}, \mathit{Sim}, \mathsf{Z}]$, *Sim* draws the secret and public part of the key itself (with the same algorithm KG that $\mathcal{F}_{\text{SIG}}$ uses, which is called by $\mathcal{F}_{\text{KM}}$). It would then produce the same output to Z. Any other message m would result in an error-message and treated as described before.
3. $\mathcal{F}_{\text{ach}}$ *has finished* `ready-receiver`. If Z sends $m = \langle \texttt{send}, x \rangle$ to Q, then Z would receive $\langle \mathsf{P}, \langle x, \sigma \rangle \rangle$, for an equally distributed σ, in both networks $[\pi_{\text{ach}}, \mathsf{A}_\mathsf{D}, \mathsf{Z}]$ and $[\mathcal{F}_{\text{ach}}, \mathit{Sim}, \mathsf{Z}]$, since, as

mentioned before, the value at the key-position is equally distributed. If Z sends $m = \langle Q, \langle \texttt{deliver}, x, \sigma \rangle\rangle$ to *Sim*, it receives $\langle Q, \texttt{error}\rangle$ as response. In $[\pi_{ach}, A_D, Z]$, A_D forwards this to Q who queries $\mathcal{F}_{KM}$. Because `deliver` was not executed yet, the array `signs` in $\mathcal{F}_{SIG}$ is empty. Since $\overline{vk}$ was created in $\mathcal{F}_{SIG}/\mathcal{F}_{KM}$ (since, by induction hypothesis, `ready-receiver` also holds in π_{ach}), and no corruption message is emitted by any party in π_{ach}, the response from $\mathcal{F}_{SIG}/\mathcal{F}_{KM}$ cannot be $\langle \texttt{verify}^\bullet, 1\rangle$. Therefore, the second accept step in `deliver` in Q fails, and $\langle Q, \texttt{error}\rangle$ is output, just as in $[\mathcal{F}_{ach}, Sim, Z]$. Any other message m would result in an error-message and treated as described before.

4. $\mathcal{F}_{ach}$ *has finished* `send`. If $m = \langle P, \texttt{done}\rangle$ is send to A_D in $[\pi_{ach}, A_D, Z]$, Z receives $\langle \texttt{done}\rangle$ from P. In $[\mathcal{F}_{ach}, Sim, Z]$, *Sim* translates this to $\langle \texttt{done}\rangle$, which $\mathcal{F}_{ach}$ relays to the same response via the dummy party P. If Z sends $m = \langle Q, \langle \texttt{deliver}, x, \sigma\rangle\rangle$ to A_D in $[\pi_{ach}, A_D, Z]$, it receives $\langle \texttt{deliver}, x\rangle$ from Q, but only in case that Z has sent a message $\langle \texttt{send}, x\rangle$ to P earlier, when P was in state `import`, and gave the output $\langle x, \sigma\rangle$ to A_D, since $\mathcal{F}_{SIG}$ keeps a list of previously signed message and their signatures. In $[\mathcal{F}_{ach}, Sim, Z]$, by definition of *Sim* and $\mathcal{F}_{ach}$, Z receives the same response from Q if x is the same x in a $\langle \texttt{send}, x\rangle$ query accepted earlier, and σ is the output produced by *Sim* before forwarding. Since $\mathcal{F}_{KM}$ may have only accepted such queries when `ready-receiver` was finished, and `done` or `deliver` were not yet called, this corresponds to the same input z in $[\pi_{ach}, A_D, Z]$. Any other message m would result in an error-message and treated as described before.

5. $\mathcal{F}_{ach}$ *has finished* `deliver`. No messages are accepted anymore (i. e., they lead to error-messages).

□

CONCLUSION

12 CONCLUSION AND PERSPECTIVES

The interface between trusted code in an enclosed environment, such as for example a security token, and untrusted code on the outside is called a *security API*. Security APIs are often used as a means to make other protocols more resistant against the corruption of protocol parties. The idea is to protect sensitive data from a potentially untrusted "outside", since a security token is simpler than a general-purpose system and designed with the intent of providing security. Thorough security analysis is indispensable for achieving this goal.

There are two orthogonal objectives for security analysis, which we have investigated in this thesis: First, there is the security of whatever protocol makes use of the security API. What security goals need to be achieved in this context depends, of course, on the protocol. Second, there is the security of the security API itself. As the designer of a security API can hardly predict all possible contexts in which his security API will be employed, a definition of security is necessary to help guiding the design of the security API, as well as its implementation, and provide evaluation criteria for both.

12.1 ANALYSIS OF PROTOCOLS USING SECURITY APIS

Previous work has identified a promising approach to establish security guarantees of protocols that employ security APIs. Modelling the security APIs as ordinary protocol parties, it is possible to use traditional protocol verification tools for analysis. However, as we pointed out in Chapter 3, security APIs are typically stateful; they contain a database that stores cryptographic keys and additional attributes to those keys. As opposed to typical network protocols, this means that a protocol execution is stateful, and more importantly, the state is mutable and non-monotonic. The approach of using Horn clause resolution to perform protocol verification, which is very successful for ordinary protocols, leads to false attacks in this case, since it is based on the (sound, but not complete) abstraction that protocol actions can be repeated arbitrarily often. Despite some efforts being put into incorporating parts of the state into the horn-clause representation of the protocol, these limitations are still present, as we have shown in a thorough evaluation of those approaches in Chapter 3.

In Section 3.6, we argued that multiset rewrite rules are better suited as a back-end for the analysis, since they model causal relationships more accurately. In Chapter 4, we gave support to this claim by presenting a case study in which we show the absence of replay

attacks in a protocol based on a security token, the Yubikey protocol, in the formal model. The verification of this protocol was particularly challenging because it involves storing values, which need to be compared to each other. An analysis of this protocol had been attempted before, using protocol verification tools based on Horn clauses, as well as a model-checking approach, but, in contrast to the present work, it was previously only possible for a fixed number of nonces. We furthermore modelled an extension to this protocol that makes use of a second, different security token and showed that an attack on this protocol is possible, but also how this attack can be thwarted. Our results has been obtained using multiset rewriting and provides formal results on both the Yubikey protocol and its extension in an unbounded model.

Building on the experience with this modelling, we recognized the need for a protocol representation which is closer to an actual implementation, yet still abstract. A manual encoding of protocols using multiset rewrite rules is a difficult process. Deriving a model that is accurate with respect to concurrent runs of processes and is able to recognises locking issues is possible; however, it is a tedious and error-prone task that requires a great deal of expertise to be done correctly. To remedy this shortfall, we introduced a calculus that can serve as a front-end in Chapter 5. This calculus is especially convenient for users familiar with the well-known ProVerif tool. It extends the applied pi calculus with synchronous communication by operators for database access and explicit locking and uses verification on multiset rewrite rules on the back-end. We presented a number of case studies to show our tool's use for the analysis of security APIs: the above mentioned Yubikey protocol, the examples that we used for evaluating Horn-clause resolution approaches in Chapter 3 and a fragment similar to the PKCS#11 standard.

Our translation makes it possible to include the subtle issues that can arise in the context of synchronisation and locking in the model, but keep the amount of time needed to model a protocol small. Encoding such models directly in multiset-rewrite rules is tiresome and difficult to do right.

We conclude that the analysis of security APIs benefits from a precise verification procedure, which can be provided by using multiset rewriting as a back-end, and our calculus and translation to allow for a convenient and versatile modelling of the protocol or security API in question.

Future work

AUTOMATIC GENERATION OF HELPING LEMMAS For 4 out of 7 case studies, our method required helping lemmas to find a proof. These lemmas are proven automatically, but they have to be given

by the user. We would like to further advance the degree of automation in our tool, for example as follows: We observed that many of the helping lemmas we formulated in the course of our case studies are instances of what Meier, Cremers and Basin call the *decryption chain case* [73]. Their work introduces two invariants which, for protocols that fall into a certain class, help generating machine-checkable proofs of security. It might be possible to instantiate these invariants to a protocol-specific helping lemma, given that the protocol is in the right class. Due to our approach by translation, we have access to the protocol in a high-level language, which gives us more structure to work with, as well as in form of low-level rewrite rules that we generate ourselves. This gives us better control and puts us in a more favorable situation to automatically generate helping lemmas.

LOOPS Certain protocols, for example the TESLA protocol [78] contain loops. There are various ways to model loops in the calculus we propose: First, using secret (and thus synchronous) channels and replication. The initial local variable are sent on a secret channel, and each process in the replication can pick these values up, perform some input/output and compute new values. If the loop shall continue, the new values are emitted on the same channel, otherwise, a message on a second secret channel permits leaving the loop and proceeding elsewhere. Second, using insert, lookup and lock. The procedure here is similar, but a lock is needed, since the store is an asynchronous channel. Third, using embedded multiset-rewrite rules and replication. Instead of transferring the local variables from process to process via a secret channel or the store, a fact can be used to the same end. The resulting multiset rewrite rules from a translation using the third method would be quite similar to the modelling of the TESLA protocol in tamarin [72]. Evaluating these different modellings of loops and, if necessary, extending the calculus by a loop operator and finding an efficient translation are possible venues of future research.

EQUIVALENCE PROPERTIES Privacy-type properties, for example the secrecy of a vote in e-voting protocols, are naturally formulated as equivalences between different instances of a protocol. Two very prominent notions of equivalence are labelled bisimilarity and observational equivalence, which have been formulated and shown to coincide in the applied pi calculus [2]. As the calculus we propose is similar to the applied pi calculus, adapting these two notions would be quite straight forward. To prove the equivalence of two processes, however, a different translation is necessary, as tamarin currently does not support equivalence between MSR systems.

ProVerif provides support for showing diff-equivalence, a stronger notion of equivalence than observation equivalence, for processes that

differ only in the choice of some terms [15]. While the method is sound, it sometimes produces false attacks. Subsequent work eliminates some of the false attacks in the case where conditionals are involved [26]. Both methods essentially translate two processes of similar structure into a set of Horn clauses and a reachability query. Future work could lift this translation to output multiset rewriting rules instead of Horn clauses, possibly using ideas from our current translation to incorporate constructs that are present in our calculus, but not in the applied pi calculus.

CASE STUDIES We see the strengths of the calculus we propose in the analysis of protocols that rely heavily on state, such as key-management APIs, and will give some possible targets in the following: A modelling of the full PKCS#11 standard would be both useful in practice and challengeing, since security policies for PKCS#11 can define recursive datatypes. Although this feature is rarely used in practice, an automated analysis could possibly help identifying new key-policies with useful properties. Academic proposals for key-management APIs that have only been proven by hand so far are of special interest, too, since they provide security properties that go beyond key secrecy. As such, we would like to mention the key-management API by Cortier, Steel and Wiedling [31] which allows for key-revocation, and the key-management API proposed by Daubignard, Lubicz and Steel [32], which supports public-key cryptography.

The TPM standard is another goal for analysis, especially since a modelling in our calculus has the potential to lift some restrictions present in previous work [35, 33, 9], for reasons detailled in Chapter 3. The new version 2.0 of the standard is currently in public review [95] and is therefore perhabs the most interesting target for a case study.

12.2 ANALYSIS OF SECURITY APIS

In the second part of this thesis, we presented a definition that covers a wide range of security APIs and helps assessing the security of a network of security APIs. This definition uses a framework that allows for universal composability, which offers the following advantages: First, security holds in arbitrary contexts. Second, we can define security for sets of security APIs, which enables us to show that a policy is respected globally. Third, we can use composition to define the security API as a combination of secure key-management and correct implementations of key-usage tasks. Fourth, our definition can be used in proofs of higher-level protocols without need for a reduction proof.

Our definition differs from previous definitions in this respect, as these were based on real-or-fake games and hence do not benefit from these advantage. With the exception of the definition proposed by

Kremer et al., they are tailored to specific APIs and thereby lack generality. In order to formulate the transport of keys in the "ideal world" setting of the universal composability framework, we introduced the concept of credentials, randomly drawn strings that allow the delegation of access rights to functionalities. We introduced our definition, along with the concept of credentials, in Chapter 8.

In Chapter 9 we established some properties that this functionality enjoys and that high-level protocol can benefit from. In the same chapter, we discussed the limitations to our approach in detail.

In Section 8.4, we demonstrated that our definition can guide the design of a *generic* implementation of key-management. This implementation is secure with respect to our definition and can serve as a template for the implementation of many security APIs, as it is independent of which functions it is supposed to offer, as long as those operations are implemented correctly. As a consequence, adding functionality to security APIs that follow this design does not require a new proof. A case study in Chapter 11 gave an example on how the functionality can be instantiated and be used in the analysis of a higher-level protocol.

Future work

GENERALISING KEY-TRANSPORT The proposed definition of security can be generalized. Currently, it is too strict. It specifies the way keys are transferred from one device to another very explicitly, thereby restricting the possible implementation of this task to deterministic, symmetric and authentic encryption schemes that allow for key-dependant messages. Future research can explore to what extend key-transport can be captured in a more abstract way, making the definition applicable to a wider range of security APIs.

GENERALISING KEY-USAGE The current interface $\mathcal{F}_{\text{KM}}$ provides to key-usage functionalities does not allow two keys to interact with another, although it might be safe to do this if the two keys belong to the same key-usage functionalities. For some applications, like the Yubikey protocol, it would also be useful to extend key-manageable implementations to be able to keep some kind of state, for example a running counter, which is then exported along with the key. Making the key-management more powerful in this regard would suggest formulating the policy in a more general way, too.

MAKING EXISTING FUNCTIONALITIES KEY-MANAGEABLE Relating key-manageable functionalities to existing functionalities in the literature could allow reusing existing proofs of emulation. Most existing functionalities, even those for signature and encryption, bind the permission to perform operations to the identity of some party.

This can be seen as unnecessarily restrictive, as many implementations could allow exporting keys, or even give them to trusted third parties, like security token.

We think it is possible to develop a technique for transforming functionalities that have implementations of a certain form, but are not key-manageable, into key-manageable functionalities. This form describes an implementation that stores a single key. Our conjecture is that, if an implementation of this form emulates an "ordinary" functionality, then we can construct a key-manageable implementation that emulates a key-manageable functionality consisting of several copies of the "ordinary" functionality. This way, existing proofs could be used to develop a secure implementation of cryptographic primitives in a plug-and-play manner. Investigating the restrictions of this approach could furthermore provide deeper insight about the modelling of keys in simulation-based security frameworks.

CASE STUDIES Case studies of bigger systems than the example we provide in this thesis would be useful in gaining insight into the use of the definition in practice, as well as an assessment of its limitations. It would be interesting to discover to what extend the PKCS#11 standard can be modelled. The fact that many PKCS#11 configurations are indeed insecure would suggest using our translation to identify a secure and practical configuration before performing analysis in the computational model. Features that do not conform our distinction between key-management and key-usage can be excluded in the first place, e. g., signatures on keys, asymmetric key-wrapping (which, in PKCS#11 is broken anyway [27]), etc. The difficulty of this analysis depends on the configuration chosen. If the configuration entails a policy that enforces a strict key-separation, then the proof can be performed in a more modular way. The resulting security token network can harden other protocols against party corruption, which suggests using the security network, or more precisely, the instantiation of $\mathcal{F}_{\text{KM}}$ it emulates, to to analyse higher-level protocols in typical domains where PKCS#11 is used, e. g., smart card sign-on mechanisms, disk encryption, and protocols like TLS.

KEY TRANSFER IN UC USING CREDENTIALS Finally, credentials might be useful in other contexts where keys can be transferred between protocol parties. One example that comes to mind is the modelling of protocols that have a key-agreement phase, followed by a communication phase that makes use of this key, for example the TLS protocol. Hardware accelerators allow for offloading the (computationally expensive) processing of the key-agreement phase and leave the server to process (less expensive) symmetric operations of the communication phase. After the key-agreement phase, session key is transferred between two different entities, the hardware accel-

erator, and the server. We think that this interaction is but one example where the use of credentials for the modelling of key-transfer between different functionalities appears to produce a very natural way of abstracting keys.

APPENDIX

A

LISTINGS FOR PART i

A.1 LISTINGS FOR CHAPTER 3

Listing 12: The modelling of Example 4 in StatVerif by Arapinis et al. [5].

```
fun pair/2.
fun aenc/3.
fun pk/1.
free left.
free right.
free init.
free c.
private free sl.
private free sr.

reduc
    car(pair(xleft, xright)) = xleft;
    cdr(pair(xleft, xright)) = xright;
    adec(u, aenc(pk(u), v, w)) = w.

query att:vs,pair(sl,sr).

let device =
    out(c, pk(k)) |
    ( ! in(c, x); lock(s); read s as y;
        if y = init then
            (if x = left then s := x; unlock(s)
            else if x = right then s := x; unlock(s))  ) |
    ( ! in(c, x); lock(s); read s as y;
        if y = left then out(c, car(adec(k, x))); unlock(s)
        else if y = right then out(c, cdr(adec(k, x))); unlock(s)
            ).

let user =
    ν r;
        out(c, aenc(pk(k), r, pair(sl,sr))).

process
    ν k; ν s; [s ↦ init] | device | ! user
```

Listing 13: An attempt to express Example 5 in StatVerif. Note that the key store is modelled as single state cell, which refers to a list of keys. This list grows in size for every newly generated key. ProVerif did not terminate on the horn clauses produced by StatVerif's translation procedure in the experiments we conducted.

```
fun pair/2.
fun senc/2.
free c.
free nil.
free store.

reduc
    car(pair(xleft, xright)) = xleft;
    cdr(pair(xleft, xright)) = xright;
    sdec(u, senc(u, v)) = v.

query att:vs,key.

let create_proc =
        in(c,x); ν key; ν handle; lock;
                        read store as list;
                        store := pair(pair(key,handle),list);
         unlock;
out(c,handle).

let wrap_proc =
        in(c,x); let h1 = car(x) in let h2 = cdr(x) in
                lock;
                        read store as list;
                unlock;
                out(input,pair(h1,list));
                in(output,k1);
                out(input,pair(h2,list));
                in(output,k2);
                if k1 = nil then 0
                else if k2 = nil then 0
                else out(c,senc(k1,k2)).

let fetch_key =
        in (input,x); let arg=car(x) in
                let list=cdr(x) in
                let h = car(list) in
                let r = cdr(list) in
                if arg=car(h) then
                        out(output,cdr(h))
                else if r=nil then
                        out(output,nil)
                else
                        out(input,pair(arg,r)).

process
        ν input; ν output;
        ν store;
        [store ↦ nil] | ! create_proc | ! wrap_proc | ! fetch_key
```

Listing 14: A modelling of Example 4 in AIF.

```
Problem: Leftrightincorrect;
% not correct: the model actually allows the attack
% since the same secrets are used for every newly generated
    machine.

Types:
Secrets: {sl,sr};
SK,sk: value;
K,M,X,Y,token : untyped;

Sets:
init(token),left(token),right(token);

Functions:
public senc/2, pair/2;
private inv/1;

Facts:
iknows/1, attack/0;

Rules:

%the intruder's deduction capabilities:
⟶ iknows(token);
iknows(K).iknows(M) ⟶ iknows(senc(K,M));
iknows(senc(M,K)).iknows(K) ⟶ iknows(M);
iknows(X).iknows(Y) ⟶ iknows(pair(X,Y));
iknows(pair(X,Y)) ⟶ iknows(X);
iknows(pair(X,Y)) ⟶ iknows(Y);

%Init:
sk ⟶ sk∈init(token);
SK∈init(token).  SK∉left(token).  SK∉right(token).
        ⟶ SK∈init(token).iknows(senc(pair(sl,sr),SK));

%Set left - or - set right:
SK∈init(token) ⟶ SK∈left(token);
SK∈init(token) ⟶ SK∈right(token);

%decryption
SK∈left(token).iknows(senc(pair(X,Y),SK)) ⟶ SK∈left(token).
    iknows(X);
SK∈right(token).iknows(senc(pair(X,Y),SK)) ⟶ SK∈right(token).
    iknows(Y);

%Security goal
iknows(sl).iknows(sr) ⟶attack;
```

Listing 15: Another modelling of Example 4 in AIF.

```
Problem: wrapdec;
```

```
Types:
% Secrets: {sl,sr};
SK: value;
K,M,X,Y,t : untyped;

Sets:
init(t),left(t),right(t);

Functions:
public senc/2, pair/2;
private sl/1, sr/1;

Facts:
iknows/1, attack/0;

Rules:

%the intruder's deduction capabilities:
iknows(K)·iknows(M) ⟶ iknows(senc(K,M));
iknows(senc(K,M))·iknows(K) ⟶ iknows(M);
iknows(X)·iknows(Y) ⟶ iknows(pair(X,Y));
iknows(pair(X,Y)) ⟶ iknows(X);
iknows(pair(X,Y)) ⟶ iknows(Y);

%Init:
 ⊣[SK]↦ SK∈init(t)·iknows(senc(SK,pair(sl(SK),sr(SK))));

%Set left - or - set right:
SK∈init(t) ⟶ SK∈left(t);
SK∈init(t) ⟶ SK∈right(t);

%decryption
SK∈left(t)·iknows(senc(SK,pair(X,Y))) ⟶ SK∈left(t)·iknows(X);
SK∈right(t)·iknows(senc(SK,pair(X,Y))) ⟶ SK∈right(t)·iknows(Y
    );

%Security goal
iknows(sl(SK)) ·iknows(sr(SK)) ⟶attack;
```

Listing 16: A modelling of Example 5 in AIF.

```
Problem: Wrapdec;

Types:
Agents: {t}; % Agents: Token, Intruder
Token:  {t}; % one token

KEY, HANDLE, KEY1,KEY2: value;
K,M: untyped;

Sets:
store(Token),init(Token),wrap(Token),dec(Token);
```

```
Functions:
public senc/2;
private h/1;

Facts:
iknows/1, attack/0;

Rules:

% The intruder's deduction capabilities:
∀ Agents· ⟶ iknows(Agents);
iknows(K)·iknows(M) ⟶ iknows(senc(K,M));
iknows(senc(K,M))·iknows(K) ⟶ iknows(M);

% Generate a key
∀ Token· ⊣[KEY]↦ iknows(h(KEY))·KEY∈store(Token)·KEY∈init(
    Token);

% Wrap
∀ Token· KEY1∈wrap(Token)· iknows(h(KEY1))· iknows(h(KEY2))·
KEY1∈store(Token)· KEY2∈store(Token)
⟶ iknows(senc(KEY1,KEY2))·
KEY1∈wrap(Token)· KEY1∈store(Token)· KEY2∈store(Token);

% SDecrypt
∀ Token·iknows(h(KEY))·iknows(senc(KEY,M))·KEY∈store(Token)·KEY
    ∈dec(Token)
⟶ iknows(M)·KEY∈store(Token)·KEY∈dec(Token);

% Security goal
∀ Token · KEY∈store(Token) ·iknows(KEY) ⟶attack;

%  Policy 1
∀ Token· KEY∈store(Token)· KEY∉dec(Token) · KEY∈init(Token) ⟶
    KEY∈store(Token)· KEY∈wrap(Token);
∀ Token· KEY∈store(Token)· KEY∉wrap(Token) · KEY∈init(Token)
    ⟶ KEY∈store(Token)· KEY∈dec(Token);

%  Policy 2
%  ∀ Token· KEY∈store(Token)· KEY∉dec(Token)  ⟶ KEY∈store(
    Token)· KEY∈wrap(Token);
%  ∀ Token· KEY∈store(Token)· KEY∉wrap(Token)  ⟶ KEY∈store(
    Token)· KEY∈dec(Token);

%  Policy 3
%  ∀ Token· KEY∈store(Token)· KEY∈init(Token)  ⟶ KEY∈store(
    Token)· KEY∈wrap(Token);
%  ∀ Token· KEY∈store(Token)· KEY∈init(Token)  ⟶ KEY∈store(
    Token)· KEY∈dec(Token);
%  ∀ Token· KEY∈store(Token)· KEY∈dec(Token)  ⟶ KEY∈store(
    Token)· KEY∈wrap(Token);
```

```
% Bad policy -- attack is reachable·
∀ Token· KEY ∈ store(Token)· KEY ∈ init(Token)   ⟶ KEY ∈ store(
    Token)· KEY ∈ wrap(Token);
∀ Token· KEY ∈ store(Token)· KEY ∈ init(Token)   ⟶ KEY ∈ store(
    Token)· KEY ∈ dec(Token);
∀ Token· KEY ∈ store(Token)· KEY ∈ wrap(Token)   ⟶ KEY ∈ store(
    Token)· KEY ∈ dec(Token);
```

A.2 LISTINGS FOR CHAPTER 4

Listing 17: Modelling of the Yubikey authentication protocol in tamarin's MSR calculus. The counter is modelled as a term constituting a multiset of constants, where the cardinality of the multiset equals the size of the counter.

```
theory Yubikey
begin

section{* The Yubikey Authentication Protocol *}


builtins: symmetric-encryption, multiset

// Initialisation and Setup of a Yubikey
rule BuyANewYubikey:
        [Fr(~k),Fr(~pid),Fr(~sid)]
        −[Protocol(), Init(~pid,~k),ExtendedInit(~pid,~sid,~k)]→
        [!Y(~pid,~sid), Y_counter(~pid,'1'),
                 Server(~pid,~sid,'1'),!SharedKey(~pid,~k),
         Out(~pid)]

rule Yubikey_Plugin:
           [Y_counter(pid,otc), In(tc) ]
        −[ Yubi(pid,tc), Smaller(otc,tc) ]→
        [Y_counter(pid, tc)]

//If the Button is pressed, the token counter is increased
rule Yubikey_PressButton:
        [!Y(pid,sid), Y_counter(pid,tc),!SharedKey(pid,k),
         In(tc),Fr(~npr),Fr(~nonce) ]
        −[ YubiPress(pid,tc) ]→
        [Y_counter(pid, tc + '1'),
         Out(<pid,~nonce,senc(<sid,tc,~npr>,k)>)
        ]

/* Upon receiving an encrypted OTP, the Server compares the
 * (unencrypted) public id to his data base to identify the
 * key to decrypt the OTP. After making sure that the secret
 * id is correct, the Server verifies that the received
 * counter value is larger than the last one stored. If the
```

```
 * Login is successful, i.e., the previous conditions were
 * fulfilled, the counter value on the Server that is
 * associated to the Yubikey is updated.
 */

rule Server_ReceiveOTP_NewSession:
       [ Server(pid,sid,otc),
         In(<pid,nonce,senc(<sid,tc,~pr>,k)>),
         !SharedKey(pid,k), In(otc) ]
        -[ Login(pid,sid,tc,senc(<sid,tc,~pr>,k)),
           LoginCounter(pid,otc,tc),
                       Smaller(otc,tc) ]->
       [ Server(pid,sid,tc) ]

//model the larger relation using the smaller action and
//exclude all traces where the predicate does *not* hold true
axiom smaller:
       "∀ #i a b. Smaller(a,b)@i ⟶ ∃ z. a+z=b"

// For sanity: Ensure that a successful login is reachable.
lemma Login_reachable:
  exists-trace
  "∃ #i pid sid x otp1. Login(pid,sid,x,otp1)@i"

// If a succesful Login happens before a second sucesfull
// Login, the counter value of the first is smaller than the
// counter value of the second
lemma invariant[reuse, use_induction]:
       "(∀ pid otc1 tc1 otc2 tc2 #t1 #t2 .
            LoginCounter(pid,otc1,tc1)@#t1 &
                        LoginCounter(pid,otc2,tc2)@#t2
       ⟶ ( #t1<#t2 ∧ ( ∃ z . tc1+z=tc2))
            ∨ #t2<#t1 ∨ #t1=#t2)
       "

// It is ¬possible to have to distinct logins with the
// same counter value
lemma no_replay:
       "¬ ( ∃ #i #j pid sid x otp1 otp2 .
        Login(pid,sid,x,otp1)@i ∧ Login(pid,sid,x,otp2)@j
         ∧ not(#i=#j))"

lemma injective_correspondance:
    "∀ pid sid x otp  #t2 . Login(pid,sid,x,otp)@t2 ⟶
         ( ∃ #t1  . YubiPress(pid,x)@#t1 ∧ #t1<#t2
        ∧ ∀ otp2 #t3 . Login(pid,sid,x,otp2)@t3 ⟶ #t3=#t2
       )"

lemma Login_invalidates_smaller_counters:
       "∀ pid otc1 tc1 otc2 tc2 #t1 #t2 #t3 .
            LoginCounter(pid,otc1,tc1)@#t1 &
                        LoginCounter(pid,otc2,tc2)@#t2
```

```
                        ∧ Smaller(tc1,tc2)@t3
                ⟶  #t1<#t2 "
end
```

Listing 18: Modelling of the Yubikey authentication protocol in conjunction with the YubiHSM, formalised in tamarin's MSR calculus. The counter is modelled as a term constituting a multiset of constants, where the cardinality of the multiset equals the size of the counter.

```
theory YubikeyHSM
begin

section{* The Yubikey-Protocol with a YubiHSM *}

builtins: symmetric-encryption, multiset

functions:
        keystream/2,  keystream_kh/1, keystream_n/1,
        xor/2, zero/0,
        mac/2, demac/2

equations:
        keystream_kh(keystream(kh,n))=kh,
    keystream_n(keystream(n,n))=n,
        /* models the way the key-stream used for encryption is
         * computed */
                        xor(xor(a,b),a)=b,
                        xor(xor(a,b),b)=a,
                        xor(a,a)=zero,
                        xor(zero,a)=a,
                        xor(a,zero)=a,
        /* incomplete modelling of xor */
        demac(mac(m,k),k)=m
        /* describes the MAC used inside the AEADs
         * using mac, adv might find out *something* about the
         * message, we over-approximate */

// Rules for intruder's control over Server

// send messages to the HSM
rule isendHSM:
   [ In( x ) ] -[ HSMWrite(x) ]-> [ InHSM( x ) ]
rule irecvHSM:
   [ OutHSM( x ) ] -[ HSMRead(x) ]-> [Out(x)]

// write and read the Authentication Server's database
rule read_AEAD:
    [ !S_AEAD(pid,aead)  ]
        -[ AEADRead(aead),HSMRead(aead) ]->
        [Out(aead)]
rule write_AEAD:
```

```
    [ In(aead), In(pid) ]
         --[ AEADWrite(aead),HSMWrite(aead) ]->
        [!S_AEAD(pid,aead) ]


// Initialisation of HSM and Authentication Server.
rule HSMInit:
    [Fr(~k), Fr(~kh)] --[MasterKey(~k), OneTime()]->
    [ !HSM(~kh,~k), Out(~kh),
//!YSM_AEAD_GENERATE(~kh), //uncomment to produce attack
!YSM_AEAD_YUBIKEY_OTP_DECODE(~kh)
]

//HSM commands
rule YSM_AEAD_RANDOM_GENERATE:
    let ks=keystream(kh,N)
        aead=<xor(senc(ks,k),~data),mac(~data,k)>
    in
    [Fr(~data), InHSM(<N,kh>),!HSM(kh,k),
          !YSM_AEAD_RANDOM_GENERATE(kh)]
    --[GenerateRandomAEAD(~data)]->
    [OutHSM( aead)
    ]

rule YSM_AEAD_GENERATE:
    let ks=keystream(kh,N)
        aead=<xor(senc(ks,k),data),mac(data,k)>
    in
    [InHSM(<N,kh,data>),!HSM(kh,k),!YSM_AEAD_GENERATE(kh)]
    --[GenerateAEAD(data,aead )]->
    [OutHSM( aead) ]

rule YSM_AES_ESC_BLOCK_ENCRYPT:
    [InHSM(<kh,data>), !HSM(kh,k),
         !YSM_AES_ESC_BLOCK_ENCRYPT(kh)]
    --[]->
    [OutHSM(senc(data,k))]

rule YSM_AEAD_YUBIKEY_OTP_DECODE:
    let ks=keystream(kh,N)
        aead=<xor(senc(ks,k),<k2,did>),mac(<k2,did>,k)>
        otp=senc(<did,sc,rand>,k2)
    in
    [InHSM(<did,kh,aead,otp>), !HSM(kh,k),
         !YSM_AEAD_YUBIKEY_OTP_DECODE(kh)
    ]
    --[
    OtpDecode(k2,k,<did,sc,rand>,sc,
                xor(senc(ks,k),<k2,did>),mac(<k2,did>,k)),
    OtpDecodeMaster(k2,k)
    ]->
    [OutHSM(sc)]
```

```
95
//Yubikey operations

rule BuyANewYubikey:
    let ks=keystream(kh,~pid)
100     aead=<xor(senc(ks,~k),<~k2,~sid>),mac(<~k2,~sid>,~k)>
    in
//This rule implicitly uses YSM_AEAD_GENERATE
    [ Fr(~k2),Fr(~pid),Fr(~sid),
          //!YSM_AEAD_GENERATE(kh),
105       //Uncomment to require the HSM to have
              YSM_AEAD_GENERATE
      !HSM(kh,~k), In('1') ]
     -[Init(~pid,~k2)]->
    [Y_counter(~pid,'1'), !Y_Key(~pid,~k2), !Y_sid(~pid,~sid),
     S_Counter(~pid,'1'), !S_AEAD(~pid,aead), !S_sid(~pid,~sid),
110  Out(~pid) ]

//On plugin, the session counter is increased and the token
//counter reset
rule Yubikey_Plugin:
115     [Y_counter(pid,sc),In(Ssc) ]
        //The old counter value sc is removed
        -[ Yubi(pid,Ssc),Smaller(sc, Ssc) ]->
        [Y_counter(pid, Ssc)]
        //and substituted by a new counter value, larger, Ssc
120

rule Yubikey_PressButton:
    [Y_counter(pid,tc),!Y_Key(pid,k2),!Y_sid(pid,sid),
     Fr(~pr),Fr(~nonce), In(tc+'1')]
    -[ YubiPress(pid,tc),
125     YubiPressOtp(pid,<sid,tc,~pr>,tc,k2) ]->
    [Y_counter(pid,tc+'1'),
         Out(<pid,~nonce,senc(<sid,tc,~pr>,k2)>)]

rule Server_ReceiveOTP_NewSession:
130 let ks=keystream(kh,pid)
        aead=<xor(senc(ks,k),<k2,sid>),mac(<k2,sid>,k)>
    in
    [In(<pid,nonce,senc(<sid,tc,~pr>,k2)>) ,
        !HSM(kh,k), !S_AEAD(pid,aead), S_Counter(pid,otc),
135     !S_sid(pid,sid) ]
     -[ Login(pid,sid,tc,senc(<sid,tc,~pr>,k2)),
                LoginCounter(pid,otc,tc), Smaller(otc,tc) ]->
    [ S_Counter(pid,tc) ]

140 //model the larger relation using the smaller action and
//exclude all traces where the predicate does *not* hold true
axiom smaller:
        "∀ #i a b. Smaller(a,b)@i ⟶ ∃ z. a+z=b"

145 axiom onetime:
```

```
    "∀ #t3 #t4 . OneTime()@#t3 ∧ OneTime()@t4 ⟶ #t3=#t4"

// For sanity: Ensure that a successful login is reachable.
lemma Login_reachable:
  exists-trace
  "∃ #i pid sid x otp1. Login(pid,sid, x, otp1)@i"

/* Every counter produced by a Yubikey could be computed by
 * the adversary anyway. (This saves a lot of steps in the
 * backwards induction of the following lemmas).  */
lemma adv_can_guess_counter[reuse,use_induction]:
    "∀ pid sc #t2 . YubiPress(pid,sc)@t2
    ⟶ (∃ #t1 . K(sc)@#t1 ∧ #t1<#t2)"

/* Everything that can be learned by using OtpDecode is the
 * counter of a Yubikey, which can be computed according to
 * the previous lemma.  */
lemma
otp_decode_does_not_help_adv_use_induction[reuse,use_induction]:
    "∀ #t3 k2 k m sc enc mac . OtpDecode(k2,k,m,sc,enc,mac)@t3
    ⟶ ∃ #t1 pid . YubiPress(pid,sc)@#t1 ∧ #t1<#t3"

/* ∀ keys shared between the YubiHSM and the
 * Authentication Server are either *not* known to the
 * adversary, or the adversary learned the key used to
 * encrypt said keys in form of AEADs.  */
lemma k2_is_secret_use_induction[use_induction,reuse]:
    "∀ #t1 #t2 pid k2 . Init(pid,k2)@t1 ∧ K(k2)@t2
    ⟶
     (∃ #t3 #t4 k . K(k)@t3 ∧ MasterKey(k)@t4 ∧ #t3<#t2)"

/* Neither of those kinds of keys are ever learned by the
    adversary */
lemma neither_k_nor_k2_are_ever_leaked_inv[use_induction,reuse]:
    "
not( ∃ #t1 #t2 k . MasterKey(k)@t1 ∧ KU(k)@t2 )
 ∧ ¬(∃ #t5 #t6 k6 pid . Init(pid,k6)@t5 ∧ KU(k6)@t6 )
    "

// Each successful login with counter value x was preceded by a
    button press
// event with the same counter value
lemma one_count_foreach_login[reuse,use_induction]:
       "∀ pid sid x otp #t2 . Login(pid,sid,x,otp)@t2 ⟶
        ( ∃ #t1 . YubiPress(pid,x)@#t1 ∧ #t1<#t2 )"

// If a successful Login happens before a second successful
// Login, the counter value of the first is smaller than the
// counter value of the second
lemma slightly_weaker_invariant[reuse, use_induction]:
       "(∀ pid otc1 tc1 otc2 tc2 #t1 #t2 .
```

```
            LoginCounter(pid,otc1,tc1)@#t1  ∧ LoginCounter(pid,
                otc2,tc2)@#t2
        ⟶ ( #t1<#t2  ∧ ( ∃ z . tc2=tc1 + z))
            ∨ #t2<#t1  ∨ #t1=#t2)
        "
induction
  case empty_trace
  by contradiction // from formulas
next
  case non_empty_trace
  simplify
  solve( (¬(#t1 < #t2))  ∨ (∀ z.2. ((otc2+z.1) = (otc1+z+z.2)) ⇒
      ⊥) )
    case case_1
    solve( (#t2 = #t1)  ∨ (#t1 < #t2) )
      case case_1
      by contradiction // from formulas
    next
      case case_2
      by contradiction // from formulas
    qed
  next
    case case_2
    solve( (#t2 = #t1)  ∨ (#t1 < #t2) )
      case case_1
      by contradiction // from formulas
    next
      case case_2
      solve( S_Counter( pid, otc2 ) ▶₃ #t2 )
        case BuyANewYubikey
        by sorry
      next
        case Server_ReceiveOTP_NewSession_case_1
        by sorry
      next
        case Server_ReceiveOTP_NewSession_case_2
        by sorry
      qed
    qed
  qed
qed


// It is *not* possible to have to distinct logins with the
// same counter value
lemma no_replay:
        "¬ ( ∃ #i #j pid sid x otp1 otp2 .
         Login(pid,sid,x,otp1)@i  ∧  Login(pid,sid,x,otp2)@j
          ∧ not(#i=#j))"

// Every Login was preceded by exactly one corresponding button
    press
```

```
lemma injective_correspondance:
    " ∀ pid sid x otp  #t2 . Login(pid,sid,x,otp)@t2 ⟶
         ( ∃ #t1  . YubiPress(pid,x)@#t1  ∧ #t1<#t2
        ∧ ∀ otp2 #t3 . Login(pid,sid,x,otp2)@t3 ⟶ #t3=#t2
        )"

// If one login has a smaller counter than the other, it must
    have occurred earlier
lemma Login_invalidates_smaller_counters:
        " ∀ pid otc1 tc1 otc2 tc2 #t1 #t2 z .
            LoginCounter(pid,otc1,tc1)@#t1
                ∧ LoginCounter(pid,otc2,tc2)@#t2
                    ∧ tc2=tc1+z
            ⟶  #t1<#t2 "

end
```

Listing 19: Modelling of the Yubikey authentication protocol formalised in tamarin's MSR calculus. The counter is modelled using a "successor of" constructor. The permanent fact !IsSmaller can be produced by the adversary to proof that one counter value is smaller than another. We need to enforce transitivity in order to be able to proof our claims – that means, we require (using an axiom) each trace to contain IsSmaller(a, c), should IsSmaller(a, b) and !IsSmaller(a, c) be present, for a, b, c ∈ $\mathcal{M}$. !Succ models a functional relation: If !Succ(a, b), then the adversary was able to show that b is the successor of a. The relation modelled by !Smaller is not functional: If !Smaller(a, b), then the adversary was able to show that a is smaller than b. The Theory() action is used to enforce that this relation (to the extend it is needed in this trace) has to be build up before running the first protocol actions.

```
theory Yubikey-noms
begin

section{* The Yubikey-Protocol (alternative modelling) *}

builtins: symmetric-encryption

functions: S/1,zero/0

// Rules modelling the "successor of" relation
rule InitSucc:
    [In(zero),In(S(zero))]
        −[Theory(), IsSucc(zero,S(zero)),IsZero(zero)]→
        [!Succ(zero,S(zero))]

rule StepSucc:
    [In(y),In(S(y)), !Succ(x,y)]
        −[Theory(), IsSucc(y,S(y)) ]→
        [!Succ(y,S(y))]
```

```
// Rules modelling the "smaller than" relation
rule SimpleSmaller:
    [!Succ(x,y)]
        −{Theory(), IsSmaller(x,y)}→
        [!Smaller(x,y)]

rule ZExtendedSmaller:
    [!Smaller(x,y),!Succ(y,z)]
        −{Theory(), IsSmaller(x,z)}→
        [!Smaller(x,z)]

// Rules modelling the Yubikey
rule BuyANewYubikey:
        [Fr(~k),Fr(~pid),Fr(~sid)]
        −{Protocol(), Init(~pid,~k),
           ExtendedInit(~pid,~sid,~k),IsZero(zero)}→
        [!Y(~pid,~sid), Y_counter(~pid,zero),
                 Server(~pid,~sid,zero),!SharedKey(~pid,~k), Out
                     (~pid)]

//On plugin, the session counter is increased and the token
    counter reset
rule Yubikey_Plugin:
        [Y_counter(pid,sc),!Smaller(sc, Ssc) ]
        −{ Yubi(pid,Ssc) }→
        [Y_counter(pid, Ssc)]

//If the Button is pressed, the token counter is increased
rule Yubikey_PressButton:
        [!Y(pid,sid), Y_counter(pid,tc),!SharedKey(pid,k),
         !Succ(tc,Stc),Fr(~npr),Fr(~nonce) ]
        −{ YubiPress(pid,tc)}→
        [Y_counter(pid, Stc),
         Out(<pid,~nonce,senc(<sid,tc,~npr>,k)>)
        ]

/* Upon receiving an encrypted OTP, the Server compares
 * the (unencrypted) public id to his data base to
 * identify the key to decrypt the OTP. After making
 * sure that the secret id is correct, the Server
 * verifies that the received counter value is larger
 * than the last one stored. If the Login is
 * successful, i.e., the previous conditions were
 * fulfilled, the counter value on the Server that is
 * associated to the Yubikey is updated.
 */

rule Server_ReceiveOTP_NewSession:
        [Server(pid,sid,otc), In(<pid,nonce,senc(<sid,tc,~pr>,k)
            >),
        !SharedKey(pid,k), !Smaller(otc,tc) ]
         −{ Login(pid,sid,tc,senc(<sid,tc,~pr>,k)),
```

```
            LoginCounter(pid,otc,tc)
          ]↦
        [Server(pid,sid,tc)]

/* The following three axioms function like filters on
 * the traces. They ensure that: */

//a) the !Smaller relation is transitive
axiom transitivity:
    "∀ #t1 #t2 a b c. IsSmaller(a,b)@t1  ∧ IsSmaller(b,c)@t2
    ⟶ ∃ #t3 . IsSmaller(a,c)@t3 "

//b) !Smaller implies inequality
axiom smaller_implies_unequal:
    "¬ ( ∃ a #t . IsSmaller(a,a)@t)"

//c) The protocol runs only after the IsSmaller and
//   IsSuccessor relation is build up
axiom theory_before_protocol:
    "∀ #i #j. Theory() @ i  ∧ Protocol() @ j ⟶ i < j"

// For sanity: Ensure that a successful login is reachable.
lemma Login_reachable:
  exists-trace
  "∃ #i pid sid x otp1. Login(pid,sid,x,otp1)@i"

// Each successful login with counter value x was
// preceded by a button press with the same counter
// value
lemma one_count_foreach_login[reuse,use_induction]:
        "∀ pid sid x otp  #t2 . Login(pid,sid,x,otp)@t2 ⟶
         ( ∃ #t1  . YubiPress(pid,x)@#t1  ∧ #t1<#t2 )"

// If a successful Login happens before a second
// successful Login, the counter value of the first is
// smaller than the counter value of the second
lemma slightly_weaker_invariant[reuse, use_induction]:
        "( ∀ pid otc1 tc1 otc2 tc2 #t1 #t2 .
             LoginCounter(pid,otc1,tc1)@#t1 &
             LoginCounter(pid,otc2,tc2)@#t2
        ⟶ ( #t1<#t2  ∧ ( ∃ #t3 . IsSmaller(tc1,tc2)@t3 ))
             ∨ #t2<#t1  ∨ #t1=#t2)
        "
induction
  case empty_trace
  by contradiction // from formulas
next
  case non_empty_trace
  simplify
  solve( (∀ pid otc1 tc1 otc2 tc2 #t1 #t2.
            (LoginCounter( pid, otc1, tc1 ) @ #t1) ∧
            (LoginCounter( pid, otc2, tc2 ) @ #t2)
```

```
        ⇒
         (last(#t2)) ∨
         (last(#t1)) ∨
         ((#t1 < #t2) ∧
          (∀ #t3. (IsSmaller( tc1, tc2 ) @ #t3) ∧ ¬(last(#t3)))
              ) ∨
         (#t2 < #t1) ∨
         (#t1 = #t2))  ∨
        (∀ #t1 #t2 a b c.
         (IsSmaller( a, b ) @ #t1) ∧ (IsSmaller( b, c ) @ #t2)
        ∧
         (¬(last(#t2))) ∧
         (¬(last(#t1))) ∧
         (∀ #t3. (IsSmaller( a, c ) @ #t3) ⇒ last(#t3))) )
  case case_1
  solve( (last(#t2))  ∨ (last(#t1))  ∨
         ((#t1 < #t2) ∧
          (∀ #t3. (IsSmaller( tc1, tc2 ) @ #t3) ∧ ¬(last(#t3)))
              )  ∨
         (#t2 < #t1)  ∨ (#t1 = #t2) )
    case case_1
    solve( Server( pid, sid, otc1 ) ▶₀ #t1 )
          case BuyANewYubikey
          solve( Server( ~pid, sid.1, otc2 ) ▶₀ #t2 )
            by sorry
      next
          case Server_ReceiveOTP_NewSession_case_1
          solve( Server( ~pid, sid.1, otc2 ) ▶₀ #t2 )
            by sorry
      next
          case Server_ReceiveOTP_NewSession_case_2
          solve( Server( ~pid, sid.1, otc2 ) ▶₀ #t2 )
            by sorry
      next
          case Server_ReceiveOTP_NewSession_case_3
          solve( Server( ~pid, sid.1, otc2 ) ▶₀ #t2 )
            by sorry
      next
          case Server_ReceiveOTP_NewSession_case_4
          solve( Server( ~pid, sid.1, otc2 ) ▶₀ #t2 )
            by sorry
      qed
  next
    case case_2
    by contradiction // cyclic
  next
    case case_3
    by contradiction // from formulas
  next
    case case_4
    by contradiction // from formulas
  next
```

```
      case case_5
      by contradiction // from formulas
    qed
  next
    case case_2
    by sorry
  qed
qed

lemma no_replay:
    "¬ ( ∃ #i #j pid sid x otp1 otp2 .
     Login(pid,sid,x,otp1)@i  ∧  Login(pid,sid,x,otp2)@j
      ∧ not(#i=#j))"

lemma injective_correspondance:
    "∀ pid sid x otp  #t2 . Login(pid,sid,x,otp)@t2 ⟶
         ( ∃ #t1  . YubiPress(pid,x)@#t1  ∧ #t1<#t2
        ∧ ∀ otp2 #t3 . Login(pid,sid,x,otp2)@t3 ⟶ #t3=#t2
        )"

lemma Login_invalidates_smaller_counters:
    "∀ pid otc1 tc1 otc2 tc2 #t1 #t2 #t3 .
         LoginCounter(pid,otc1,tc1)@#t1 &
         LoginCounter(pid,otc2,tc2)@#t2
          ∧ IsSmaller(tc1,tc2)@t3
        ⟶  #t1<#t2 "
end
```

Listing 20: Modelling of the Yubikey authentication protocol in conjunction with the YubiHSM, modelling the "smaller-than" relation as described in Listing 19.

```
theory YubikeyHSMnoms
begin

section{* The Yubikey-Protocol with a YubiHSM (alternative
    modelling) *}

builtins: symmetric-encryption

functions: S/1,zero/0

functions: keystream/2,  keystream_kh/1, keystream_n/1,
           xor/2,
           mac/2, demac/2

equations: keystream_kh(keystream(kh,n))=kh,
           keystream_n(keystream(n,n))=n,
                       xor(xor(a,b),a)=b,
                       xor(xor(a,b),b)=a,
                       xor(a,a)=zero,
                       xor(zero,a)=a,
```

```
                        xor(a,zero)=a,
          demac(mac(m,k),k)=m

// Rules modelling the "successor of" relation
rule InitSucc:
    [In(zero),In(S(zero))]
     -[Theory(), IsSucc(zero,S(zero)),IsZero(zero)]->
        [!Succ(zero,S(zero))]

rule StepSucc:
    [In(y),In(S(y)), !Succ(x,y)]
        -[Theory(), IsSucc(y,S(y)) ]->
        [!Succ(y,S(y))]

// Rules modelling the "smaller than" relation
rule SimpleSmaller:
    [!Succ(x,y)]
        -[Theory(), IsSmaller(x,y)]->
        [!Smaller(x,y)]

rule ZExtendedSmaller:
    [!Smaller(x,y),!Succ(y,z)]
        -[Theory(), IsSmaller(x,z)]->
        [!Smaller(x,z)]

// Intruder's control over Server

/* The attacker can send messages to the HSM, i.e., on
 * behalf of the authentication server. Likewise, he
 * can receive messages.  */
rule isendHSM:
   [ In( x ) ] -[ HSMWrite(x) ]-> [ InHSM( x ) ]
rule irecvHSM:
   [ OutHSM( x ) ] -[ HSMRead(x) ]-> [Out(x)]

// The attacker can write and read the Authentication
// Server's database.
rule read_AEAD:
    [ !S_AEAD(pid,aead)  ] -[ AEADRead(aead),HSMRead(aead) ]-> [
        Out(aead)]
rule write_AEAD:
    [ In(aead), In(pid) ] -[ AEADWrite(aead),HSMWrite(aead) ]->
        [!S_AEAD(pid,aead) ]

/* Initialisation of HSM and Authentication Server.
 * OneTime() Assures that this can only happen a single
 * time in a trace */
rule HSMInit:
    [Fr(~k), Fr(~kh)]
    -[ Protocol(), GenerateRole1(~k),
        MasterKey(~k), OneTime()]->
    [ !HSM(~kh,~k), Out(~kh),
```

```
    //!YSM_AEAD_GENERATE(~kh), //uncomment to produce attack
      !YSM_AEAD_YUBIKEY_OTP_DECODE(~kh)
    ]

//Some commands on the HSM:
rule YSM_AEAD_RANDOM_GENERATE:
    let ks=keystream(kh,N)
        aead=<xor(senc(ks,k),~data),mac(~data,k)>
    in
    [ Fr(~data), InHSM(<N,kh>),
      !HSM(kh,k),!YSM_AEAD_RANDOM_GENERATE(kh)]
    -[GenerateRandomAEAD(~data)]->
    [OutHSM( aead)
    ]

rule YSM_AEAD_GENERATE:
    let ks=keystream(kh,N)
        aead=<xor(senc(ks,k),data),mac(data,k)>
    in
    [InHSM(<N,kh,data>),!HSM(kh,k),!YSM_AEAD_GENERATE(kh)]
    -[GenerateAEAD(data,aead )]->
    [OutHSM( aead) ]

rule YSM_AES_ESC_BLOCK_ENCRYPT:
    [InHSM(<kh,data>), !HSM(kh,k), !YSM_AES_ESC_BLOCK_ENCRYPT(kh)
        ]
    -[ ]->
    [OutHSM(senc(data,k))]

rule YSM_AEAD_YUBIKEY_OTP_DECODE:
    let ks=keystream(kh,N)
        aead=<xor(senc(ks,k),<k2,did>),mac(<k2,did>,k)>
        otp=senc(<did,sc,rand>,k2)
    in
    [ InHSM(<did,kh,aead,otp>), !HSM(kh,k),
      !YSM_AEAD_YUBIKEY_OTP_DECODE(kh)
    ]
    -[
        OtpDecode(k2,k,<did,sc,rand>,sc,
                  xor(senc(ks,k),<k2,did>),mac(<k2,did>,k)),
        OtpDecodeMaster(k2,k)
    ]->
    [ OutHSM(sc) ]

//Yubikey operations
rule BuyANewYubikey:
    let ks=keystream(kh,~pid)
        aead=<xor(senc(ks,~k),<~k2,~sid>),mac(<~k2,~sid>,~k)>
    in
/* This rule implicitly uses YSM_AEAD_GENERATE to
 * produce the AEAD that stores the secret identity and
 * shared key of a Yubikey. By disabling the
```

```
 * YSM_AEAD_GENERATE flag but nevertheless permitting
 * this operation, we model a scenario where
 * YSM_AEAD_GENERATE can be safely used to guarantee
 * the operation, but ¬by the attacker. This
 * corresponds to a setup where Yubikey initialisation
 * takes place on a different server, or a setup where
 * the initialisation takes place before the server is
 * plugged into the network.  Uncomment the following
 * line to require the HSM to have the
 * YSM_AEAD_GENERATE flag set.
 */
    [
    //!YSM_AEAD_GENERATE(kh),
    Fr(~k2), Fr(~pid), Fr(~sid), !HSM(kh,~k), !Succ(zero,one) ]
     −[ Init(~pid,~k2) ]→
    [ Y_counter(~pid,one), !Y_Key(~pid,~k2), !Y_sid(~pid,~sid),
      S_Counter(~pid,zero), !S_AEAD(~pid,aead),
      !S_sid(~pid,~sid), Out(~pid) ]

//On plugin, the session counter is increased and the token
    counter reset
rule Yubikey_Plugin:
    [Y_counter(pid,sc),!Smaller(sc, Ssc) ]
    −[ Yubi(pid,Ssc) ]→
    [Y_counter(pid, Ssc)]

rule Yubikey_PressButton:
    [Y_counter(pid,tc),!Y_Key(pid,k2),!Y_sid(pid,sid),
     !Succ(tc,Stc),Fr(~pr),Fr(~nonce) ]
    −[ YubiPress(pid,tc),
       YubiPressOtp(pid,<sid,tc,~pr>,tc,k2) ]→
    [Y_counter(pid,Stc), Out(<pid,~nonce,senc(<sid,tc,~pr>,k2)>)]

rule Server_ReceiveOTP_NewSession:
    let ks=keystream(kh,pid)
        aead=<xor(senc(ks,k),<k2,sid>),mac(<k2,sid>,k)>
    in
    [ In(<pid,nonce,senc(<sid,tc,~pr>,k2)>),
      !HSM(kh,k), !S_AEAD(pid,aead), S_Counter(pid,otc),
      !S_sid(pid,sid), !Smaller(otc,tc) ]
     −[ Login(pid,sid,tc,senc(<sid,tc,~pr>,k2)),
        LoginCounter(pid,otc,tc) ]→
    [ S_Counter(pid,tc) ]

/* The following three axioms function like filters on
 * the traces. They ensure that: */

//a) the !Smaller relation is transitive
axiom transitivity: //axiomatic
    "∀ #t1 #t2 a b c. IsSmaller(a,b)@t1  ∧ IsSmaller(b,c)@t2
    ⟶ ∃ #t3 . IsSmaller(a,c)@t3 "

```

```
//b) !Smaller implies unequality
axiom smaller_implies_unequal: //axiomatic
    "¬(∃ a #t . IsSmaller(a,a)@t)"

/*c) The protocol runs only after the IsSmaller and IsSuccessor
    relation is
 *   build up */
axiom theory_before_protocol:
    "∀ #i #j. Theory() @ i ∧ Protocol() @ j ⟶ i < j"

axiom onetime:
    "∀ #t3 #t4 . OneTime()@#t3 ∧ OneTime()@t4 ⟶ #t3=#t4"

//LEMMAS:

// For sanity: Ensure that a successful login is reachable.
/* lemma Login_reachable: */
/*   exists-trace "∀ #i pid sid x otp1. Login( pid, sid, x, otp1
    ) @ #i" */
/* simplify */
/* solve( !KU( senc (<sid, (otc+z), ~pr>, k2) ) @ #vk.4 ) */
/*   case Yubikey_PressButton */
/*   solve( !S_sid( pid, ~sid ) ▶4 #i ) */
/*     case BuyANewYubikey */
/*     solve( S_Counter( ~pid, otc ) ▶3 #i ) */
/*       case BuyANewYubikey */
/*       solve( !S_AEAD( ~pid, */
/*                       <xor(senc(keystream(kh, ~pid), k), <~k2,
    ~sid>), */
/*                        mac(<~k2, ~sid>, k)> */
/*             ) ▶2 #i ) */
/*         case BuyANewYubikey */
/*         solve( !HSM( ~kh, ~k ) ▶1 #i ) */
/*           case HSMInit */
/*           solve( Y_counter( ~pid, ('1'+z) ) ▶0 #vr ) */
/*             case Yubikey_Plugin_case_1 */
/*             solve( Y_counter( ~pid, '1' ) ▶0 #vr.5 ) */
/*               case BuyANewYubikey */
/*               solve( !KU( ~pid ) @ #vk.3 ) */
/*                 case BuyANewYubikey */
/*                 SOLVED // trace found */
/*               qed */
/*             qed */
/*           qed */
/*         qed */
/*       next */
/*         case write_AEAD */
/*         by sorry */
/*       qed */
/*     next */
/*       case Server_ReceiveOTP_NewSession_case_1 */
/*       by sorry */
```

```
/*     next */
/*       case Server_ReceiveOTP_NewSession_case_2 */
/*       by sorry */
/*     qed */
/*   qed */
/* next */
/*   case csenc */
/*   by sorry */
/* next */
/*   case irecvHSM */
/*   by sorry */
/* qed */

/* Every counter produced by a Yubikey could be
 * computed by the adversary anyway. (This saves a lot
 * of steps in the backwards induction of the following
 * lemmas).    */
lemma adv_can_guess_counter[reuse,use_induction]:
    "∀pid sc #t2 . YubiPress(pid,sc)@t2
    ⟶ (∃#t1 . K(sc)@#t1  ∧ #t1<#t2)"

/* Everything that can be learned by using OtpDecode is
 * the counter of a Yubikey, which can be computed
 * according to the previous lemma.  */
lemma otp_decode_does_not_help_adv_use_induction
    [reuse,use_induction]:
    "∀#t3 k2 k m sc enc mac . OtpDecode(k2,k,m,sc,enc,mac)@t3
    ⟶ ∃#t1 pid . YubiPress(pid,sc)@#t1  ∧ #t1<#t3"

/* ∀keys shared between the YubiHSM and the
 * Authentication Server are either ¬known to the
 * adversary, or the adversary learned the key used to
 * encrypt said keys in form of AEADs.  */
lemma k2_is_secret_use_induction[use_induction,reuse]:
    "∀#t1 #t2 pid k2 . Init(pid,k2)@t1  ∧ K(k2)@t2
    ⟶
     (∃#t3 #t4 k . K(k)@t3  ∧ MasterKey(k)@t4  ∧ #t3<#t2)"

/* Neither of those kinds of keys are ever learned by
 * the adversary */
lemma neither_k_nor_k2_are_ever_leaked_inv[use_induction,reuse]:
    "
not( ∃#t1 #t2 k . MasterKey(k)@t1  ∧ KU(k)@t2 )
 ∧ ¬(∃  #t5 #t6 k6 pid . Init(pid,k6)@t5  ∧ KU(k6)@t6 )
    "

// Each successful login with counter value x was
// preceded by a button press with the same counter
// value
lemma one_count_foreach_login[reuse,use_induction]:
        "∀pid sid x otp  #t2 . Login(pid,sid,x,otp)@t2 ⟶
         ( ∃#t1  . YubiPress(pid,x)@#t1  ∧ #t1<#t2 )"
```

```
induction
  case empty_trace
  by contradiction // from formulas
next
  case non_empty_trace
  simplify
  solve( !S_AEAD( pid,
                  <xor(senc(keystream(kh, pid), k), <k2, sid>),
                      mac(<k2, sid>, k)>
        ) ▶2 #t2 )
    case BuyANewYubikey_case_1
    by sorry
  next
    case BuyANewYubikey_case_2
    by sorry
  next
    case write_AEAD
    solve( !KU( xor(senc(keystream(kh, pid), k), <k2, sid>)
          ) @ #vk.6 )
      case irecv
      by contradiction // cyclic
    next
      case cxor
      by sorry
    next
      case read_AEAD_case_1
      by sorry
    next
      case read_AEAD_case_2
      by sorry
    qed
  qed
qed

end
```

A.3 LISTINGS FOR CHAPTER 5

Listing 21: The running example from Chapter 5.

```
theory RunningExample

begin

builtins: symmetric-encryption

functions: encSucc/2, true/0
equations: encSucc(senc(x,y),y) = true

!(
        (  ν h; ν k; event NewKey(h,k);
```

```
                insert <'key',h>,k;
                insert <'att',h>, 'dec'; out(h) )
  | //allow wrap
        (  in(h);  insert <'att',h>, 'wrap' )
  | //Dec
        ( in(<h,c>);
          lookup <'att',h> as a in
                        if a='dec' then lookup <'key',h> as k in
                                if encSucc(c,k)=true() then
                                event DecUsing(k,sdec(c,k)); out(
                                    sdec(c,k))
        )
  | //Wrap
        ( in(<h1,h2>); lookup <'att',h1> as a1 in
                        if a1='wrap' then lookup <'key',h1> as k1
                             in
                        lookup <'key',h2> as k2 in
                                event Wrap(k1,k2);
                                out(senc(k2,k1))
        )
)

lemma can_create_key: //for sanity
        exists-trace
        "∃ #t h k. NewKey(h,k)@t"

lemma can_obtain_wrapping: //for sanity
        exists-trace
        "∃ #t k1 k2. Wrap(k1,k2)@t"

lemma dec_limits[reuse,typing]:
        //a message that can be decrypted was
        //either encrypted on the device, or some key leaked, or
        // ¬a valid enc
        "∀ k m #t1. DecUsing(k,m)@t1 ⟹
        ( ∃h k2 #t2 #t3. NewKey(h, k2)@t2 ∧ KU(k2)@t3 ∧ t2<t1
            ∧ t3<t1)
        "


lemma cannot_obtain_key_ind[reuse,use_induction]:
        "¬(∃ #i #j  h k . NewKey(h,k)@i ∧ KU(k) @j)"

lemma cannot_obtain_key:
        "¬(∃ #i #j  h k. NewKey(h,k)@i ∧ K(k) @j)"

end
```

Listing 22: Security à la PKCS#11.

```
theory EncWrapDecUnwrap

begin
```

```
builtins: symmetric-encryption

!(
    ( in('create'); ν h; ν k; event NewKey(h,k);
      insert <'key',h>,k;
      insert <'att',h>, 'init'; out(h) )
  |
    ( in(<'set_wrap',h>); lock <'att',h>; lookup <'att',h> as a
      in
      if a='init' then delete <'att',h>;
          insert <'att',h>, 'wrap';
          event WrapHandle(h); unlock <'att',h>
    )
  |
    ( in(<'set_dec',h>); lock <'att',h>; lookup <'att',h> as a in
      if a='init' then delete <'att',h>; insert <'att',h>, 'dec
          ';
      unlock <'att',h>
    )
  | //Dec
    ( in(<h,senc(m,k)>); lookup <'att',h> as a in
            if a='dec' then
                lookup <'key',h> as kp in
                    if kp=k then
                event DecUsing(k,m); out(m)
    )
  | //Enc
    ( in(<h,m>); lookup <'att',h> as a in
            if a='dec' then lookup <'key',h> as k in
                event EncUsing(k,m); out(senc(m,k))
    )
  | //Wrap
    ( in(<h1,h2>); lookup <'att',h1> as a1 in
            if a1='wrap' then lookup <'key',h1> as k1 in
            lookup <'key',h2> as k2 in
                event Wrap(k1,k2);
                out(senc(k2,k1))
    )
  | //Unwrap
    ( in(<h1,senc(m,k)>); lookup <'att',h1> as a1 in
            if a1='wrap' then lookup <'key',h1> as k1 in
                if k1=k then
                    ν h2;
                    insert <'key',h2>, m;
                    insert <'attr',h2>, 'wrap'
    )
)
lemma can_create_key: //for sanity
    exists-trace
    "∃ #t h k. NewKey(h,k)@t"
```

```
lemma can_set_wrap: //for sanity
    exists-trace
    "∃ #t h .WrapHandle(h)@t"

lemma can_obtain_wrapping: //for sanity
    exists-trace
    "∃ #t k1 k2. Wrap(k1,k2)@t"

lemma dec_limits[reuse,typing]:
/* a message that can be decrypted was either encrypted on the
    device, or some
 * key leaked */
    "∀ k m #t1. DecUsing(k,m)@t1 ⟹
    ( ∃h k2 #t2 #t3. NewKey(h, k2)@t2 ∧ KU(k2)@t3 ∧ t2<t1 ∧
        t3<t1)
     ∨ ∃ #t2 . EncUsing(k,m)@t2 ∧ t2<t1 "

lemma cannot_obtain_key_ind[reuse,use_induction]:
    "¬(∃ #i #j h k . NewKey(h,k)@i ∧ KU(k) @j)"

lemma cannot_obtain_key:
    "¬(∃ #i #j h k. NewKey(h,k)@i ∧ K(k) @j)"

end
```

Listing 23: Needham-Schoeder-Lowe protocol.

```
theory NeedhamSchroeder

begin

builtins: asymmetric-encryption

! ( ν skA;
        event HonestA(pk(skA));
        out(pk(skA));
        !( in(pk(xB));
        ν Na;
        event OUT_I_1(aenc(<Na,pk(skA)>,pk(xB)));
        out(aenc( <Na,pk(skA) > ,pk(xB)));
        in(aenc(<Na,xNb,pk(xB)>,pk(skA)));
        event IN_I_2_nr(xNb,aenc(<Na,xNb,pk(xB)>,pk(skA)));
        ν k; out(aenc(<xNb,k>,pk(xB)));
        event SessionA(pk(skA),pk(xB),k)
        ) )
|
! ( ν skB;
        event HonestB(pk(skB));
        out(pk(skB));
        !( in(aenc(<xNa,pk(xA)>,pk(skB)));
           event IN_R_1_ni(xNa,aenc(<xNa,pk(xA)>,pk(skB)));
           ν Nb;
```

```
            event OUT_R_1(aenc(<xNa,Nb,pk(skB)>,pk(xA)));
            out(aenc(<xNa,Nb,pk(skB)>,pk(xA)));
            in(aenc(<Nb,xk>,pk(skB)));
            event SessionB(pk(xA),pk(skB),xk)
        ))

lemma sanity1: //make sure that a valid protocol run exists
exists-trace
        "∃pka pkb k #t1 . SessionA(pka,pkb,k)@t1"

lemma sanity2:
exists-trace
        "∃pka pkb k #t1 . SessionB(pka,pkb,k)@t1"

lemma types [typing, reuse]:
  " (∀ni m1 #i.
       IN_R_1_ni( ni, m1) @ i
       ⟹
       ( (∃#j. KU(ni) @ j ∧ j < i)
        ∨ (∃#j. OUT_I_1( m1 ) @ j)
       )
    )
   ∧ (∀nr m2 #i.
       IN_I_2_nr( nr, m2) @ i
       ⟹
       ( (∃#j. KU(nr) @ j ∧ j < i)
        ∨ (∃#j. OUT_R_1( m2 ) @ j)
       )
    )
"

lemma secrecy:
        "¬(
                ∃pka pkb k #t1 #t2 .
                 SessionA(pka,pkb,k)@t1
                 ∧ K(k)@t2
                 ∧ ( ∃#i . HonestA(pka)@i)
                 ∧ ( ∃#i . HonestB(pkb)@i)
        )"

end
```

Listing 24: Yubikey protocol.

```
theory Yubikey
begin

section{* The Yubikey-Protocol *}

builtins: symmetric-encryption, multiset

let Yubikey=
```

```
 ν k; ν pid; ν secretid;
 insert <'Server',pid>, <secretid,k,'one'>;
 insert <'Yubikey',pid>,'one';
 event ExtendedInit(pid,secretid,k);
 out(pid);
 !(( //Plug
        lock pid;
          lookup <'Yubikey',pid> as sc in
                in(sc); //just a trick to enforce adv learning sc
                insert <'Yubikey',pid>, sc+'one';
                event Yubi(pid,sc + 'one');
        unlock pid
  )|( //ButtonPress
        lock pid;
          lookup <'Yubikey',pid> as tc in
                in(tc); //just a trick to enforce adv learning tc
                insert <'Yubikey',pid>, tc + 'one';
                ν nonce;
                ν npr;
                event YubiPress(pid,secretid,k,tc);
                out(<pid,nonce,senc(<secretid,tc,npr>,k)>);
        unlock pid
  )
 )

let Server=
 !(
         in(<pid,nonce,senc(<secretid,tc,npr>,k)>);
         lock pid;
         lookup <'Server',pid> as tuple in
                if fst(tuple)=secretid then
                        if fst(snd(tuple))=k then
                                in(otc);
                                if snd(snd(tuple))=otc then
                                        event Smaller(otc,tc);
                                        event InitStuff(pid,
                                            secretid,k,tuple,otc,
                                            tc);
                                        event Login(pid,k,tc);
                insert <'Server',pid>, <secretid,k,tc>;
         unlock pid
 )

(Server | !Yubikey)

// we model the larger relation using the smaller action
// and excluding all traces where it is ¬correct
axiom smaller:
        "∀#i a b. Smaller(a,b)@i ⟹ ∃z. a+z=b"

// For sanity: Ensure that a successful login is reachable.
lemma Login_reachable:
```

```
  exists-trace
  "∃ #i pid k x . Login(pid,k,x)@i"

// typing lemmas:
// There exists a Initialisation for every Login on the Server
lemma init_server[typing]:
                "∀ pid sid k tuple otc tc #i . InitStuff(pid,sid,
                    k,tuple,otc,tc)@i
                ⟹
                 tuple=<sid,k,otc>
                &
                  ∃ #j.  ExtendedInit(pid, sid, k)@j ∧ #j<#i
        "

lemma init_yubikey[typing]:
                "∀ pid sid k tc #i . YubiPress(pid,sid,k,tc)@i
                    ⟹ ∃ #j.
                ExtendedInit(pid, sid, k)@j ∧ #j<#i"

// If a succesful Login happens before a second sucesfull Login,
// the counter value of the first is smaller than the counter
// value of the second
lemma slightly_weaker_invariant[reuse, use_induction]:
        "(∀ pid k tc1 tc2 #t1 #t2 .
             Login(pid,k,tc1)@#t1 ∧ Login(pid,k,tc2)@#t2
        ⟹ ( #t1<#t2 ∧ ( ∃ z . tc1+z=tc2))
             ∨ #t2<#t1 ∨ #t1=#t2)
        "

/* It is impossible to have to distinct logins with the same
 * counter  value */
lemma no_replay[reuse]:
        "¬(∃ #i #j pid k x .
         Login(pid,k,x)@i ∧ Login(pid,k,x)@j
          ∧ not(#i=#j))"

/*  Each succesful login with counter value x was preceeded by a
    PressButton */
/*  event with the same counter value */
/*  proof needs to be guided */
lemma one_count_foreach_login[reuse,use_induction]:
        "∀ pid k x #t2 . Login(pid,k,x)@t2 ⟹
         ( ∃ #t1 sid . YubiPress(pid,sid,k,x)@#t1 ∧ #t1<#t2
                )"


lemma injective_correspondance[reuse, use_induction]:
    "∀ pid k x #t2 . Login(pid,k,x)@t2 ⟹
         ( ∃ #t1 k . YubiPress(pid,k,k,x)@#t1 ∧ #t1<#t2 )
        ∧ ∀ #t3 . Login(pid,k,x)@t3 ⟹ #t3=#t2
        "
```

```
lemma Login_invalidates_smaller_counters:
        " ∀ pid k tc1 tc2 #t1 #t2 #t3 .
             Login(pid,k,tc1)@#t1  ∧ Login(pid,k,tc2)@#t2
                                ∧ Smaller(tc1,tc2)@t3
                ⟹  #t1<#t2 "
end
```

Listing 25: Garay, Jakobsson and MacKenzie's optimistic contract signing protocol.

```
theory Contract
begin

section{* GM Protocol for Contract signing *}

builtins: signing

functions: pcs/3, checkpcs/5, convertpcs/2, check_getmsg/2,
    fakepcs/4

equations:
check_getmsg(sign(xm, xsk), pk(xsk)) = xm,
checkpcs(xc, pk(xsk), ypk, zpk, pcs(sign(xc, xsk), ypk, zpk)) =
    true(),
convertpcs(zsk, pcs(sign(xc, xsk), ypk, pk(zsk))) = sign(xc, xsk)
    ,
/* fakepcs () */
checkpcs(xc, xpk, pk(ysk), zpk, fakepcs(xpk, ysk, zpk, xc)) =
    true()

let Abort1 =
  in(<'abort',<ct,pk1,pk2,ysig>>);
    if check_getmsg(ysig, pk1) = <ct,pk1,pk2> then
      (lock  ct;
       lookup ct as state in unlock ct/* TODO: maybe output state
           */
       else
        (
              insert ct, 'aborted';
              event Abort1(ct);
              unlock ct;
              out(sign(<<ct,pk1,pk2>,ysig>,skT))
        )
      )

let Resolve2 =
in(<'resolve2',<ct,pk1,pk2,ypcs1,ysig2>>);
if check_getmsg(ysig2, pk2)=ct then
(
    /* check validity of the pcs ..something ¬done in StatVerif/
        Tamarin modelling */
    if check_getmsg(convertpcs(skT,ypcs1),pk1) = ct then
```

```
    (
        if checkpcs(ct, pk1, pk2, pk(skT), ypcs1)=true then
        (
            lock ct;
            lookup ct as status in unlock ct
            else
            (
                insert ct, 'resolved2';
                event Resolve2(ct);
                unlock ct;
                out(sign(<convertpcs(skT,ypcs1), ysig2>,skT))
            )
        )
    )
)

let Resolve1 =
in(<'resolve1',<ct,pk1,pk2,ysig1,ypcs2>>);
if check_getmsg(ysig1,pk1)=ct then
(
    if check_getmsg(convertpcs(skT,ypcs2),pk2) = ct then
    (
        if checkpcs(ct, pk2, pk1, pk(skT), ypcs2)=true then
        (
            lock ct;
            lookup ct as status in unlock ct
            else
            (
                insert ct, 'resolved1';
                event Resolve1(ct);
                unlock ct;
                out(sign(<ysig1,convertpcs(skT, ypcs2)>,skT))
            )
        )
    )
)

let WitnessAbort =
 in (sign(pcs(sign(ct,sk1), pk(ysk), pk(skT)),skT) );
    event AbortCert(ct)
let WitnessResolved =
 in (sign(<sign(ct,sk1), sign(ct,sk2)>,skT));
  event ResolveCert(ct)

let HonestClient =
    ν skA;
    out(pk(skA));
    in (<ct,xpkB>);
    ( //First decision: Sign the contract!
        out(sign(ct,skA));
        (
            in(sigB);
```

```
            if verify(sigB,ct,xpkB)=true() then
                //if we get a signature back..good!
                event AhasSignature(ct)
            else //if not
                0 //Accept silently
        )
    )
    | //Or we decide ¬to sign it
    0



!(
ν skT;
event TrustedParty(skT);
out(pk(skT));
      (
! Abort1 | ! Resolve2 | ! Resolve1  |
!WitnessAbort | !WitnessResolved |
!HonestClient
)
)

/* Lemmas taken from tamarin files */

lemma aborted_and_resolved_exclusive:
  "¬(∃ ct #i #j. AbortCert(ct) @ i  ∧ ResolveCert(ct) @ j)"

/* Sanity checks */
lemma resolved1_contract_reachable:
  exists-trace
  " (∃ ct #i. ResolveCert(ct) @ i)
    // Ensure that this is possible with at most one Resolve1
        step.
   ∧ (∀ ct #i. Abort1(ct) @ i ⟹ F)
   ∧ (∀ ct1 ct2 #i1 #i2 .
        Resolve1(ct1) @ i1  ∧ Resolve1(ct2) @ i2 ⟹ #i1 = #i2)
   ∧ (∀ ct #i. Resolve2(ct) @ i ⟹ F)
  "

lemma resolved2_contract_reachable:
  exists-trace
  "(∃ ct #i. ResolveCert(ct) @ i)
    // Ensure that this is possible with at most one Resolve1
        step.
   ∧ (∀ ct #i. Abort1(ct) @ i ⟹ F)
   ∧ (∀ ct #i. Resolve1(ct) @ i ⟹ F)
   ∧ (∀ ct1 ct2 #i1 #i2 .
        Resolve2(ct1) @ i1  ∧ Resolve2(ct2) @ i2 ⟹ #i1 = #i2)
  "
end
```

Listing 26: Key-server example from [74].

```
theory SetAbst
begin

section{* The PKI-example *}

builtins: asymmetric-encryption, signing

let Client=
    (
    // Revoke key
    ν ~nsk;
    [ ClientKey(user, ~sk) ] −[ HonestKey(~nsk) ]→ [ ClientKey(
        user,~nsk) ];
    out(<'revoke',user,pk(~nsk)>);
    out(sign(<'revoke',user,pk(~nsk)>,~sk));
    in(sign(<'confirm',sign(<'revoke',user,pk(~nsk)>,~sk)>,pki));
    event Revoked(~sk);
    out(~sk)
    )

let Server=
(( in(<'create',user>); //Create Honest Keys
    ν ~sk;
    [ ] −[ HonestKey(~sk) ]→ [ ServerDB(pki,user,pk(~sk)),
ClientKey(user,~sk)];
    out(pk(~sk))
)
| //Allow creating Dishonest Keys
( in(<user,sk>);
  [] −→ [ServerDB(pki,user,pk(sk))]
)
| //Revoke Key
(
    in(<'revoke',user,pk(nsk)>);
    in(sign(<'revoke',user,pk(nsk)>,sk));
    [ServerDB(pki,user,pk(sk))]−→[ServerDB(pki,user,pk(nsk))];
    out(sign(<'confirm',sign(<'revoke',user,pk(nsk)>,sk)>,pki))
))

!(ν pki; ! Server | (ν user; ! Client) )

rule SetupDishonestKey:
    [ In(sk) ] −→ [ ServerDB($A, pk(sk)) ]

lemma Knows_Honest_Key_imp_Revoked:
  " ∀ sk #i #d. HonestKey(sk) @ i ∧ K(sk) @ d ⟹
      ( ∃ #r. Revoked(sk) @ r)
  "

end
```

Listing 27: Key-server example from [74], modelled under the use of insert and lookup.

```
theory SetAbst
begin

section{* The PKI-example *}

builtins: asymmetric-encryption, signing

let Client=
 /* Revoke key */
        ν ~nsk;
        lock user;
        lookup <'USER',user> as ~sk in
                event HonestKey(~nsk);
                delete <'USER',user>;
                insert <'USER',user>,~nsk;
        unlock user;
        out(<'revoke',user,pk(~nsk)>);
        out(sign(<'revoke',user,pk(~nsk)>,~sk));
        in(sign(<'confirm',sign(<'revoke',user,pk(~nsk)>,~sk)>,
            pki));
        event Revoked(~sk);
        out(~sk)

let Server=
(
( /* Create Honest Keys */
 in(<'create',user>);
        ν ~sk;
        lock user;
                event HonestKey(~sk);
                insert <'SERVER',pki,user>, pk(~sk);
                insert <'USER',user>, ~sk;
                event HonestKey(~sk);
        unlock user;
        out(pk(~sk))
)
|
( /* Revoke key */
        in(<'revoke',user,pk(nsk)>);
        in(sign(<'revoke',user,pk(nsk)>,sk));
        lock user;
        lookup <'SERVER',pki,user> as pksk in
                if pksk = pk(sk) then
                        delete <'SERVER',pki,user>;
                        insert <'SERVER',pki,user>, pk(nsk);
                        unlock user;
                        out(sign(<'confirm',sign(<'revoke',user,
                            pk(nsk)>,sk)>,pki))
)
)
```

```
!(ν pki; ! Server | (ν user; ! Client) )

rule SetupDishonestKey:
    [ In(sk) ] −→ [ ServerDB($A, pk(sk)) ]


lemma Knows_Honest_Key_imp_Revoked:
  " ∀ sk #i #d. HonestKey(sk) @ i  ∧ K(sk) @ d ⟹
      ( ∃ #r. Revoked(sk) @ r)
  "


/* Sanity check. Commented out for runtime comparison to [1]. */
lemma Honest_Revoked_Known_Reachable:
  exists-trace
  "( ∃ sk #i #j #r. HonestKey(sk) @ i
                  ∧ K(sk) @ j
                  ∧ Revoked(sk) @ r
   )"

end
```

Listing 28: Left-right example from [5], described in Example 4 on p.22.

```
theory StatVerif_Security_Device begin

builtins: asymmetric-encryption

let Device=(
    out(pk(sk))
    |
    (   lock s ;
        in(req);
        lookup s as status in
            if status='init' then
                insert s,req;
                unlock s
    )
    |
    (
        lock s;
        in(aenc{<x,y>}pk(sk));
        lookup s as status in
            if status='left' then
                event Access(x); out(x); unlock s
            else if status='right' then
                event Access(y); out(y); unlock s
        )
)

let User=
```

```
        ν lm; ν rm; event Exclusive(lm,rm);
        out(aenc{<lm,rm>}pk(sk))

!( ν sk; ν s; insert s,'init'; ( Device | ! User ))

// Typing lemma, taken from Tamarin's example directory:
lemma types [typing]:
  "∀m #i. Access(m) @ i ⟹
      (∃#j. KU(m) @ j ∧ j < i)
    ∨ (∃x #j. Exclusive(x,m) @ j)
    ∨ (∃y #j. Exclusive(m,y) @ j)
  "

// Check that there is some trace where the intruder knows the
// left message of an exclusive message-tuple. In contrast to the
// typing lemma, we use the standard 'K'-fact, which is logged by
// the built-in 'ISend' rule.
lemma reachability_left:
  exists-trace
  "∃x y #i #j. Exclusive(x,y) @i ∧ K(x) @ j"

lemma reachability_right:
  exists-trace
  "∃x y #i #k. Exclusive(x,y) @i ∧ K(y) @ k"

// Check that exclusivity is maintained
lemma secrecy:
  "not(∃x y #i #k1 #k2.
        Exclusive(x,y) @i ∧ K(x) @ k1 ∧ K(y) @ k2
    )
  "

end
```

B

PROOFS FOR PART i

B.1 CORRECTNESS OF TAMARIN'S SOLUTION PROCEDURE FOR TRANSLATED RULES

The multiset rewrite system produced by our translation for a well-formed process P could actually contain rewrite rules that are not valid with respect to Definition 2, because they violate the third condition, which is: For each $l' -[a']\rightarrow r' \in R \in_E \mathit{ginsts}(l -[a]\rightarrow r)$ we have that $\cap_{r''=_E r'} \mathit{names}(r'') \cap \mathit{FN} \subseteq \cap_{l''=_E l'} \mathit{names}(l'') \cap \mathit{FN}$

This does not hold for rules in $[\![P]\!]_{=p}$ where p is the position of the lookup-operator. The right hand-side of this rule can be instantiated such that, assuming the variable bound by the lookup is named v, this variable v is substituted by a names that does not appear on the left-hand side. In the following, we will show that the results from [89] still hold. In practice, this means that the tamarin-prover can be used for verification, despite the fact that it outputs well-formedness errors for each rule that is a translation of a lock.

We will introduce some notation first. We re-define $[\![P]\!]$ to contain the INIT rule and $[\![\overline{P}, [], []]\!]$, but not MD (which is different to Definition 14). We furthermore define a translation with dummy-facts, denoted $[\![P]\!]^D$, that contains INIT and $[\![\overline{P}, [], []]\!]^D$, which is defined as follows:

Definition 32: We define $[\![P]\!]^D := \text{INIT} \cup [\![\overline{P}, [], []]\!]^D$, where $[\![\overline{P}, [], []]\!]^D$ is defined just as $[\![\overline{P}, [], []]\!]$, with the exception of two cases, $P = \text{lookup } M \text{ as } v \text{ in } P \text{ else } Q$ and $P = \text{insert } s, t; P$. In those cases, it is defined as follows:

$$
\begin{aligned}
&[\![\text{lookup } M \text{ as } v \text{ in } P \text{ else } Q, p, \tilde{x}]\!]^D = \\
&\quad \{[\mathsf{state}_p(\tilde{x}), !\mathsf{Dum}(v)] -[\mathsf{IsIn}(M, v)]\rightarrow \quad [\mathsf{state}_{p\cdot 1}(\tilde{M}, v)], \\
&\qquad\qquad [\mathsf{state}_p(\tilde{x})] -[\mathsf{IsNotSet}(M)]\rightarrow \quad [\mathsf{state}_{p\cdot 2}(\tilde{x})]\} \\
&\qquad \cup [\![P, p\cdot\ 1, (\tilde{x}, v)]\!]^D \cup [\![Q, p\cdot 2, \tilde{x}]\!]^D \\
&[\![\text{insert } s, t; P, p, \tilde{x}]\!]^D = [\mathsf{state}_p(\tilde{x})] -[\mathsf{Insert}(s, t)]\rightarrow \\
&\qquad\qquad\qquad\qquad [\mathsf{state}_{p\cdot 1}(\tilde{x}), !\mathsf{Dum}(t)] \cup [\![P, p\cdot 1, \tilde{x}]\!]^D
\end{aligned}
$$

The only difference between $[\![P]\!]$ and $[\![P]\!]^D$ is therefore that $[\![P]\!]^D$ produces a permanent fact !Dum for every value v that appears in an action $\mathsf{insert}(k, v)$, which is a premise to every rule instance with an action $\mathsf{IsIn}(k', v)$. We see that $[\![P]\!]^D$ contains now only valid multiset rewrite rules.

In the following, we would like to show that the tamarin-prover's solution algorithm is correct for $[\![P]\!]$. To this end, we make use of the

proof of correctness of tamarin as presented in Benedikt Schmidt's Ph.D. thesis [88]. We will refer to Lemmas, Theorems and Corollaries in this work by their numbers. We will use the notation of this work, to make it easier to the reader to compare our statements against the statements there. In particular, $\overline{trace(execs(R))}$ is $traces^{msr}(\mathtt{R})$ in our notation. We have to show that:

Lemma 9: For all well-formed process P and guarded trace properties ϕ,

$$trace(execs([\![\mathtt{P}]\!] \cup \mathrm{MD}) \vDash_{\mathcal{DH}_e} \neg\alpha_{in} \vee \phi$$

if and only if

$$trace(ndgraphs([\![\mathtt{P}]\!])) \vDash_{\mathtt{ACC}} \neg\alpha_{in} \vee \phi.$$

Proof. The proof proceeds similar to the proof to Theorem 3.27. We refer to results in [88], whenever their proofs apply despite the fact that the rules in $[\![\mathtt{P}]\!]$ do not satisfy the third condition of multiset rewrite rules.

$$\begin{aligned}
& trace(execs([\![\mathtt{P}]\!] \cup \mathrm{MD}) \vDash_{\mathcal{DH}_e} \neg\alpha_{in} \vee \psi \\
\Leftrightarrow\ & \overline{trace(execs([\![\mathtt{P}]\!] \cup \mathrm{MD})} \vDash_{\mathcal{DH}_e} \neg\alpha_{in} \vee \phi \\
& \qquad\qquad \text{(Lemma 3.7 (unaltered))} \\
\Leftrightarrow\ & \overline{trace(execs([\![\mathtt{P}]\!] \cup \mathrm{MD}))} \downarrow_{\mathcal{RDH}_e} \vDash_{\mathcal{DH}_e} \neg\alpha_{in} \vee \phi \\
& \qquad\qquad \text{(Definition of } \vDash_{\mathcal{DH}_e}) \\
\Leftrightarrow\ & \overline{trace(dgraphs_{\mathcal{DH}_e}([\![\mathtt{P}]\!] \cup \mathrm{MD}))} \downarrow_{\mathcal{RDH}_e} \vDash_{\mathcal{DH}_e} \neg\alpha_{in} \vee \phi \\
& \qquad\qquad \text{(Lemma 3.10 (unaltered))} \\
\Leftrightarrow\ & \overline{trace(\{dg \mid dg \in dgraphs_{\mathcal{ACC}}(\lceil [\![\mathtt{P}]\!] \cup \mathrm{MD} \rceil^{\mathcal{RDH}_e}_{insts})} \\
& \overline{\wedge dg \downarrow_{\mathcal{RDH}_e} \text{-normal}\})} \vDash_{\mathcal{DH}_e} \neg\alpha_{in} \vee \phi \\
& \qquad\qquad \text{(Lemma 3.11 (unaltered))} \\
\Leftrightarrow\ & \overline{trace(ndgraphs([\![\mathtt{P}]\!]))} \vDash_{\mathcal{DH}_e} \neg\alpha_{in} \vee \phi \qquad \text{(Lemma A.12 (*))} \\
\Leftrightarrow\ & trace(ndgraphs([\![\mathtt{P}]\!])) \vDash_{\mathcal{ACC}} \neg\alpha_{in} \vee \phi \\
& \qquad\qquad \text{(Lemma 3.7 and A.20(both unaltered))}
\end{aligned}$$

It is only in Lemma A.12 where the third condition is used: The proof to this lemma applies Lemma A.14, which says that all factors (or their inverses) are known to the adversary. We will quote Lemma A.14 here:

Lemma 10 (Lemma A.14 in [88])*:* For all $ndg \in ndgraphs(\mathtt{P})$, conclusions $(\mathtt{i}, \mathtt{u})$ in ndg with conclusion fact $\mathtt{f}$ and terms $\mathtt{t} \in afactors(\mathtt{f})$, there is a conclusion $(\mathtt{j}, v)$ in ndg with $\mathtt{j} < \mathtt{i}$ and conclusion fact $\mathsf{K}^{\mathtt{d}}(\mathtt{m})$ such that $\mathtt{m} \in_{ACC} \{\mathtt{t}, (\mathtt{t}^{-1}) \downarrow_{\mathcal{RBP}_e}\}$.

If there is $ndg \in ndgraphs([\![P]\!])$, such that $trace(ndg) \vDash_{ACC} \alpha_{in}$, then

$$\begin{aligned} & trace(ndgraphs([\![P]\!])) \vDash_{\mathcal{ACC}} \neg\alpha_{in} \vee \phi \\ \Leftrightarrow & \forall ndg \in ndgraphs([\![P]\!]) \text{ s.t. } trace(ndg) \vDash_{ACC} \alpha_{in} \\ & \quad trace(ndg) \vDash_{\mathcal{ACC}} \vDash \phi \end{aligned}$$

Since for the empty trace, $[] \vDash_{ACC} \alpha_{in}$, we only have to show that Lemma A.14 holds for $ndg \in ndgraphs([\![P]\!])$, such that $trace(ndg) \vDash_{ACC} \alpha_{in}$.

For every $ndg \in ndgraphs([\![P]\!])$, such that $trace(ndg) \vDash_{ACC} \alpha_{in}$, there is a trace equivalent $ndg' \in ndgraphs([\![P]\!]^D)$, since the only difference between $[\![P]\!]$ and $[\![P]\!]^D$ lies in the dummy conclusion and premises, and α_{in} requires that any v in an action $\text{IsIn}(u, v)$ appeared previously in an action $\text{Insert}(u, v)$ (equivalence modulo *ACC*). Therefore, ndg' has the same K^d-conclusions ndg has, and every conclusion in ndg is a conclusion in ndg'.

We have that Lemma A.14 holds for $[\![P]\!]^D$, since all rules generated in this translation are valid multiset rewrite rules. Therefore, Lemma A.14 holds for all $ndg \in ndgraphs([\![P]\!])$, such that $trace(ndg) \vDash_{ACC} \alpha_{in}$, too, concluding the proof by showing the marked (*) step.

□

B.2 PROOFS FOR SECTION 5.4

The following two lemmas are used in the proof to Lemma 2. In order to prove the first one, Lemma 1, we need a few additional lemmas.

We say that a set of traces *Tr* is prefix closed if for all $tr \in Tr$ and for all tr' which is a prefix of tr we have that $tr' \in Tr$.

Lemma 11 (*filter* is prefix-closed)*:* Let *Tr* be a set of traces. If *Tr* is prefix closed then $filter(Tr)$ is prefix closed as well.

Proof. It is sufficient to show that for any trace $tr = tr' \cdot a$ we have that if $\forall\theta.\, (tr, \theta) \vDash \alpha$ then $\forall\theta.\, (tr', \theta) \vDash \alpha$. This can be shown by inspecting each of the conjuncts of α. □

We next show that the translation with dummy facts defined in Definition 32 produces the same traces as $[\![P]\!]$, excluding traces not consistent with the axioms. For this we define the function d which removes any dummy fact from an execution, i.e.,

$$d(\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_n} S_n) = \emptyset \xrightarrow{F_1} S'_1 \xrightarrow{F_2} \ldots \xrightarrow{F_n} S'_n$$

where $S'_i = S_i \setminus^{\#} \cup_{t \in \mathcal{T}} !\text{Dum}(t)$.

Lemma 12*:* Given a ground process P, we have that

$$filter(exec^{msr}([\![P]\!])) = filter(d(exec^{msr}([\![P]\!]^D \cup MD)))$$

Proof. The only rules in $[\![P]\!]^D$ that differ from $[\![P]\!]$ are translations of insert and lookup. The first one only adds a permanent fact, which by the definition of d, is removed when applying d. The second one requires a fact $!Dum(t)$, whenever the rule is instantiated such the actions equals $lsIn(s,t)$ for some s. Since the translation is otherwise the same, we have that

$$\mathit{filter}(d(\mathit{exec}^{msr}([\![P]\!]^D \cup MD))) \subseteq \mathit{filter}(\mathit{exec}^{msr}([\![P]\!]))$$

For any trace in $\mathit{filter}(d(\mathit{exec}^{msr}([\![p]\!] \cup MD)))$ and any action $lsIn(s,t)$ in this trace, there is an earlier action $lnsert(s',t')$ such that $s = s'$ and $t = t'$, as otherwise α_{in} would not hold. Therefore the same trace is part of $\mathit{filter}(d(\mathit{exec}^{msr}([\![p]\!]^D \cup MD)))$, as this means that whenever $!Dum(t)$ is in the premise, $!Dum(t')$ for $t = t'$ has previously appeared in the conclusion. Since it is a permanent fact, it has not disappeared and therefore

$$\mathit{filter}(d(\mathit{exec}^{msr}([\![P]\!]^D \cup MD))) \subseteq \mathit{filter}(\mathit{exec}^{msr}([\![P]\!]))$$

□

We slightly abuse notation by defining *filter* on executions to filter out all traces contradicting the axioms, see Definition 16.

Lemma 13: Let P be a ground process and $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_n} S_n \in \mathit{filter}(\mathit{exec}^{msr}([\![P]\!]))$. For all $1 \leqslant i \leqslant n$, if $Fr(a) \in S_i$ and $F(t_1,\ldots,t_k) \in S_i$ for any $F \in \Sigma_{fact} \setminus \{Fr\}$, then $a \notin \cap_{t=_E t'} \mathit{names}(t')$, for any $t \in \{t_1,\ldots,t_k\}$.

Proof. The translation with the dummy fact introduced in Section B.1 (see Definition 32) will make this proof easier as for $[\![P]\!]^D \cup MD$, we have that the third condition of Definition 2 holds, namely,

$$\forall l' -[a']\rightarrow r' \in_E \mathit{ginsts}(l -[a]\rightarrow r) : \\ \cap_{r''=_E r'} \mathit{names}(r'') \cap FN \subseteq \cap_{l''=_E l'} \mathit{names}(l'') \cap FN \quad (3)$$

We will show that the statement holds for all $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_n} S_n \in \mathit{filter}(\mathit{exec}^{msr}([\![P]\!]^D \cup MD))$, which implies the claim, since

$$\mathit{filter}(\mathit{exec}^{msr}([\![P]\!])) = \mathit{filter}(d(\mathit{exec}^{msr}([\![P]\!]^D \cup MD)))$$

by Lemma 12. We proceed by induction on n, the length of the execution.

- Base case, $n = 0$. We have that $S_0 = \emptyset$ and therefore the statement holds trivially.
- Inductive case, $n \geqslant 1$. We distinguish two cases.

1. A rule that is not FRESH was applied and there is a fact $F(t_1,\dots,t_k) \in S_n$, such that $F(t_1,\dots,t_k) \notin S_{n-1}$, and $\mathsf{Fr}(a) \in S_n$ such that $a \in \cap_{t_i =_E t'} \mathit{names}(t')$ for some t_i. (If there are no such $F(t_1,\dots,t_k)$ and $\mathsf{Fr}(a)$ we immediately conclude by induction hypothesis.) By Equation 3, $a \in t'_j$ for some $F'(t'_1,\dots,t'_l) \in S_{n-1}$. Since FRESH is the only rule that adds a Fr-fact and $\mathsf{Fr}(a) \in S_n$, it must be that $\mathsf{Fr}(a) \in S_{n-1}$, contradicting the induction hypothesis. Therefore this case is not possible.
2. The rule FRESH was applied, i. e., $\mathsf{Fr}(a) \in S_n$ and $\mathsf{Fr}(a) \notin S_{n-1}$. If there is no $a \in \cap_{t_i =_E t'} \mathit{names}(t')$ for some t_i, and $F(t_1,\dots,t_k) \in S_n$, then we conclude by induction hypothesis. Otherwise, if there is such a $F(t_1,\dots,t_k) \in S_n$, then, by Equation 3, $a \in t'_j$ for some $F'(t'_1,\dots,t'_l) \in S_i$ for $i < n$. We construct a contradiction to the induction hypothesis by taking the prefix of the execution up to i and appending the instantiation of the FRESH rule to its end. Since $\mathsf{d}(\mathit{exec}^{msr}(\llbracket P \rrbracket^D \cup MD))$ is prefix closed by Lemma 11 we have that $\emptyset \xrightarrow{F_1} S'_1 \xrightarrow{F_2} \dots \xrightarrow{F_i} S_i \in \mathit{filter}(\mathsf{d}(\mathit{exec}^{msr}(\llbracket P \rrbracket^D \cup MD)))$. Moreover as rule FRESH was applied adding $\mathsf{Fr}(a) \in S_n$ it is also possible to apply the same instance of FRESH to the prefix (by Definition 4) and therefore

$$\emptyset \xrightarrow{F_1} S'_1 \xrightarrow{F_2} \dots \xrightarrow{F_i} S_i \longrightarrow S_i \cup \{\mathsf{Fr}(a)\} \in \mathit{filter}(\mathsf{d}(\mathit{exec}^{msr}(\llbracket P \rrbracket^D \cup MD)))$$

contradicting the induction hypothesis.

□

Lemma 14: For any frame $\nu\tilde{n}.\sigma$, $t \in \mathcal{M}$ and $a \in FN$, if $a \notin \mathit{st}(t)$, $a \notin \mathit{st}(\sigma)$ and $\nu\tilde{n}.\sigma \vdash t$, then $\nu\tilde{n}, a.\sigma \vdash t$.

Proof. In [1, Proposition 1] it is shown that $\nu\tilde{n}.\sigma \vdash t$ if and only if $\exists M.\mathit{fn}(M) \cap \tilde{n} = \emptyset$ and $M\sigma =_E t$. Define M' as M renaming a to some fresh name, i.e., not appearing in $\tilde{n}, \sigma, t$. As $a \notin \mathit{st}(\sigma, t)$ and the fact that equational theories are closed under bijective renaming of names we have that $M'\sigma =_E t$ and $\mathit{fn}(M') \cap (\tilde{n}, a) = \emptyset$. Hence $\nu\tilde{n}, a.\sigma \vdash t$. □

Lemma 2: Let P be a ground process and $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \dots \xrightarrow{F_n} S_n \in \mathit{filter}(\mathit{exec}^{msr}(\llbracket P \rrbracket))$. Let

$$\{t_1,\dots,t_m\} = \{t \mid \mathsf{Out}(t) \in_{1 \leqslant j \leqslant n} S_j\},$$
$$\sigma = \{^{t_1}/_{x_1},\dots,^{t_m}/_{x_m}\}, \text{ and}$$
$$\tilde{n} = \{a : \mathit{fresh} \mid \mathit{ProtoNonce}(a) \in_E \bigcup_{1 \leqslant j \leqslant n} E_j\}.$$

We have that

1. if $!K(t) \in S_n$ then $\nu\tilde{n}.\sigma \vdash t$;
2. if $\nu\tilde{n}.\sigma \vdash t$ then there exists S such that
   - $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_n} S_n \longrightarrow^* S \in \mathit{filter}(\mathit{exec}^{msr}_E([\![P]\!]))$,
   - $!K(t) \in_E S$ and
   - $S_n \rightarrow^*_R S$ for $R = \{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl}, \text{Fresh}\}$.

Proof. We prove both items separately.

1. The proof proceeds by induction on n, the number of steps of the execution.

   BASE CASE: N=0. This case trivially holds as $S_n = \emptyset$.

   INDUCTIVE CASE: N>0. By induction we suppose that if $!K(t) \in S_{n-1}$ then $\nu\tilde{n}'.\sigma' \vdash t$ where $\tilde{n}', \sigma'$ are defined in a similar way as $\tilde{n}, \sigma$ but for the execution of size $n-1$. We proceed by case analysis on the rule used to extend the execution.

   - MDOut. Suppose that

     $$\mathsf{Out}(u) \dashv[]\rightarrow K(u) \in \mathit{ginsts}(\text{MDOut})$$

     is the rule used to extend the execution. Hence $\mathsf{Out}(u) \in S_{n-1}$ and by definition of σ there exists x such that $x\sigma = u$. We can apply deduction rule DFrame and conclude that $\nu\tilde{n}.\sigma \vdash u$. If $K(t) \in S_n$ and $t \neq u$ we conclude by induction hypothesis as $\tilde{n} = \tilde{n}', \sigma = \sigma'$.
   - MDPub. Suppose that

     $$\dashv[]\rightarrow K(a : \mathit{pub}) \in \mathit{ginsts}(\text{MDPub})$$

     is the rule used to extend the execution. As names of sort pub are never added to $\tilde{n}$ we can apply deduction rule DName and conclude that $\nu\tilde{n}.\sigma \vdash a$. If $K(t) \in S_n$ and $t \neq a$ we conclude by induction hypothesis as $\tilde{n} = \tilde{n}', \sigma = \sigma'$.
   - MDFresh. Suppose that

     $$\mathsf{Fr}(a : \mathit{fresh}) \dashv[]\rightarrow K(a : \mathit{fresh}) \in \mathit{ginsts}(\text{MDFresh})$$

     is the rule used to extend the execution. By definition of an execution we have that $\mathsf{Fr}(a : \mathit{fresh}) \neq (S_{j+1} \setminus S_j)$ for any $j \neq n-1$. Hence $n \notin \tilde{n}$. We can apply deduction rule DName and conclude that $\nu\tilde{n}.\sigma \vdash a$. If $K(t) \in S_n$ and $t \neq a$ we conclude by induction hypothesis as $\tilde{n} = \tilde{n}', \sigma = \sigma'$.

- MDAPPL. Suppose that

$$!K(t_1), \ldots, K(t_k) \dashrightarrow K(f(t_1, \ldots, t_k)) \in \mathit{ginsts}(\text{MDAPPL})$$

  is the rule used to extend the execution. We have that $K(t_1), \ldots, K(t_k) \in S_{n-1}$, and thus by induction hypothesis, $\nu\tilde{n}'.\sigma' \vdash t_i$ for $1 \leqslant i \leqslant k$. As $\tilde{n} = \tilde{n}', \sigma = \sigma'$ we have that $\nu\tilde{n}.\sigma \vdash t_i$ for $1 \leqslant i \leqslant k$ and can apply deduction rule DAPPL and conclude that $\nu\tilde{n}.\sigma \vdash f(t_1, \ldots, t_k)$. If $K(t) \in S_n$ and $t \neq f(t_1, \ldots, t_k)$ we conclude by induction hypothesis as $\tilde{n} = \tilde{n}', \sigma = \sigma'$.
- If $S_{n-1} \xrightarrow{\mathit{ProtoNonce}(a)} S_n$, then $\mathsf{Fr}(a) \in S_{n-1}$. By Lemma 13, we obtain that if $!K(t) \in S_{n-1}$ then $a \notin \mathit{st}(t)$ and $a \notin \mathit{st}(\sigma')$. For each $!K(u) \in S_n$ there is $!K(u) \in S_{n-1}$, and by induction hypothesis, $\nu\tilde{n}'.\sigma' \vdash u$. By Lemma 14 and the fact that $\sigma = \sigma'$ we conclude that $\nu\tilde{n}.\sigma \vdash u$.
- All other rules do neither add $!K(\ )$-facts nor do they change $\tilde{n}$ and may only extend σ. Therefore we conclude by the induction hypothesis.

2. Suppose that $\nu\tilde{n}.\sigma \vdash t$. We proceed by induction on the proof tree witnessing $\nu\tilde{n}.\sigma \vdash t$.

BASE CASE. The proof tree consists of a single node. In this case one of the deduction rules DNAME or DFRAME has been applied.

- DNAME. We have that $t \notin \tilde{n}$. If $t \in \mathit{PN}$ we use rule MDPUB and we have that $S_n \to S = S_n \cup \{!K(t)\}$. In case $t \in \mathit{FN}$ we need to consider 3 different cases: *(i)* $!K(t) \in S_n$ and we immediately conclude (by letting $S = S_n$), *(ii)* $\mathsf{Fr}(t) \in S_n$ and applying rule MDFRESH we have that $S_n \to S = S_n \cup \{!K(t)\}$, *(iii)* $\mathsf{Fr}(t) \notin S_n$. By inspection of the rules we see that $\mathsf{Fr}(t) \notin S_i$ for any $1 \leqslant i \leqslant n$: The only rules that could remove $\mathsf{Fr}(t)$ are MDFRESH which would have created the persistent fact $!K(t)$, or the *ProtoNonce* rules which would however have added t to $\tilde{n}$. Hence, applying successively rules FRESH and MDFRESH yields a valid extension of the execution $S_n \to S_n \cup \{\mathsf{Fr}(t)\} \to S = S_n \cup \{!K(t)\}$.
- DFRAME. We have that $x\sigma = t$ for some $x \in \mathbf{D}(\sigma)$, that is, $t \in \{t_1, \ldots, t_m\}$. By definition of $\{t_1, \ldots, t_m\}$, $\mathsf{Out}(t) \in S_i$ for some $i \leqslant n$. If $\mathsf{Out}(t) \in S_n$ we have that $S_n \to S = (S_n \setminus \{\mathsf{Out}(t)\}) \cup \{!K(t)\}$ applying rule MDOUT. If $\mathsf{Out}(u) \notin S_n$, the fact that the only rule in $[\![P]\!]$ that allows to remove an Out-fact is MDOUT, suggests that it was applied before, and thus $!K(u) \in S$.

INDUCTIVE CASE. We proceed by case distinction on the last deduction rule which was applied.

- DAPPL. In this case $t = f(t_1, \ldots, t_k)$, such that $f \in \Sigma^k$ and $\nu\tilde{n}\tilde{r}.\sigma \vdash t_i$ for every $i \in \{1, \ldots, k\}$. Applying the induction hypothesis we obtain that there are k transition sequences $S_n \to_R^* S^i$ for $1 \leqslant i \leqslant k$ which extend the execution such that $t_i \in S^i$. All of them only add $!K$ facts which are persistent facts. If any two of these extensions remove the same $\mathsf{Out}(t)$-fact or the same $\mathsf{Fr}(a)$-fact it also adds the persistent fact $!K(t)$, respectively $!K(a)$, and we simply remove the second occurrence of the transition. Therefore, applying the same rules as for the transitions $S_n \to^* S^i$ (and removing duplicate rules) we have that $S_n \to^* S'$ and $!K(t_1), \ldots, !K(t_k) \in S'$. Applying rule MDAPPL we conclude.
- DEQ. By induction hypothesis there exists S as required with $!K(t') \in_E S$ and $t =_E t'$ which allows us to immediately conclude that $!K(t) \in_E S$.

□

B.2.1 *Proofs for Section 5.4.2*

Remark 3: Note that $\leftrightarrow_P$ (see Definition 19) has the following properties (by the fact that it defines a bijection between multisets).

- If $\mathcal{P}_1 \leftrightarrow_P S_1$ and $\mathcal{P}_2 \leftrightarrow_P S_2$ then $\mathcal{P}_1 \cup^\# \mathcal{P}_2 \leftrightarrow_P S_1 \cup^\# S_2$.
- If $\mathcal{P}_1 \leftrightarrow_P S_1$ and $Q \leftrightarrow_P \mathsf{state}_p(\tilde{t})$ for $Q \in \mathcal{P}_1$ and $\mathsf{state}_p(\tilde{t}) \in S_1$ (i.e. Q and $\mathsf{state}_p(\tilde{t})$ are related by the bijection defined by $\mathcal{P}_1 \leftrightarrow_P S_1$) then $\mathcal{P}_1 \setminus^\# \{Q\} \leftrightarrow_P S_1 \setminus^\# \{\mathsf{state}_p(\tilde{t})\}$.

The following lemma will be used throughout Lemma 3.

Lemma 15: If $\nu\tilde{n}.\sigma \vdash t$, $\tilde{n} =_E \tilde{n}'$, $\sigma =_E \sigma'$ and $t =_E t'$, then $\nu\tilde{n}'.\sigma' \vdash t'$.

Proof. Assume $\nu\tilde{n}.\sigma \vdash t$. Since an application of DEQ can be appended to the leafs of its proof tree, we have $\nu\tilde{n}.\sigma' \vdash t$. Since DEQ can be applied to its root, we have $\nu\tilde{n}.\sigma' \vdash t'$. Since $\tilde{n}, \tilde{n}'$ consist only of names, $\tilde{n} = \tilde{n}'$ and thus $\nu\tilde{n}'.\sigma' \vdash t'$. □

The following proposition is used in Lemma 3.

Proposition 3: Let A be a finite set, $<$ a strict total order on A and p a predicate on elements of A. We have that

$$\begin{aligned} \forall i \in A.p(i) \quad &\Leftrightarrow \quad \forall i \in A.\ p(i) \vee \exists j \in A.\ i < j \wedge \neg p(j) \\ &(\Leftrightarrow \quad \forall i \in A.\ \neg p(i) \to \exists j \in A.\ i < j \wedge \neg p(j)) \end{aligned}$$

and

$$\exists i \in A.p(i) \quad \Leftrightarrow \quad \exists i \in A.p(i) \wedge \forall j \in A.\ i < j \to \neg p(j)$$

The following Lemma implies the first direction of the inclusion.

Lemma 3: Let P be a well-formed ground process and

$$(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) \xrightarrow{E_1} (\mathcal{E}_1, \mathcal{S}_1, \mathcal{S}_1^{MS}, \mathcal{P}_1, \sigma_1, \mathcal{L}_1) \xrightarrow{E_2} \\ \ldots \xrightarrow{E_n} (\mathcal{E}_n, \mathcal{S}_n, \mathcal{S}_n^{MS}, \mathcal{P}_n, \sigma_n, \mathcal{L}_n)$$

where $(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) = (\emptyset, \emptyset, \emptyset, \{P\}, \emptyset, \emptyset)$. Then, there are $(F_1, S_1), \ldots, (F_{n'}, S_{n'})$ such that

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \ldots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \in exec^{msr}([\![P]\!])$$

and there exists a monotonic, strictly increasing function $f\colon \mathbb{N}_n \to \mathbb{N}_{n'}$ such that $f(n) = n'$ and for all $i \in \mathbb{N}_n$

1. $\mathcal{E}_i = \{a \mid ProtoNonce(a) \in \bigcup_{1 \leqslant j \leqslant f(i)} F_j\}$
2. $\forall t \in \mathcal{M}.\, \mathcal{S}_i(t) = \begin{cases} u & \text{if } \exists j \leqslant f(i).\mathrm{Insert}(t,u) \in F_j \\ & \wedge \forall j', u'. \\ & \quad j < j' \leqslant f(i) \Rightarrow \mathrm{Insert}(t,u') \notin_E F_{j'} \\ & \quad \wedge \mathrm{Delete}(t) \notin_E F_{j'} \\ \bot & \text{otherwise} \end{cases}$
3. $\mathcal{S}_i^{MS} = S_{f(i)} \setminus^{\#} \mathcal{F}_{res}$
4. $\mathcal{P}_i \leftrightarrow_P S_{f(i)}$
5. $\{x\sigma_i \mid x \in \mathbf{D}(\sigma_i)\}^{\#} = \{\mathsf{Out}(t) \in \cup_{k \leqslant f(i)} S_k\}^{\#}$
6. $\mathcal{L}_i =_E \{t \mid \exists j \leqslant f(i), u.\ \mathrm{Lock}(u,t) \in_E F_j \wedge \forall j < k \leqslant f(i).\ \mathrm{Unlock}(u,t) \notin_E F_k\}$
7. $[F_1, \ldots, F_{n'}] \vDash \alpha$ where α is defined as in Definition 15.
8. $\exists k.\ f(i-1) < k \leqslant f(i)$ and $E_i = F_k$ and $\cup_{f(i-1)<j\neq k\leqslant f(i)} F_j \subseteq \mathcal{F}_{res}$

Proof. We proceed by induction over the number of transitions n.

Base Case. For $n = 0$, we let $f(n) = 1$ and S_1 be the multiset obtained by using the Rule INIT:

$$\emptyset \xrightarrow{\mathsf{Init}} \{\mathsf{state}_{[]}()\}^{\#}$$

Condition 1, Condition 2, Condition 3, Condition 5, Condition 6, Condition 7 and Condition 8 hold trivially. To show that Condition 4 holds, we have to show that $\mathcal{P}_0 \leftrightarrow_P \{\mathsf{state}_{[]}()\}^{\#}$. Note that $\mathcal{P}_0 = \{P\}^{\#}$. We choose the bijection such that $P \leftrightarrow_P \mathsf{state}_{[]}()$. For $\tau = \emptyset$ and $\rho = \emptyset$ we have that $P|_{[]}\tau = P\tau = P\rho$. By Definition 18, $[\![P]\!]_{=[]} = [\![P, [], []]\!]_{=[]}$. We see from Figure 14 that for every P we have that $\mathsf{state}_{[]}() \in prems([\![P, [], []]\!]_{=[]})$. Hence, we conclude that there is a ground instance $ri \in_E ginsts([\![P]\!]_{=[]})$ with $\mathsf{state}_{[]}() \in prems(ri)$.

Inductive step. Assume the invariant holds for $n-1 \geqslant 0$. We have to show that the lemma holds for n transitions

$$(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) \xrightarrow{E_1} (\mathcal{E}_1, \mathcal{S}_1, \mathcal{S}_1^{MS}, \mathcal{P}_1, \sigma_1, \mathcal{L}_1) \xrightarrow{E_2} \ldots \xrightarrow{E_n} (\mathcal{E}_n, \mathcal{S}_n, \mathcal{S}_n^{MS}, \mathcal{P}_n, \sigma_n, \mathcal{L}_n)$$

By induction hypothesis, we have that there exists a monotonically increasing function from $\mathbb{N}_{n-1} \to \mathbb{N}_{n'}$ and an execution

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \ldots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \in exec^{msr}([\![P]\!])$$

such that Conditions 1 to 8 hold. Let f_p be this function and note that $n' = f_p(n-1)$. Fix a bijection such that $\mathcal{P}_{n-1} \leftrightarrow_P S_{f_p(n-1)}$. We will abuse notation by writing $P \leftrightarrow_P \mathsf{state}_p(\tilde{t})$, if this bijection goes from P to $\mathsf{state}_p(\tilde{t})$.

We now proceed by case distinction over the type of transition from $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}_{n-1}^{MS}, \mathcal{P}_{n-1}, \sigma_{n-1}, \mathcal{L}_{n-1})$ to the last state $(\mathcal{E}_n, \mathcal{S}_n, \mathcal{S}_n^{MS}, \mathcal{P}_n, \sigma_n, \mathcal{L}_n)$. We will (unless stated otherwise) extend the previous execution by a number of steps, say s, from $S_{n'}$ to some $S_{n'+s}$, and prove that Conditions 1 to 8 hold for n (since by induction hypothesis, they hold for all $i < n$) and a function $f\colon \mathbb{N}_n \to \mathbb{N}_{n'+s}$ that is defined as follows:

$$f(i) := \begin{cases} f_p(i) & \text{if } i \in \mathbb{N}_{n-1} \\ n' + s & \text{if } i = n \end{cases}$$

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}_{n-1}^{MS}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{0\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \to (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}_{n-1}^{MS}, \mathcal{P}', \sigma_{n-1}, \mathcal{L}_{n-1})$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that $0 \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}([\![P]\!]_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose

$$ri = [\mathsf{state}_p(\tilde{t})] \multimap [] \to [].$$

We can extend the previous execution by one step using *ri*, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \ldots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \to_{[\![P]\!]} S_{n'+1} \in exec^{msr}([\![P]\!])$$

with $S_{n'+1} = \{ S_{f(n-1)} \setminus \{\mathsf{state}_p(\tilde{t})\}$. It is left to show that Conditions 1 to 8 hold for n.

Condition 1, Condition 2, Condition 3, Condition 5, Condition 6, Condition 7, and Condition 8 hold trivially.

Condition 4 holds because $\mathcal{P}' = \mathcal{P}_{n-1} \setminus^{\#} \{0\}$, $S_{f(n)} = S_{f(n-1)} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\}^{\#}$, and $0 \leftrightarrow_P \mathsf{state}_p(\tilde{t})$ (see Remark 3).

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}_{n-1}^{MS}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{Q|R\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \to (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}_{n-1}^{MS}, \mathcal{P}' \cup \{Q, R\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that $Q|R \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19,

there is a $ri \in ginsts([\![P]\!]_{=p})$ such that $state_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose

$$ri = [state_p(\tilde{t})] \dashv[]\rightarrow [state_{p.1}(\tilde{t}), state_{p.2}(\tilde{t})].$$

We can extend the previous execution by one step using ri, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \rightarrow_{[\![P]\!]} S_{n'+1} \in exec^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{f(n-1)} \setminus \{state_p(\tilde{t})\}^{\#} \cup \{state_{p.1}(\tilde{t}), state_{p.2}(\tilde{t})\}^{\#}$. It is left to show that Conditions 1 to 8 hold for n.

Condition 1, Condition 2, Condition 3, Condition 5, Condition 6, Condition 7 and Condition 8 hold trivially. We now show that Condition 4 holds.

Condition 4 holds because $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{Q|R\} \cup^{\#} \{Q, R\}$, $\{Q\} \leftrightarrow_p \{state_{p.1}(\tilde{x})\}$ and $\{R\} \leftrightarrow_p \{state_{p.2}(\tilde{x})\}$ (by definition of the translation) (see Remark 3).

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{!Q\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \rightarrow (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup \{!Q, Q\}, \sigma_{n-1}, \mathcal{L}_{n-1})$**.** Let p and $\tilde{t}$ such that $!^i Q \leftrightarrow_p state_p(\tilde{t})$. By Definition 19, there is a $ri \in ginsts([\![P]\!]_{=p})$ such that $state_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose $ri = [state_p(\tilde{t})] \dashv[]\rightarrow [state_p(\tilde{t}), state_{p.1}(\tilde{t})]$. We can extend the previous execution by 1 step using ri, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{(ri)}_{[\![P]\!]} S_{n'+1} \in exec^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{f(n)} \cup^{\#} \{state_{p.1}(\tilde{t})\}^{\#}$. Condition 4 holds because $\mathcal{P}_n = \mathcal{P}_{n-1} \cup^{\#} \{Q\}$ and $\{Q\} \leftrightarrow_p \{state_{p.1}(\tilde{t})\}$ (by definition of $[\![P]\!]_{=p}$). Condition 1, Condition 2, Condition 3, Condition 5, Condition 6, Condition 7 and Condition 8 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{i_n-1}, \mathcal{S}^{MS}_{i_n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\nu a; Q\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \rightarrow (\mathcal{E}_{n-1} \cup \{a'\}, \mathcal{S}_{i_n-1}, \mathcal{S}^{MS}_{i_n-1}, \mathcal{P}' \cup \{Q\{a'/a\}\}, \sigma_{n-1}, \mathcal{L}_{n-1})$ **for a fresh** a'**.** Let p and $\tilde{t}$ be such that $\{\nu a; Q\} \leftrightarrow_p state_p(\tilde{t})$. There is a $ri \in ginsts([\![P]\!]_{=p})$ such that $state_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, there is a $ri \in ginsts([\![P]\!]_{=p})$, $ri = [state_p(\tilde{t}), Fr(a' : fresh)] \dashv[ProtoNonce(a' : fresh)] \rightarrow [state_{p.1}(\tilde{t}, a' : fresh)]$. Assume there is an $i < n'$ such that $Fr(a') \in S_i$. If $Fr(a') \in S_n$, then we can remove the application of the instance of FRESH that added $Fr(a')$ while still preserving Conditions 1 to 8. If $Fr(a')$ is consumned at some point, by the definition of $[\![P]\!]$, the transition where it is consumned is annotated either $ProtoNonce(a')$ or $Lock(a', t)$ for some t. In the last case, we can apply a substitution to the execution that substitutes a by a different fresh name that never appears in $\cup_i \leqslant n' S_i$. The conditions we have by induction hypothesis hold on this execution, too, since $Lock \in \mathcal{F}_{res}$, and therefore Condition 8 is not affected. The first case implies that $a' \in \mathcal{E}_{n-1}$, contradicting the assumption that a' is fresh

with respect to the process execution. Therefore, without loss of generality, the previous execution does not contain an $i < n'$ such that $\mathsf{Fr}(a') \in S_i$, and we can extend the previous execution by two steps using the FRESH rule and *ri*, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \cdots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{(\text{FRESH})}_{[\![P]\!]} S_{n'+1} \xrightarrow{(ri)}_{[\![P]\!]} S_{n'+2} \in \mathit{exec}^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{n'} \cup^{\#} \{\mathsf{Fr}(a' : \mathit{fresh})\}^{\#}$ and $S_{n'+2} = S_{f(n)} = S_{n'} \cup^{\#} \{\mathsf{state}_{p.1}(\tilde{t}, a' : \mathit{fresh})\}^{\#}$. We define $f(i) := f_p(i)$ for $i < n$ and $f(n) := f(n-1) + 2$. We now show that Condition 4 holds. As by induction hypothesis $\nu a; Q \leftrightarrow_P \mathsf{state}_{p.1}(\tilde{t})$ we also have that $P|_p \sigma = \nu a; Q\rho$ for some σ and ρ. Extending ρ with $\{a' \mapsto a\}$ it is easy to see from definition of $[\![P]\!]_{=p}$ that $\{Q\{^{a'}/_a\}\} \leftrightarrow_P \{\mathsf{state}_{p.1}(\tilde{t}, a')\}$. As $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\nu a; Q\} \cup^{\#} \{Q\{^{a'}/_a\}\}^{\#}$, we also immediately obtain that $\mathcal{P}_n \leftrightarrow_P S_{f(n)}$. Since $F_n = \mathit{ProtoNonce}(a')$ and a' is fresh, and therefore $\{a'\} = \mathcal{E}_n \setminus \mathcal{E}_{n-1}$, Condition 1 holds. Condition 2, Condition 3, Condition 5, Condition 6, Condition 7 and Condition 8 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1}, \sigma_{n-1}, \mathcal{L}_{n-1}) \xrightarrow{K(t)} (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1}, \sigma_{n-1}, \mathcal{L}_{n-1})$. This step requires that $\nu\mathcal{E}_{n-1}.\sigma_{n-1} \vdash t$. From Lemma 2 follows that there is an execution $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_{n'}} S_{n'} \rightarrow^* S \in \mathit{exec}^{msr}_E([\![P]\!])$ such that $!K(t) \in_E S$ and $S_{n'} \rightarrow^*_R S$ for $R = \{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}\}$.

From S, we can go one further step using MDIN, since $!K(t) \in S$:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \cdots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \rightarrow^*_{R \subset [\![P]\!]} S = S_{n'+s-1} \xrightarrow{K(t)}_{[\![P]\!]} S_{n'+s} \in \mathit{exec}^{msr}([\![P]\!])$$

where $S_{n'+s} = S \cup \{\mathsf{In}(t)\}$.

From the fact that $S_{f(n-1)} \rightarrow^*_R S_{f(n)} = S$, and the induction hypothesis, we can conclude that Condition 8 holds. Condition 4 holds since $\mathcal{P}_n = \mathcal{P}_{n-1}$ and no `state`-facts where neither removed nor added. Condition 1, Condition 2, Condition 3, Condition 5, Condition 6 and Condition 7 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\mathsf{out}(t, t'); Q\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \xrightarrow{K(t)} (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup^{\#} \{Q\}, \sigma_{n-1} \cup \{^{t'}/_x\}, \mathcal{L}_{n-1})$. This step requires that x is fresh and $\nu\mathcal{E}_{n-1}.\sigma \vdash t$. Using Lemma 2, we have that there is an execution $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_{f(n)}} S_{f(n-1)} \rightarrow^* S \in \mathit{exec}^{msr}_E([\![P]\!])$ such that $!K(t) \in_E S$ and $S_{f(n-1)} \rightarrow^*_R S$ for $R = \{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}\}$. Let p and $\tilde{t}$ s.t. $\{\mathsf{out}(t, t'); Q\} \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}([\![P]\!]_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. From the definition of $[\![P]\!]_{=p}$, we see that we can choose $ri = [\mathsf{state}_p(\tilde{t}), \mathsf{In}(t)] \; -[\,\mathsf{InEvent}(t)\,]\rightarrow \; [\mathsf{Out}(t'), \mathsf{state}_{p.1}(\tilde{t})]$. To apply

this rule, we need the fact $\mathsf{In}(t)$. Since $\nu\mathcal{E}_{n-1}.\sigma \vdash t$, as mentioned before, we can apply Lemma 2. It follows that there is an execution $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_{n'}} S_{n'} \to^* S \in \mathit{exec}^{\mathit{msr}}_E([\![P]\!])$ such that $!K(t) \in_E S$ and $S_{n'} \to^*_R S$ for $R = \{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}\}$. From S, we can now go two steps further, using MDIN and *ri*:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \ldots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \to^*_{R \subset [\![P]\!]} S = S_{n'+s-2} \xrightarrow{K(t)}_{[\![P]\!]} S_{n'+s-1} \xrightarrow{\mathsf{InEvent}(t)}_{[\![P]\!]} S_{n'+s} \in \mathit{exec}^{\mathit{msr}}([\![P]\!])$$

where $S_{n'+s-1} = S \cup^\# \{\mathsf{In}(t)\}^\#$ and

$$S_{f(n)} = S \setminus^\# \{\mathsf{state}_p(\tilde{t})\} \cup^\# \{\mathsf{Out}(t'), \mathsf{state}_{p.1}(\tilde{t})\}.$$

Taking $k = n' + s - 1$ we immediately obtain that Condition 8 holds. Note first that, since $S_{n'} \to_R S$, $\mathit{set}(S_{n'}) \setminus \{\mathsf{Fr}(t), \mathsf{Out}(t) | t \in \mathcal{M}\} \subset \mathit{set}(S)$ and $\mathit{set}(S) \setminus \{!K(t) | t \in \mathcal{M}\} \subset \mathit{set}(S_{n'})$. Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus \{\mathsf{out}(t, t'); Q\} \cup \{Q\}$ and $\{Q\} \leftrightarrow_P \{\mathsf{state}_{p.1}(\tilde{t})\}$ (by definition of $[\![P]\!]_{=p}$), we have that $\mathcal{P}_n \leftrightarrow_P S_{f(n)}$, i.e., Condition 4 holds. Condition 5 holds since t' was added to σ_{n-1} and $\mathsf{Out}(t)$ added to $S_{f(n-1)}$. Condition 7 holds since $K(t)$ appears right before $\mathsf{InEvent}(t)$. Condition 1, Condition 2, Condition 3 and Condition 6 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\mathsf{in}(t, N); Q\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \to (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup^\# \{Q\theta\}, \sigma_{n-1}, \mathcal{L}_{n-1})$**.** This step requires that θ is grounding for N and that $\nu\mathcal{E}_{n-1}.\sigma_{n-1} \vdash \langle t, N\theta\rangle$. Using Lemma 2, we have that there is an execution $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_{f(n-1)}} S_{f(n-1)} \to^* S \in \mathit{exec}^{\mathit{msr}}_E([\![P]\!])$ such that $!K(t) \in_E S$ and $S_{f(n-1)} \to^*_R S$ for $R = \{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}\}$. The same holds for $N\theta$. We can combine those executions, by removing duplicate instantiations of FRESH, MDFRESH and MDOUT. (This is possible since $!K$ is persistent.) Let $\emptyset \xrightarrow{F_1} S_1 \xrightarrow{F_2} \ldots \xrightarrow{F_{f(n-1)}} S_{f(n-1)} \to^*_R \overline{S} \in \mathit{exec}^{\mathit{msr}}_E([\![P]\!])$ this combined execution, and $!K(t), !K(N\theta) \in_E \overline{S}$.

Let p and $\tilde{t}$ be such that, $\mathsf{in}(t, N); Q \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19 there is a $\mathit{ri} \in \mathit{ginsts}([\![P]\!]_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. From the definition of $[\![P]\!]_{=p}$ and the fact that θ is grounding for $N\theta$, we have $\mathsf{state}_p(\tilde{t})$ in their premise, namely,

$$\mathit{ri} = [\mathsf{state}_p(\tilde{t}), \mathsf{In}(\langle t, N\theta\rangle)] \text{–}[\mathsf{InEvent}(\langle t, N\theta\rangle)]\to [\mathsf{state}_{p.1}(\tilde{t} \cup (\mathit{vars}(N)\theta)].$$

From $S_{n'}$, we can first apply the above transition $S_{n'} \to_R^* \overline{S}$, and then, (since $!K(t), !K(N\theta), state_p(\tilde{x}) \in \overline{S}$), MDAPPL for the pair constructer, MDIN and *ri*:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \ldots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \to^*_{R\subset[\![P]\!]} \overline{S} = S_{n'+s-3}$$
$$\xrightarrow{(\text{MDAPPL})}_{[\![P]\!]} S_{n'+s-2} \xrightarrow{K(\langle t, N\theta\rangle)}_{[\![P]\!]} S_{n'+s-1}$$
$$\xrightarrow{InEvent(\langle t, N\theta\rangle)}_{[\![P]\!]} S_{n'+s} \in exec^{msr}([\![P]\!]), \text{ where}$$

- since $S_{n'} \to_R S$, S is such that $set(S_{n'}) \setminus \{Fr(t), Out(t) | t \in \mathcal{M}\} \subseteq set(S)$, $set(S) \setminus \{!K(t) | t \in \mathcal{M}\} \subseteq set(S_{n'})$, and $!K(t), !K(N\theta) \in S$
- $S_{n'+s-2} = S \cup^\# \{!K(\langle t, N\theta\rangle)\}^\#$,
- $S_{n'+s-1} = S \cup^\# \{In(\langle t, N\theta\rangle)\}^\#$,
- $S_{n'+s} = S \setminus^\# \{state_p(\tilde{t})\} \cup^\# \{state_{p.1}(\tilde{t} \cup (vars(N)\theta))\}$.

Letting $k = n' + s - 1$ we immediately have that Condition 8 holds.

We now show that Condition 4 holds. Since by induction hypothesis, $in(t, N); Q \leftrightarrow_P state_p(\tilde{t})$, we have that $P|_p\tau = in(t, N); Q\rho$ for some τ and ρ. Therefore we also have that $P|_{p.1}\tau \cup (\theta\rho) = Q\rho(\theta\rho)$ and it is easy to see from definition of $[\![P]\!]_{=p}$ that $\{Q\theta\} \leftrightarrow_P \{state_{p.1}(\tilde{t}, (vars(N)\theta))\}$. Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^\# \{in(t, N); Q\} \cup^\# \{Q\}$, we have that $\mathcal{P}_n \leftrightarrow_P S_{f(n)}$, i. e., Condition 4 holds. Condition 7 holds since $!K(\langle t, N\theta\rangle)$ appears right before $InEvent\langle t, N\theta\rangle)$. Condition 1, Condition 2, Condition 3, Condition 5 and Condition 7 hold trivially.

Case: $(\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{out(c, m); Q\} \cup \{in(c', N); R\}, \sigma, \mathcal{L}) \to (\mathcal{E}, \mathcal{S}, \mathcal{S}^{MS}, \mathcal{P} \cup \{Q, R\theta\}, \sigma, \mathcal{L})$. This step requires that θ grounding for N, $t =_E N\theta$ and $c =_E c'$. Let p, p' and $\tilde{t}, \tilde{N}$ such that $\{out(c, m); P\} \leftrightarrow_P state_p(\tilde{t})$, $\{in(c', N); Q\} \leftrightarrow_P state_{p'}(\tilde{t}')$, and there are $ri \in ginsts([\![P]\!]_{=p})$ and $ri' \in ginsts([\![P]\!]_{=p'})$ such that $state_p(\tilde{t})$ and $state_{p'}(\tilde{t}')$ are part of their respective premise. From the definition of $[\![P]\!]_{=p}$ and the fact that θ is grounding for N, we have:

$$\begin{aligned}
ri_1 = \quad & [state_p(\tilde{t})] \to [Msg(t, N\theta), state^{semi}_{p.1}(\tilde{t})] \\
ri_2 = \quad & [state_{p'}(\tilde{t}'), Msg(t, N\theta)] \to [state_{p'.1}(\tilde{t}' \cup (vars(N)\theta)), \\
& \qquad\qquad Ack(t, N\theta)] \\
ri_3 = \quad & [state^{semi}_p(\tilde{t}), Ack(t, N\theta)] \to [state_{p.1}(\tilde{t})].
\end{aligned}$$

Hence, we can extend the previous execution by 3 steps:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \ldots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{(ri_1)}_{[\![P]\!]} S_{n'+s-2}$$
$$\xrightarrow{(ri_2)}_{[\![P]\!]} S_{n'+s-1} \xrightarrow{(ri_2)}_{[\![P]\!]} S_{n'+s} \in exec^{msr}([\![P]\!])$$

where:

- $S_{n'+s-2} = S_{n'} \setminus^\# \{state_p(\tilde{t})\} \cup^\# \{Msg(t, N\theta), state^{semi}_{p.1}(\tilde{t})\}^\#$,

- $S_{n'+s-1} = S_{n'} \setminus^{\#} \{\mathsf{state}_p(\tilde{t}), \mathsf{state}_{p'}(\tilde{t}')\} \cup^{\#} \{\mathsf{state}^{\mathsf{semi}}_{p.1}(\tilde{t}), \mathsf{Ack}(t, N\theta), \mathsf{state}_{p'.1}(\tilde{t}' \cup (\mathit{vars}(N)\theta))\}^{\#}$,
- $S_{n'+s} = S_{n'} \setminus^{\#} \{\mathsf{state}_p(\tilde{t}), \mathsf{state}_{p'}(\tilde{t}')\} \cup^{\#} \{\mathsf{state}_{p.1}(\tilde{t}), \mathsf{state}_{p'.1}(\tilde{t}' \cup (\mathit{vars}(N)\theta))\}$.

We have that $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\mathsf{out}(c, m); Q,\ \mathsf{in}(c', t'); R\} \cup^{\#} \{Q, R\theta\}^{\#}$. Exactly as in the two previous cases we have that $Q \leftrightarrow \mathsf{state}_{p.1}(\tilde{t})$, as well as $R\theta \leftrightarrow \mathsf{state}_{p'.1}(\tilde{t}')$. Hence we have that, Condition 4 holds. Condition 1, Condition 2, Condition 3, Condition 5, Condition 6, Condition 8 and Condition 7 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\textbf{if } t = t' \textbf{ then } Q \textbf{ else } Q'\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \to (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup \{Q\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. This step requires that $t =_E t'$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that if $t = t'$ then Q else $Q' \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}(\llbracket P \rrbracket_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $\llbracket P \rrbracket_{=p}$, we can choose

$$ri = [\mathsf{state}_p(\tilde{t})] \dashv[\mathrm{Eq}(t, t')]\rightarrow [\mathsf{state}_{p.1}(\tilde{t})].$$

We can extend the previous execution by one step using ri, therefore:

$$\emptyset \xrightarrow{F_1}_{\llbracket P \rrbracket} S_1 \xrightarrow{F_2}_{\llbracket P \rrbracket} \dots \xrightarrow{F_{n'}}_{\llbracket P \rrbracket} S_{n'} \xrightarrow{\mathrm{Eq}(t,t')}_{\llbracket P \rrbracket} S_{n'+1} \in \mathit{exec}^{msr}(\llbracket P \rrbracket)$$

with $S_{n'+1} = \{S_{n'} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\}^{\#} \cup^{\#} \{\mathsf{state}_{p.1}(\tilde{t})\}^{\#}\}$. It is left to show that Conditions 1 to 8 hold for n. The last step is labelled $F_{f(n)} = \{\mathrm{Eq}(t, t')\}^{\#}$. As $t =_E t'$, Condition 7 holds, in particular, α_{eq} is not violated. Since Eq is reserved, Condition 8 holds as well.

As before, since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\textbf{if } t = t' \textbf{ then } Q \textbf{ else } Q'\} \cup^{\#} \{Q\}$ and $\{Q\} \leftrightarrow \{\mathsf{state}_{p.1}(\tilde{t}, a)\}$ (by definition of the translation), we have that $\mathcal{P}_n \leftrightarrow_P S_{f(n)}$, and therefore Condition 4 holds. Condition 1, Condition 2, Condition 3, Condition 5 and Condition 6 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\textbf{if } t = t' \textbf{ then } Q' \textbf{ else } Q\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \longrightarrow (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup \{Q'\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. This step requires that $t \neq_E t'$. This case is similar to the previous case, except ri is chosen to be

$$[\mathsf{state}_p(\tilde{t})] \dashv[\mathrm{NotEq}(t, t')]\rightarrow [\mathsf{state}_{p.2}(\tilde{t})].$$

The condition in α_{noteq} holds since $t \neq_E t'$.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\textbf{event(F)};Q\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \xrightarrow{F}$ $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1},$
$\mathcal{P}' \cup \{Q\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that $\mathsf{event(F)};Q \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}(\llbracket P \rrbracket_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $\llbracket P \rrbracket_{=p}$, we can choose

$$ri = [\mathsf{state}_p(\tilde{t})] \dashv[F, \mathrm{Event}()]\rightarrow [\mathsf{state}_{p.1}(\tilde{t})].$$

We can extend the previous execution by one step using *ri*, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{F,\mathrm{Event}()}_{[\![P]\!]} S_{n'+1} \in exec^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{n'}) \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\} \cup^{\#} \{\mathsf{state}_{p.1}(\tilde{t})\}$. It is left to show that Conditions 1 to 8 hold for n. Condition 4 holds because $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\mathrm{event(F);Q}\} \cup^{\#} \{Q\}$ and $\{Q\} \leftrightarrow \{\mathsf{state}_{p.1}(\tilde{t})\}$ (by definition of $[\![P]\!]_{=p}$). Taking $k = f(n)$ Condition 8 holds. Condition 1, Condition 2, Condition 3, Condition 5, Condition 6 and Condition 7 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\textbf{insert } t, t'; Q\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \to (\mathcal{E}_{n-1}, \mathcal{S}_n = \mathcal{S}_{n-1}[t \mapsto t'],$
$\mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup \{Q\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that $\mathrm{insert}\ t, t'; Q \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in ginsts([\![P]\!]_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose

$$ri = [\mathsf{state}_p(\tilde{t})] \dashv[\mathrm{Insert}(t, t')]\rightarrow [\mathsf{state}_{p.1}(\tilde{t})].$$

We can extend the previous execution by one step using *ri*, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{\mathrm{Insert}(t,t')}_{[\![P]\!]} S_{n'+1} \in exec^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{f(n-1)} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\}^{\#} \cup^{\#} \{\mathsf{state}_{p.1}(\tilde{t})\}^{\#}$. It is left to show that Conditions 1 to 8 hold for n.

This step is labelled $F_{f(n)} = \mathrm{Insert}(t, t')$, hence Condition 8 holds. To see that Condition 2 holds we let $j = f(n)$ for which both conjuncts trivially hold. Since, by induction hypothesis, Condition 7 holds, i.e., $[F_1, \dots F_{n'}] \vDash \alpha$, it holds for this step too. In particular, if $[F_1, \dots F_{n'}] \vDash \alpha_{in}$ and $[F_1, \dots F_{n'}] \vDash \alpha_{notin}$, we also have that $[F_1, \dots F_{n'}, F_{f(n)}] \vDash \alpha_{in}$ and $[F_1, \dots F_{n'}, F_{f(n)}] \vDash \alpha_{notin}$: As the Insert-action was added at the last position of the trace, it appears after any InIn or IsNotSet-action and by the semantics of the logic the formula holds.

Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\mathrm{insert}\ t, t'; Q\} \cup^{\#} \{Q\}$ and $\{Q\} \leftrightarrow \{\mathsf{state}_{p.1}(\tilde{t})\}$ (by definition of $[\![P]\!]_{=p}$), we have that Condition 4 holds. Condition 1, Condition 3, Condition 5 and Condition 6 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\textbf{delete } t; Q\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \to (\mathcal{E}_{n-1}, \mathcal{S}_n = \mathcal{S}_{n-1}[t \mapsto \bot],$
$\mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup \{Q\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that $\mathrm{delete}\ t; Q \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in ginsts([\![P]\!]_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose

$$ri = [\mathsf{state}_p(\tilde{t})] \dashv[\mathrm{Delete}(t)]\rightarrow [\mathsf{state}_{p.1}(\tilde{t})].$$

We can extend the previous execution by one step using *ri*, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{\mathrm{Delete}(t)}_{[\![P]\!]} S_{n'+1} \in exec^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{f(n-1)} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\} \cup^{\#} \{\mathsf{state}_{p.1}(\tilde{t})\}$. It is left to show that Conditions 1 to 8 hold for n.

This step is labelled $F_{f(n)} = \mathsf{Delete}(t)$, hence Condition 8 holds. Since, by induction hypothesis, Condition 7 holds, i.e., $[F_1, \dots F_{n'}] \vDash \alpha$, it holds for this step too. In particular, if $[F_1, \dots F_{n'}] \vDash \alpha_{in}$ and $[F_1, \dots F_{n'}] \vDash \alpha_{notin}$, we also have that $[F_1, \dots F_{n'}, F_{f(n)}] \vDash \alpha_{in}$ and $[F_1, \dots F_{n'}, F_{f(n)}] \vDash \alpha_{notin}$: As the Insert-action was added at the last position of the trace, it appears after any InIn or IsNotSet-actions and by the semantics of the logic the formula holds.

We now show that Condition 2 holds. We have that $\mathcal{S}_n = \mathcal{S}_{n-1}[t \mapsto \bot]$ and therefore, for all $t' \neq_E Tt$, $\mathcal{S}_n(x) = \mathcal{S}_{n-1}(x)$. Hence for all such t' we have by induction hypothesis that for some u,

$$\exists j \leqslant n'.\mathsf{Insert}(t', u) \in F_j \wedge \forall j', u'.j < j' \leqslant n' \\ \rightarrow \mathsf{Insert}(t', u') \notin_E F_{j'} \wedge \mathsf{Delete}(t') \notin_E F_{j'}$$

As $F_{n'+1} \neq_E \mathsf{Delete}(x, u)$ and $F_{n'+1} \neq_E \mathsf{Insert}(x, u')$ for all $u' \in \mathcal{M}$, we also have that

$$\exists j \leqslant n' + 1.\mathsf{Insert}(t', u) \in F_j \wedge \forall j', u'.j < j' \leqslant n' + 1 \\ \rightarrow \mathsf{Insert}(t', u') \notin_E F_{j'} \wedge \mathsf{Delete}(t') \notin_E F_{j'}.$$

For $t' =_E t$, the above condition can never be true, because $F_{n'+1} = \mathsf{Delete}(t)$ which allows us to conclude that Condition 2 holds.

Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\, \mathsf{delete}\ t;\ Q \,\} \cup^{\#} \{\, Q \,\}$ and $\{P\} \leftrightarrow \{\mathsf{state}_{p.1}(\tilde{t})\}$ (by definition of $[\![P]\!]_{=p}$), we have that Condition 4 holds. Condition 1, Condition 3, Condition 5 and Condition 6 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\, \mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ Q\ \mathsf{else}\ Q' \,\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \rightarrow (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup \{\, Q\{v/x\} \,\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. This step requires that $\mathcal{S}_{n-1}(t') =_E v$ for some $t' =_E t$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that $\mathsf{lookup}\ t\ \mathsf{as}\ v\ \mathsf{in}\ Q\ \mathsf{else}\ Q' \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}([\![P]\!]_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose

$$ri = [\mathsf{state}_p(\tilde{t})] \dashv[\mathsf{IsIn}(t, v)]\rightarrow [\mathsf{state}_{p.1}(\tilde{t}, v)].$$

We can extend the previous execution by one step using ri, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{\mathsf{IsIn}(t,v)}_{[\![P]\!]} S_{n'+1} \in \mathit{exec}^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{f(n-1)} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\} \cup^{\#} \{\mathsf{state}_{p.1}(\tilde{t})\}$. It is left to show that Conditions 1 to 8 hold for n.

This step is labelled $F_{f(n)} = \mathsf{IsIn}(t, v)$, hence Condition 8 holds.

From the induction hypothesis, Condition 2, we have that there is a j such that $\mathsf{Insert}(t, t') \in_E F_j$, $j \leqslant n'$ and

$$\forall j', u'.\ j < j' \leqslant n' \rightarrow \mathsf{Insert}(t, u') \notin_E F_{j'} \wedge \mathsf{Delete}(t) \notin_E F_{j'}$$

This can be strengthened, since $F_{f(n)} = \{\mathrm{IsIn}(t, v)\}$:

$$\forall j', u'.\, j < j' \leqslant f(n) \to \mathrm{Insert}(t, u') \notin_E F_{j'} \wedge \mathrm{Delete}(t) \notin_E F_{j'}.$$

We can now conclude that Condition 2 holds. From Condition 2 it also follows that Condition 7, in particular α_{in}, holds.

We now show that Condition 4 holds. By induction hypothesis we have that lookup t as x in Q else $Q' \leftrightarrow_P \mathrm{state}_p(\tilde{t})$, and hence $P|_p\tau = (\text{lookup t as x in Q else } Q')\rho$ for some τ and ρ. Therefore we also have that $P|_{p.1}\tau \cup (\{^{v\rho}/_x\}) = Q\rho\{^{v\rho}/_x\})$ and it is easy to see from definition of $[\![P]\!]_{=p}$ that $\{Q\{^v/_x\}\} \leftrightarrow_P \{\mathrm{state}_{p.1}(\tilde{t}, v)\}$. Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\text{lookup t as x in Q else } Q'\} \cup^{\#} \{Q\{^v/_x\}\}$ we have that $\mathcal{P}_n \leftrightarrow_P S_{f(n)}$, i. e., Condition 4 holds.

Condition 1, Condition 3, Condition 5 and Condition 6 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\textbf{lookup t as x in Qelse } Q'\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \longrightarrow (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup \{Q'\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. This step requires that $S(t')$ is undefined for all $t' =_E t$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that lookup t as x in Q else $Q' \leftrightarrow_P \mathrm{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}([\![P]\!]_{=p})$ such that $\mathrm{state}_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose

$$ri = [\mathrm{state}_p(\tilde{t})] \text{–}[\mathrm{IsNotSet}(t)]\text{→} [\mathrm{state}_{p.1}(\tilde{t})].$$

We can extend the previous execution by one step using *ri*, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{\mathrm{IsNotSet}(t)}_{[\![P]\!]} S_{n'+1} \in \mathit{exec}^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{f(n-1)} \setminus^{\#} \mathrm{state}_p(\tilde{t}) \cup^{\#} \mathrm{state}_{p.1}(\tilde{t})$. It is left to show that Conditions 1 to 8 hold for n.

This step is labelled $F_{f(n)} = \mathrm{IsNotSet}(t)$, hence Condition 8 holds. Condition 2 also holds trivially and will be used to show Condition 7. Since this step requires that $S(t')$ is undefined for all $t' =_E t$, we have by Condition 2 that

$$\begin{aligned}\forall j \leqslant f(n), u.\, \mathrm{Insert}(t, u) \in_E F_j \\ \to \exists j', u'. j < j' \leqslant f(n) \\ \wedge (\mathrm{Insert}(t, u') \in_E F_{j'} \vee \mathrm{Delete}(t) \in_E F_{j'})\end{aligned}$$

Now suppose that

$$\exists i \leqslant f(n), y. \mathrm{Insert}(t, y) \in_E F_i)$$

As there exists an insert, there is a last insert and hence we also have

$$\begin{aligned}\exists i \leqslant f(n), y.\, \mathrm{Insert}(t, y) \in_E F_i \\ \wedge \forall i', y'. i < i' \leqslant f(n) \to \mathrm{Insert}(t, y') \notin_E F_{i'}\end{aligned}$$

Applying Condition 2 (cf above) we obtain that

$$\begin{aligned}\exists i \leqslant f(n), y.\, &\mathsf{Insert}(t,y) \in_E F_i \\ &\wedge \forall i', y'.\, i < i' \leqslant f(n) \rightarrow \mathsf{Insert}(t,y') \notin_E F_{i'} \\ &\wedge \exists j', u'.\, i < j' \leqslant f(n) \\ &\quad \wedge (\mathsf{Insert}(t,u') \in_E F_{j'} \vee \mathsf{Delete}(t) \in_E F_{j'})\end{aligned}$$

which simplifies to

$$\begin{aligned}\exists i \leqslant f(n), y.\, &\mathsf{Insert}(t,y) \in_E F_i \\ &\wedge \forall i', y'.\, i < i' \leqslant f(n) \rightarrow \mathsf{Insert}(t',y') \notin F_{i'} \\ &\wedge\ \exists j'.\, i < j' \leqslant f(n) \wedge \mathsf{Delete}(t) \in_E F_{j'}\end{aligned}$$

Now we weaken the statement by dropping the first conjunct and restricting the quantification $\forall i'.i < i' \leqslant f(n)$ to $\forall i'.j' < i' \leqslant f(n)$, since $i < j'$.

$$\begin{aligned}\exists i \leqslant f(n).\, \exists j'.\, i < j' \leqslant f(n) \wedge \forall i'.\, j' < i' \leqslant f(n)\\ \rightarrow \mathsf{Insert}(t',y') \notin F_{i'} \wedge \mathsf{Delete}(t) \in_E F_{j'}\end{aligned}$$

We further weaken the statement by weakening the scope of the existential quantification $\exists j'.\, i < j' \leqslant f(n)$ to $\exists j'.\, j' \leqslant f(n)$. Afterwards, i is not needed anymore.

$$\begin{aligned}\exists j'.\, j' \leqslant f(n) \wedge \forall i'.\, j' < i' \leqslant f(n)\\ \rightarrow \mathsf{Insert}(t',y') \notin F_{i'} \wedge \mathsf{Delete}(t) \in_E F_{j'}\end{aligned}$$

This statement was obtained under the hypothesis that $\exists i \leqslant f(n), y.\ \mathsf{Insert}(t,y) \in_E F_i)$. Hence we have that

$$\begin{aligned}&\forall i \leqslant f(n), y.\, \mathsf{Insert}(t,y) \notin_E F_i \\ &\vee \exists j' \leqslant f(n).\, \mathsf{Delete}(t) \in_E F_{j'} \wedge \forall i'.\, j' < i' \leqslant f(n) \\ &\qquad \rightarrow \mathsf{Insert}(t',y') \notin F_{i'}\end{aligned}$$

This shows that Condition 7, in particular α_{notin}, holds.

Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\,\mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ Q\ \mathsf{else}\ Q'\,\} \cup^{\#} \{\,Q'\,\}$ and $\{Q'\} \leftrightarrow \{\mathsf{state}_{p.1}(\tilde{t})\}$ (by definition of $[\![P]\!]_{=p}$), we have that Condition 4 holds. Condition 1, Condition 3, Condition 5 and Condition 6 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{\mathrm{MS}}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\,\mathbf{lock}\ t;\ Q\,\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \longrightarrow (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{\mathrm{MS}}_{n-1}, \mathcal{P}' \cup^{\#} \{\,Q'\,\}, \sigma_{n-1}, \mathcal{L}_{n-1} \cup \{\,t\,\})$**.** This step requires that for all $t' =_E t$, $t' \notin \mathcal{L}_{n-1}$. Let p and $\tilde{t}$ such that $\mathsf{lock}\ t;\ Q \leftrightarrow_p \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}([\![P]\!]_{=p})$ such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose $ri = [\mathsf{Fr}(l), \mathsf{state}_p(\tilde{t})] -[\mathsf{Lock}(l,t)]\rightarrow [\mathsf{state}_{p.1}(\tilde{t}, l)]$ for a fresh name l, that never appeared in a Fr-fact in $\cup_{j \leqslant f(n-1)} S_j$. We can extend the

previous execution by $s = 2$ steps using an instance of FRESH for l and *ri*:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \to_{\{\text{FRESH}\}} S_{n'+s-1} \xrightarrow{\text{Lock}(l,t)}_{[\![P]\!]} S_{n'+s} \in exec^{msr}([\![P]\!])$$

with $S_{n'+s-1} = S_{f(n-1)} \setminus^{\#} \{\, \mathsf{state}_p(\tilde{t}) \,\}^{\#} \cup^{\#} \{\, \mathsf{Fr}(l) \,\}$ and $S_{n'+s} = S_{f(n-1)} \setminus^{\#} \{\, \mathsf{state}_p(\tilde{t}) \,\}^{\#} \cup^{\#} \{\, \mathsf{state}_{p.1}(\tilde{t}) \,\}^{\#}$. It is left to show that Conditions 1 to 8 hold for n.

The step from $S_{f(n)-1}$ to $S_{f(n)}$ is labelled $F_{f(n)} = \text{Lock}(l, t)$, hence Condition 8 and Condition 2 hold.

$F_{f(n)}$ also preserves Condition 6 for the new set of active locks $\mathcal{L}_{f(n)} = \mathcal{L}_{f(n-1)} \cup \{ t \}$.

In the following we show by contradiction that α_{lock}, and therefore Condition 7 holds. α_{lock} held in the previous step, and $F_{f(n-1)+1}$ is empty, so we assume (by contradiction), that $F_{f(n)} = \text{Lock}(l, t)$ violates α_{lock}. If this was the case, then:

$$\begin{aligned} \exists i < f(n), l_1.\ \text{Lock}(l_1, t) \in_E F_i \wedge \\ \wedge \forall j.\ i < j < f(n) \to \text{Unlock}(l_1, t) \notin_E F_j \\ \vee \exists l_2, k.\ i < k < j \\ \wedge (\text{Lock}(l_2, t) \in_E F_k \vee \text{Unlock}(l_2, t) \in_E F_k) \end{aligned} \tag{4}$$

Since the semantics of the calculus requires that for all $t' =_E t$, $t' \notin \mathcal{L}_{n-1}$, by induction hypothesis, Condition 6, we have that

$$\begin{aligned} \forall i < f(n-1), l_1.\ \text{Lock}(l_1, t) \in_E F_i \to \\ \exists j.\ i < j < f(n-1) \wedge \text{Unlock}(l_1, t) \in_E F_j \end{aligned}$$

Since $F_{f(n-1)+1} = \emptyset$ and $f(n) = f(n-1) + 2$, we have:

$$\begin{aligned} \forall i < f(n), l_1. \text{Lock}(l_1, t) \in_E F_i \to \\ \exists j.\ i < j < f(n) \wedge \text{Unlock}(l_1, t) \in_E F_j \end{aligned}$$

We apply Proposition 3 for the total order $>$ on the integer interval $i+1..f(n)-1$:

$$\begin{aligned} \forall i < f(n), l_1.\ \text{Lock}(l_1, t) \in_E F_i \to \\ \exists j.\ i < j < f(n) \wedge \text{Unlock}(l_1, t) \in_E F_j \\ \wedge \quad \forall k.\ i < k < j \to \text{Unlock}(l_1, t) \notin_E F_k \end{aligned}$$

Combining this with (4) we obtain that

$$\begin{aligned} \exists i < f(n), l_1.\ \text{Lock}(l_1, t) \in_E F_i \wedge \\ \exists j.\ i < j < f(n) \wedge \text{Unlock}(l_1, t) \in_E F_j \\ \wedge \exists l_2, k.\ i < k < j \wedge (\text{Lock}(l_2, t) \in_E F_k \\ \vee (\text{Unlock}(l_2, t) \in_E F_k \wedge l_2 \neq_E l_1)) \end{aligned}$$

Fix $i < f(n)$, j such that $i < j < f(n)$, and l_1 such that $\mathsf{Lock}(l_1, t) \in_E F_i$ and $\mathsf{Unlock}(l_1, t) \in_E F_j$. Then, there are l_2 and k such that $i < k < j$ and either $\mathsf{Lock}(l_2, t) \in_E F_k$ or $\mathsf{Unlock}(l_2, t) \in_E F_k$, but $l_2 \neq_E l_1$. We proceed by case distinction.
Case 1: there is no unlock in between i and j, i. e., for all m, $i < m < j$, $\mathsf{Unlock}(l', t) \notin F_m$. Then there is a k and l_2 such that $\mathsf{Lock}(l_2, t) \in_E F_k$. In this case, α_{lock} is already invalid at the trace produced by the k-prefix of the execution, contradicting the induction hypothesis.
Case 2: there are l' and m, $i < m < j$ such that $\mathsf{Unlock}(l', t) \in F_m$ (see Figure 16).

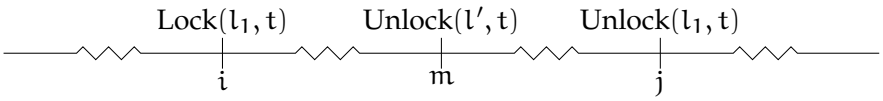


Figure 16: Visualisation of Case 2.

We first observe that for any l, u, i_1, i_2 , if $\mathsf{Unlock}(l, u) \in_E F_{i_1}$ and $\mathsf{Unlock}(l, u) \in_E F_{i_2}$, then $i_1 = i_2$. We proceed by contradiction. By definition of $[\![P]\!]$ and well-formedness of P, the steps from $i_1 - 1$ to i_1 and from $i_2 - 1$ to i_2 must be ground instances of rules $[\![P]\!]_{=q}$ and $[\![P]\!]_{=q'}$ such that $P|_q$ and $P|_{q'}$ start with unlock commands that are labelled the same and have the same parameter, since every variable $lock_l$ in $[\![P]\!]$ appears in a Fr-fact in the translation for the corresponding lock command. By definition of $\overline{P}$, this means q and q' have a common prefix q_l that starts with a lock with this label.

Let $q_l \leqslant q$ denote that q_l is a prefix of q. Since $\overline{P}$ gives $\perp$ if there is a replication or a parallel between q_l and q or q', and since P is well-formed (does not contain $\perp$), we have that every state fact state_r for $q_l \leqslant r \leqslant q$ or $q_l \leqslant r \leqslant q'$ appearing in $[\![P]\!]$ is a linear fact, since no replication is allowed between q_l and q or q'. This implies that $q' \neq q$. Furthermore, every rule in $\cup_{q_l \leqslant r \leqslant q \vee q_l \leqslant r \leqslant q'} [\![P]\!]_{=r}$ adds at most one fact state_r and if it adds one fact, it either removes a fact $\mathsf{state}_{r'}$ where $r = r' \cdot 1$ or $r' \cdot 2$, or removes a fact $\mathsf{state}^{\mathsf{semi}}_{r'}$ where $r = r' \cdot 1$, which in turn requires removing $\mathsf{state}_{r'}$ (see translation of out). Therefore, either $q \leqslant q'$ or $q' \leqslant q$. But this implies that both have different labels, and since $[\![P]\!]_{=q_l}$ requires $\mathsf{Fr}(l)$, and E distinguishes fresh names, we have a contradiction. (A similiar observation is possible for locks: For any l, u, i_1, i_2 , if $\mathsf{Lock}(l, u) \in_E F_{i_1}$ and $\mathsf{Lock}(l, u) \in_E F_{i_2}$, then $i_1 = i_2$, since by definition of the translation, the transition from $i_1 - 1$ to i_1 or $i - 2 - 1$ to i_2 removes fact $\mathsf{Fr}(l)$.)

From the first observation we learn that , $l' \neq_E l_1$ for any l' and m, $i < m < j$ such that $\mathsf{Unlock}(l', t) \in F_m$. We now choose the smallest such m. By definition of $[\![P]\!]$, the step from S_{m-1} to S_m must be ground instance of a rule from $[\![P]\!]_{=q}$ for $P|_q$ starting with unlock. Since P is well-formed, there is a q_l such that $P|_{q_l}$ starts with lock, with the same label and parameter as the unlock. As before, since P is well-formed, and therefore there are no replications and parallels

between q_l and q, there must be n such that $\text{Lock}(l', t) \in F_n$ and $n < m$. We proceed again by case distinction.
Case 2a: $n < i$ (see Figure 17). By the fact that $m > i$ we have that there is no o such that $n < o < i$ and $\text{Unlock}(l', t) \in_E F_o$ (see first observation). Therefore, the trace produced by the i-prefix of this execution does already not satisfy α_{lock}, i. e., $[F_1, \ldots, F_i] \nvDash \alpha_{lock}$.

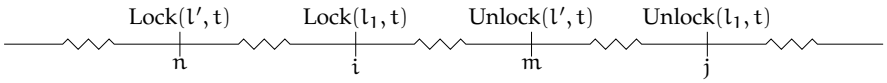


Figure 17: Visualisation of Case 2a.

Case 2b: $i < n$ (see Figure 18). Again $[F_1, \ldots, F_n] \nvDash \alpha_{lock}$, since there is no o such that $i < o < n$ and $\text{Unlock}(l_1, t) \in_E F_o$.

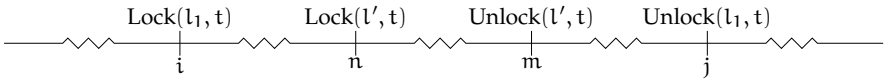


Figure 18: Visualisation of Case 2b.

Since we could, under the assumption that Condition 1 to Condition 8 hold for $i \leqslant n'$, reduce every case in which $[F_1, \ldots, F_{n'+1}] \nvDash \alpha_{lock}$ to a contradiction, we can conclude that Condition 7 holds for $n' + 1$.

Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\, \text{lock } t;\, Q \,\} \cup^{\#} \{\, Q \,\}$ and $\{Q\} \leftrightarrow \{\text{state}_{p \cdot 1}(\tilde{t})\}$ (by definition of the translation), we have that Condition 4 holds. Condition 1, Condition 3 and Condition 5 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\, \textbf{unlock } t;\, Q \,\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \rightarrow (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{MS}_{n-1}, \mathcal{P}' \cup^{\#} \{\, Q' \,\}, \sigma_{n-1}, \mathcal{L}_{n-1} \setminus \{\, t' \colon t' =_E t \,\})$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that $\text{unlock } t;\, Q \leftrightarrow_P \text{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}(\llbracket P \rrbracket_{=p})$ such that $\text{state}_p(\tilde{t})$ is part of its premise. By definition of $\llbracket P \rrbracket_{=p}$, we can choose

$$ri = [\text{state}_p(\tilde{t})] \,\text{—}[\text{Unlock}(l, t)]\rightarrow [\text{state}_{p \cdot 1}(\tilde{t})].$$

We can extend the previous execution by one step using ri, therefore:

$$\emptyset \xrightarrow{F_1}_{\llbracket P \rrbracket} S_1 \xrightarrow{F_2}_{\llbracket P \rrbracket} \ldots \xrightarrow{F_{n'}}_{\llbracket P \rrbracket} S_{n'} \xrightarrow{\text{Unlock}(l,t)}_{\llbracket P \rrbracket} S_{n'+1} \in \mathit{exec}^{msr}(\llbracket P \rrbracket)$$

with $S_{n'+1} = S_{f(n-1)} \setminus^{\#} \{\, \text{state}_p(\tilde{t}) \,\} \cup^{\#} \{\, \text{state}_{p \cdot 1}(\tilde{t}) \,\}$. It is left to show that Conditions 1 to 8 hold for n.

The step from $S_{f(n-1)}$ to $S_{f(n)}$ is labelled $F_{f(n)} = \text{Unlock}(l, t)$, hence Condition 8 and Condition 2 hold.

In order to show that Condition 6 holds, we perform a case distinction. Assume $t \notin_E \in \mathcal{L}_{n-1}$. Then, $\mathcal{L}_{f(n-1)} = \mathcal{L}_{f(n)}$. In this case, Condition 6 holds by induction hypothesis. In the following, we assume $t \in_E \mathcal{L}_{n-1}$. Thus, there is $j \in n', l'$ such that $\text{Lock}(l', t) \in_E F_j$ and for all k such that $j < k \leqslant n'$, $\text{Unlock}(l', t) \notin_E F_k$.

Since $P|_p$ is an unlock node and P is well-formed, there is a prefix q of p, such that $P|_q$ is a lock with the same parameter and annotation. By definition of $\overline{P}$, there is no parallel and no replication between q and p. Note that any rule in $[\![P]\!]$ that produces a state named state_p for a non-empty p is such that it requires a fact with name $\mathsf{state}_{p'}$ for $p = p' \cdot 1$ or $p = p' \cdot 2$ (in case of the translation of out, it might require $\mathsf{state}^{\mathsf{semi}}_{p'}$, which in turn requires $\mathsf{state}_{p'}$). This means that, since $\mathsf{state}_p(\tilde{t}) \in S_{n'}$, there is an i such that $\mathsf{state}_q(\tilde{t}') \in S_i$ and $\mathsf{state}_q(\tilde{t}') \notin S_{i-1}$ for $\tilde{t}'$ a prefix to t. This rule is an instance of $[\![P]\!]_{=q}$ and thus labelled $F_i = \mathrm{Lock}(l, t)$. We proceed by case distinction.

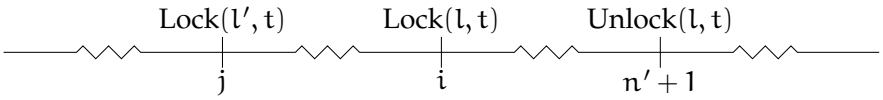


Figure 19: Visualisation of Case 1.

Case 1: $j < i$ (see Figure 19). By induction hypothesis, Condition 7 holds for the trace up to n'. But, $[F_1, \ldots, F_i] \not\models \alpha_{lock}$, since we assumed that for all k such that $j < k \leqslant n'$, $\mathrm{Unlock}(l', t) \notin_E F_k$.

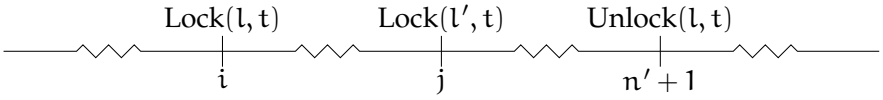


Figure 20: Visualisation of Case 2.

Case 2: $i < j$ (see Figure 20). As shown in the lock case, any k such that $\mathrm{Unlock}(l, t) \in_E F_k$ is $k = n' + 1$. This contradicts Condition 7 for the trace up to j, since $[F_1, \ldots, F_j] \not\models \alpha_{lock}$, because there is not k such that $i < k < j$ such that $\mathrm{Unlock}(l, t) \in_E F_k$. This concludes the proof that Condition 6 holds for $n + 1$.

Condition 7 holds, since none of the axioms, in particular not α_{lock}, become unsatisfied if they were satisfied for the trace up to $f(n-1)$ and an Unlock is added.

Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^{\#} \{\, \mathrm{unlock}\ t;\ Q \,\} \cup^{\#} \{\, Q \,\}$ and $\{Q\} \leftrightarrow \{\mathsf{state}_{p \cdot 1}(\tilde{t})\}$ (by definition of the translation), we have that Condition 4 holds. Condition 1, Condition 3 and Condition 5 hold trivially.

Case: $(\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{\mathrm{MS}}_{n-1}, \mathcal{P}_{n-1} = \mathcal{P}' \cup \{\, l \dashv[a]\rightarrow r;\ Q \,\}, \sigma_{n-1}, \mathcal{L}_{n-1}) \xrightarrow{a} (\mathcal{E}_{n-1}, \mathcal{S}_{n-1}, \mathcal{S}^{\mathrm{MS}}_{n-1} \setminus \mathit{lfacts}(l') \cup^{\#} \mathit{mset}(r), \mathcal{P}' \cup^{\#} \{\, Q \,\}, \sigma_{n-1}, \mathcal{L}_{n-1})$. This step requires that $l' \dashv[a']\rightarrow r' \in_E \mathit{ginsts}(l \dashv[a]\rightarrow r)$ and $\mathit{lfacts}(l') \subset^{\#} \mathcal{S}^{\mathrm{MS}}_{n-1}, \mathit{pfacts}(l') \subset \mathit{mset}(\mathcal{S}^{\mathrm{MS}}_{n-1})$. Let θ be a substitution such that $(l \dashv[a]\rightarrow r)\theta = (l' \dashv[a']\rightarrow r')$. Since, by induction hypothesis, $\mathcal{S}^{\mathrm{MS}}_{n-1} = S_{n'} \setminus^{\#} \mathcal{F}_{res}$, we therefore have $\mathit{lfacts}(l') \subset^{\#} S_{n'}, \mathit{pfacts}(l') \subset \mathit{mset}(S_{n'})$. By induction hypothesis $\mathcal{P}_{n-1} \leftrightarrow_P S_{n'}$. Let p and $\tilde{t}$ be such that $l \dashv[a]\rightarrow r;\ Q \leftrightarrow_P \mathsf{state}_p(\tilde{t})$. By Definition 19, there is a $ri \in \mathit{ginsts}([\![P]\!]_{=p})$

such that $\mathsf{state}_p(\tilde{t})$ is part of its premise. By definition of $[\![P]\!]_{=p}$, we can choose

$$ri = [\mathsf{state}_p(\tilde{t}), l'] -[a', \mathrm{Event}()]\rightarrow [r', \mathsf{state}_{p.1}(\tilde{t} \cup (\mathit{vars}(l)\theta))].$$

We can extend the previous execution by one step using *ri*, therefore:

$$\emptyset \xrightarrow{F_1}_{[\![P]\!]} S_1 \xrightarrow{F_2}_{[\![P]\!]} \dots \xrightarrow{F_{n'}}_{[\![P]\!]} S_{n'} \xrightarrow{a', \mathrm{Event}()}_{[\![P]\!]} S_{n'+1} \in \mathit{exec}^{msr}([\![P]\!])$$

with $S_{n'+1} = S_{f(n-1)} \setminus^\# \{ \mathsf{state}_p(\tilde{t}) \}^\# \setminus^\# \mathit{lfacts}(l') \cup^\# \mathit{mset}(r') \cup^\# \{ \mathsf{state}_{p.1}(\tilde{t} \cup (\mathit{vars}(l)\theta)) \}^\#$. It is left to show that Conditions 1 to 8 hold for n.

Condition 3 holds since

$$\begin{aligned}
\mathcal{S}^{MS}_n &= \mathcal{S}^{MS}_{n-1} \setminus^\# \mathit{lfacts}(l') \cup^\# \mathit{mset}(r) \\
&= (S_{n'} \setminus^\# \mathcal{F}_{\mathit{res}}) \setminus \mathit{lfacts}(l') \cup^\# \mathit{mset}(r) \quad \text{(induction hypothesis)} \\
&= (S_{n'} \setminus^\# \mathit{lfacts}(l') \cup^\# \mathit{mset}(r) \setminus^\# \{ \mathsf{state}_p(\tilde{t}) \}^\# \\
&\quad \cup^\# \{ \mathsf{state}_{p.1}(\tilde{t} \cup (\mathit{vars}(l)\theta)) \}^\#) \setminus^\# \mathcal{F}_{\mathit{res}} \\
&\qquad \text{(since } \mathsf{state}_p(\tilde{t}), \mathsf{state}_{p.1}(\tilde{t} \cup (\mathit{vars}(l)\theta)) \in \mathcal{F}_{\mathit{res}}) \\
&= S_{f(n)} \setminus^\# \mathcal{F}_{\mathit{res}}
\end{aligned}$$

The step from $S_{f(n-1)}$ to $S_{f(n)}$ is labelled $F_{f(n)} = a$, and does not contain actions in $\mathcal{F}_{\mathit{res}}$, since P is well-formed. Hence Condition 2, Condition 6, Condition 7 and Condition 8 hold.

Since $\mathcal{P}_n = \mathcal{P}_{n-1} \setminus^\# \{ l -[a]\rightarrow r; Q \} \cup^\# \{ Q \}$ and $\{Q\} \leftrightarrow \{\mathsf{state}_{p.1}(\tilde{t} \cup (\mathit{vars}(l)\theta))\}$ (by definition of $[\![P]\!]_{=p}$), we have that Condition 4 holds. Condition 1, and Condition 5 hold trivially.

□

B.2.2 *Proofs for Section 5.4.3*

Definition 33 (normal MSR execution)*:* A MSR execution

$$\emptyset \xrightarrow{E_1}_{[\![P]\!]} \dots \xrightarrow{E_n}_{[\![P]\!]} S_n \in \mathit{exec}^{msr}([\![P]\!])$$

for the multiset rewrite system $[\![P]\!]$ defined by a ground process P is *normal* if:

1. The first transition is an instance of the Init rule, i. e., $S_1 = \mathsf{state}_{[]}()$ and there is at least this transition.
2. S_n neither contains any fact with the symbol $\mathsf{state}^{\mathsf{semi}}_p$ for any p, nor any fact with symbol Ack.
3. if for some $i \in \mathbb{N}$ and $t_1, t_2 \in \mathcal{M}$, $\mathsf{Ack}(t_1, t_2) \in (S_{i-1} \setminus^\# S_i)$, then there are p and q such that:

$$S_{i-3} \rightarrow_{R_1} S_{i-2} \rightarrow_{R_2} S_{i-1} \rightarrow_{R_3} S_i \quad \text{, where:}$$

- $R_1 = [\mathsf{state}_p(\tilde{x})] \to [\mathsf{Msg}(t_1, t_2), \mathsf{state}_p^{\mathsf{semi}}(\tilde{x})]$
- $R_2 = [\mathsf{state}_q(\tilde{y}), \mathsf{Msg}(t_1, t_2)] \to [\mathsf{state}_{q \cdot 1}(\tilde{y} \cup \tilde{y}'), \mathsf{Ack}(t_1, t_2)]$
- $R_3 = [\mathsf{state}_p^{\mathsf{semi}}(\tilde{x}), \mathsf{Ack}(t_1, t_2)] \to [\mathsf{state}_{p \cdot 1}(\tilde{x})]$.

4. $S_{n-1} \xrightarrow{E_n}_{[\![P,[],[]]\!],\text{MDIN},\text{INIT}} S_n$

5. if $\mathsf{In}(t) \in (S_{i-1} \setminus^{\#} S_i)$, then $S_{i-2} \xrightarrow{K(t)}_{\text{MDIN}} S_{i-1}$ (for some $i \in \mathbb{N}$ and $t \in \mathcal{M}$)

6. if $n \geqslant 2$ and no Ack-fact in $(S_{i-1} \setminus^{\#} S_i)$, then there exists $m < n$ such that $S_m \to_R^* S_{n-1}$ for $R = \{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}, \text{FRESH}\}$ and $\emptyset \xrightarrow{E_1}_{[\![P]\!]} \cdots \xrightarrow{E_m}_{[\![P]\!]} S_m \in \mathit{exec}^{msr}([\![P]\!])$ is normal.

7. if for some $t_1, t_2 \in \mathcal{M}$, $\mathsf{Ack}(t_1, t_2) \in (S_{n-1} \setminus^{\#} S_n)$, then there exists $m \leqslant n-3$ such that $\emptyset \xrightarrow{E_1}_{[\![P]\!]} \cdots \xrightarrow{E_m}_{[\![P]\!]} S_m \in \mathit{exec}^{msr}([\![P]\!])$ is normal and $S_m \to_R^* S_{n-3}$ for $R = \{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}, \text{FRESH}\}$.

Lemma 4 (Normalisation)*:* Let P be a well-formed ground process. If

$$S_0 = \emptyset \xrightarrow{E_1}_{[\![P]\!]} S_1 \xrightarrow{E_2}_{[\![P]\!]} \ldots \xrightarrow{E_n}_{[\![P]\!]} S_n \in \mathit{exec}^{msr}([\![P]\!])$$

and $[E_1, \ldots, E_n] \vDash \alpha$, then there exists a normal MSR execution

$$T_0 = \emptyset \xrightarrow{F_1}_{[\![P]\!]} T_1 \xrightarrow{F_2}_{[\![P]\!]} \ldots \xrightarrow{F_{n'}}_{[\![P]\!]} T_{n'} \in \mathit{exec}^{msr}([\![P]\!])$$

such that $\mathit{hide}([E_1, \ldots, E_n]) = \mathit{hide}(F_1, \ldots, F_{n'})$ and $[F_1, \ldots, F_{n'}] \vDash \alpha$.

Proof. We will modify $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_n}_{[\![P]\!]} S_n$ by applying one transformation after the other, each resulting in an MSR execution that still fulfills the conditions on its trace.

1. If an application of the INIT rule appears in $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_n}_{[\![P]\!]} S_n$, we move it to the front. Therefore, $S_1 = \mathsf{state}_{[]}()$. This is possible since the left-hand side of the INIT rule is empty. If the rule is never instantiated, we prepend it to the trace. Since $\mathsf{Init}() \in \mathcal{F}_{res}$, the resulting MSR execution

   $$S_0^{(1)} \xrightarrow{E_1^{(1)}}_{[\![P]\!]} \ldots \xrightarrow{E_n^{(1)}}_{[\![P]\!]} S_{n^{(1)}}^{(1)}$$

   is such that $\mathit{hide}([E_1, \ldots, E_n]) = \mathit{hide}([E_1^{(1)}, \ldots, E_{n^{(1)}}^{(1)}])$. Since $\mathsf{Init}()$ is only added if it was not present before, $[E_1^{(1)}, \ldots, E_{n^{(1)}}^{(1)}] \vDash \alpha$, especially α_{init}.

2. For each fact $\mathsf{Ack}(t_1, t_2)$ contained in $S_{n^{(1)}}^{(1)}$, it also contains a fact $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t})$ for some p and $\tilde{t}$ such that there exists a rule of type

R_3 that consumes both of them, since $\mathsf{Ack}(t_1, t_2)$ can only be produced by a rule of type R_2 which consumes $\mathsf{Msg}(t_1, t_2)$ which in turn can only be produced along with a fact $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t})$, and by definition of $[\![P]\!]$, there exists a rule in $[\![P]\!]_{=p}$ of form R_3 that consumes $\mathsf{Ack}(t_1, t_2)$ and $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t})$. We append as many applications of rules of type R_3 as there are facts $\mathsf{Ack}(t_1, t_2) \in S^{(1)}_{n^{(1)}}$, and repeat this for all t_1, t_2 such that $\mathsf{Ack}(t_1, t_2) \in S^{(1)}_{n^{(1)}}$. Then, $S^{(1)}_{n^{(1)}} \to_{[\![P]\!]} S^{(1)}_{n'}$ and $S^{(1)}_{n'}$ does not contain Ack-facts anymore.

If $S^{(1)}_{n'}$ contains a fact $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t})$, we remove the last transition that produced this fact, i.e., for i such that $S_i = S_{i-1} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\}^{\#} \cup^{\#} \{\mathsf{Msg}(t_1, t_2), \mathsf{state}_p^{\mathsf{semi}}(\tilde{t})\}^{\#}$, we define $S^{(1)'}_j = S^{(1)}_j$ if $j \leqslant i-1$ and $S^{(1)'}_j = S^{(1)}_{j+1} \setminus^{\#} \{\mathsf{Msg}(t_1, t_2), \mathsf{state}_p^{\mathsf{semi}}(\tilde{t})\}^{\#} \cup^{\#} \{\mathsf{state}_p(\tilde{t})\}^{\#}$ if $i-1 < j < n'$. The resulting execution is valid, since $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t}) \in S^{(1)}_{n'}$ and since $\mathsf{Msg}(t_1, t_2) \in S^{(1)}_{n'}$. The latter is the case because if $\mathsf{Msg}(t_1, t_2)$ would be consumned at a later point, say j, $j+1$ would contain $\mathsf{Ack}(t_1, t_2)$, but since $S^{(1)'}_{n'-1}$ does not contain Ack-facts, they can only be consumned by a rule of type R_3, which would have consumned $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t})$. We repeat this procedure for every remaining $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t}) \in S^{(1)}_{n'}$, and call the resulting trace

$$S^{(2)}_0 \xrightarrow{E^{(2)}_1}_{[\![P]\!]} \dots \xrightarrow{E^{(2)}_n}_{[\![P]\!]} S^{(2)}_{n^{(2)}}$$

Since no rule added or removed or removed has an action, $\mathit{hide}([E_1, \dots, E_n]) = \mathit{hide}([E^{(2)}_1, \dots, E^{(2)}_{n^{(2)}}])$ and $[E^{(2)}_1, \dots, E^{(2)}_{n^{(2)}}] \vDash \alpha$.

3. We transform $S^{(1)}_0 \xrightarrow{E^{(1)}_1}_{[\![P]\!]} \dots \xrightarrow{E^{(1)}_n}_{[\![P]\!]} S^{(1)}_{n^{(1)}}$ as follows (all equalities are modulo E): Let us call instances of R_1, R_2 or R_3 that appear outside a chain

$$S_{i-3} \to_{R_1} S_{i-2} \to_{R_2} S_{i-1} \to_{R_3} S_i$$

for some i $t_1, t_2 \in \mathcal{M}$ "unmarked". Do the following for the smallest i that is an unmarked instance of R_3 (we will call the instance of R_3 ri_3 and suppose it is applied from S_{i-1} to S_i): Apply ri_3 after $j < i$ such that S_{j-1} to S_j is the first unmarked instance of R_2, for some q and $\tilde{y}$,i.e., this instance produces a fact $\mathsf{state}_{q\cdot 1}(\tilde{y}, \tilde{y}')$ and a fact $\mathsf{Ack}(t_1, t_2)$. Since there is no rule between j and i that might consume $\mathsf{Ack}(t_1, t_2)$ (only rules of form R_3 do, and ri_3 is the first unmarked instance of such a rule) and since ri_3 does not consume $\mathsf{state}_{q\cdot 1}(\tilde{y}, \tilde{y}')$, we can move ri_3 between j and $j+1$, adding the conclusions of ri_3 and removing the premises of ri_3 from every $S_{j+1}, \dots, S_i$. Note that unmarked instances of R_2 and R_3 are guaranteed to be preceeded by a marked R_1, and therefore only remove facts of form

$\mathsf{Ack}(\ldots)$ or $\mathsf{Msg}(\ldots)$ that have been added in that preceeding step. Since the transition at step j requires a fact $\mathsf{Msg}(t_1, t_2)$, there is an instance of R_1 prior to j, say at $k < j$, since only rules of form R_1 produces facts labelled $\mathsf{Msg}(t_1, t_2)$. Since ri_3 is now applied from Sj to S_{j+1}, we have that an instance ri_1 of a rule of form R_1 that produces $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t})$ must appear before j, i. e., $ri_1 \in \mathit{ginsts}([\![P]\!]_{=p})$. Therefore, it produces a fact $\mathsf{Msg}(t_1, t_2)$ indeed. We choose the largest k that has an unmarked R_1 which produces $\mathsf{Msg}(t_1, t_2)$ and $\mathsf{state}_p^{\mathsf{semi}}(\tilde{t})$ and move it right before j, resulting in the following MSR execution:

$$S_t^{(1)\prime} := \begin{cases} S_t^{(1)} & \text{if } t < k \\ S_{t+1}^{(1)} \cup^{\#} \{\, \mathsf{Msg}(t_1, t_2), \mathsf{state}_p^{\mathsf{semi}}(\tilde{t}) \,\}^{\#} \\ \qquad \setminus^{\#} \{\, \mathsf{state}_p(\tilde{t}) \,\}^{\#} & \text{if } k \leqslant t < j-1 \\ S_{(t)}^{(1)} & \text{if } j-1 \leqslant t < j+1 \\ S_{(t-1)}^{(1)} \cup^{\#} \{\, \mathsf{state}_{p\cdot 1}(\tilde{t}) \,\}^{\#} \\ \qquad \setminus^{\#} \{\, \mathsf{state}_p^{\mathsf{semi}}(\tilde{t}), \mathsf{Ack}(t_1, t_2) \,\}^{\#} & \text{if } j+1 \leqslant t < i+1 \\ S_t^{(1)} & \text{if } i+1 \leqslant t \end{cases}$$

We apply this procedure until it reaches a fixpoint and call the resulting trace

$$S_0^{(3)} \xrightarrow{E_1^{(3)}}_{[\![P]\!]} \cdots \xrightarrow{E_n^{(3)}}_{[\![P]\!]} S_{n^{(3)}}^{(3)}$$

No rule that has an action moved during the procedure, hence we can conclude $\mathit{hide}([E_1, \ldots, E_n]) = \mathit{hide}([E_1^{(3)}, \ldots, E_{n^{(3)}}^{(3)}])$, as well as $[E_1^{(3)}, \ldots, E_{n^{(3)}}^{(3)}] \vDash \alpha$.

4. If the last transition is in the set $\{\,$MDOut, MDPub, MDFresh, MDAppl, Fresh$\,\}$, we remove it. Repeat until fixpoint is reached and call the resulting trace

$$S_0^{(4)} \xrightarrow{E_1^{(4)}}_{[\![P]\!]} \cdots \xrightarrow{E_n^{(4)}}_{[\![P]\!]} S_{n^{(4)}}^{(4)}$$

Since this procedure removes no rule that is annotated with an action, $\mathit{hide}([E_1, \ldots, E_n]) = \mathit{hide}([E_1^{(4)}, \ldots, E_{n^{(4)}}^{(4)}])$ and $[E_1^{(4)}, \ldots, E_{n^{(4)}}^{(4)}] \vDash \alpha$.

5. If there is $\mathsf{In}(t) \in S_{n^{(4)}-1}^{(4)}$, then there is a transition where $\mathsf{In}(t)$ is produced and never consumned until $n^{(4)} - 1$. The only rule producing $\mathsf{In}(t)$ is MDIn. We can move this transition to just before $n^{(4)} - 1$ and call the resulting trace

$$S_0^{(5)} \xrightarrow{E_1^{(5)}}_{[\![P]\!]} \cdots \xrightarrow{E_n^{(5)}}_{[\![P]\!]} S_{n^{(5)}}^{(5)}$$

Since $[E_1^{(4)}, \ldots, E_{n^{(4)}}^{(4)}] \vDash \alpha$, especially α_{inEv}, there are only action in $\mathcal{F}_{res}$ between the abovementioned instance of MDIn and $n^{(4)}$. Therefore, $hide([E_1, \ldots, E_n]) = hide([E_1^{(5)}, \ldots, E_{n^{(5)}}^{(5)}])$ holds. Since α_{inEv} is the only part of α that mentions K, and since the tranformation preserved α_{inEv}, we have that $[E_1^{(5)}, \ldots, E_{n^{(5)}}^{(5)}] \vDash \alpha$.

6. We will show that 6 and 7 hold for

$$S_0^{(5)} \xrightarrow{E_1^{(5)}}_{[\![P]\!]} \cdots \xrightarrow{E_n^{(5)}}_{[\![P]\!]} S_{n^{(5)}}^{(5)}$$

in one step.

If $n^{(5)} \geqslant 2$ and there is no Ack-fact in $S_{n^{(5)-1}}^{(5)} \setminus S_{n^{(5)}}^{(5)}$, then we chose the largest $m < n$ such that $S_{m-1}^{(5)} \xrightarrow{E_m^{(5)}}_{[\![P,[],[]]\!],\text{Init,MDIn}} S_m^{(5)}$, or, if there is an Ack-fact in $S_{n^{(5)-1}}^{(5)} \setminus S_{n^{(5)}}^{(5)}$, we will chose the largest $m' < n-2$ such that $S_{m'-1}^{(5)} \xrightarrow{E_{m'}^{(5)}}_{[\![P,[],[]]\!],\text{Init,MDIn}} S_{m'}^{(5)}$.

This trivially fulfills 4. $S_m^{(5)} \to_R^* S_{n^{(5)}}^{(5)}$ and $S_{m'}^{(5)} \to_R^* S_{n^{(5)-3}}^{(5)}$, since otherwise there would be a larger m or m'. This also implies 2, as none of the rules in $R = \{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl}, \text{Fresh}\}$ remove Ack- or $\text{state}^{\text{semi}}$-facts, and the chain of rules R_1, R_2, R_3 consumes as many as it produces. Thus, if they where in $S_m^{(5)}$, they would be in $S_{n^{(5)}}^{(5)}$, too. Since $n > 2$, $m > 1$, and therefore 1. 3 holds for all parts of the trace, and therefore also for the m prefix. Similar for 5.

Since we can literally apply the same argument for the largest $\tilde{m} < m$ such that $S_{m-1}^{(5)} \xrightarrow{E_m^{(5)}}_{[\![P,[],[]]\!],\text{Init,MDIn}} S_m^{(5)}$, or, in case that there is an Ack-fact in $S_{m-1}^{(5)} \setminus S_m^{(5)}$, for the largest $\tilde{m} < m-2$, can show that 6 and 7 hold for the trace up to m or m', concluding it is normal.

□

Remark 4: Note that $\leftrightsquigarrow_P$ has the following properties (by the fact that it defines a bijection between multisets).

- If $\mathcal{P}_1 \leftrightsquigarrow_P S_1$ and $\mathcal{P}_2 \leftrightsquigarrow_P S_2$ then $\mathcal{P}_1 \cup^\# \mathcal{P}_2 \leftrightsquigarrow_P S_1 \cup^\# S_2$.
- If $\mathcal{P}_1 \leftrightsquigarrow_P S_1$ and $Q \leftrightsquigarrow_P \text{state}_p(\tilde{t})$ for $Q \in \mathcal{P}_1$ and $\text{state}_p(\tilde{t}) \in S_1$ (i.e. Q and $\text{state}_p(\tilde{t})$ are related by the bijection defined by $\mathcal{P}_1 \leftrightsquigarrow_P S_1$) then $\mathcal{P}_1 \setminus^\# \{Q\} \leftrightsquigarrow_P S_1 \setminus^\# \{\text{state}_p(\tilde{t})\}$.

Lemma 5: Le P be a well-formed ground process. If

$$S_0 = \emptyset \xrightarrow{E_1}_{[\![P]\!]} S_1 \xrightarrow{E_2}_{[\![P]\!]} \cdots \xrightarrow{E_n}_{[\![P]\!]} S_n \in exec^{msr}([\![P]\!])$$

is normal and $[E_1, \ldots, E_n] \vDash \alpha$ (see Definition 33 and 15), then there are $(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0), \ldots, (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}_{n'}^{MS}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'})$ and F_1, $\ldots, F_{n'}$ such that:

$$(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) \xrightarrow{F_1} (\mathcal{E}_1, \mathcal{S}_1, \mathcal{S}_1^{MS}, \mathcal{P}_1, \sigma_1, \mathcal{L}_1) \xrightarrow{F_2} \ldots \xrightarrow{F_{n'}} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}_{n'}^{MS}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'})$$

where $(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) = (\emptyset, \emptyset, \emptyset, \{P\}, \emptyset, \emptyset)$ and there exists a monotonically increasing, surjective function $f\colon \mathbb{N}_n \setminus \{0\} \to \mathbb{N}_{n'}$ such that $f(n) = n'$ and for all $i \in \mathbb{N}_n$

1. $\mathcal{E}_{f(i)} = \{ a \in FN \mid ProtoNonce(a) \in_E \bigcup_{1 \leqslant j \leqslant i} E_j \}$
2. $\forall\, t \in \mathcal{M}.\, \mathcal{S}_{f(i)}(t) = \begin{cases} u & \text{if } \exists j \leqslant i. Insert(t,u) \in_E E_j \\ & \quad \wedge \forall j', u'. \\ & \qquad j < j' \leqslant i \Rightarrow Insert(t,u') \notin_E E_{j'} \\ & \qquad \wedge Delete(t) \notin_E E_{j'} \\ \bot & \text{otherwise} \end{cases}$
3. $\mathcal{S}_{f(i)}^{MS} =_E S_i \setminus^{\#} \mathcal{F}_{res}$
4. $\mathcal{P}_{f(i)} \leftrightsquigarrow_P S_i$
5. $\{ x\sigma_{f(i)} \mid x \in \mathbf{D}(\sigma_{f(i)}) \}^{\#} =_E \{ Out(t) \in \cup_{k \leqslant i} S_k \}^{\#}$
6. $\mathcal{L}_{f(i)} =_E \{ t \mid \exists j \leqslant i, u.\, Lock(u,t) \in_E E_j \text{ and } \forall j < k \leqslant i.\ Unlock(u,t) \notin_E E_k \}$.

Furthermore,

7. $hide([E_1, \ldots, E_n]) =_E [F_1, \ldots, F'_n]$.

Proof. We proceed by induction over the number of transitions n.

Base Case. A normal MSR execution contains at least an application of the init rule, thereby the shortest normal MSR execution is

$$\emptyset \rightarrow_{[\![P]\!]} S_1 = \{ state_{[]}() \}^{\#}$$

We chose $n' = 0$ and thus

$$(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) = (\emptyset, \emptyset, \emptyset, \{P\}^{\#}, \emptyset, \emptyset).$$

We define $f : \{1\} \to \{0\}$ such that $f(1) = 0$.

To show that Condition 4 holds, we have to show that $\mathcal{P}_0 \leftrightsquigarrow_P \{ state_{[]}(s : fresh) \}^{\#}$. Note that $\mathcal{P}_0 = \{P\}^{\#}$. We choose the bijection such that $P \leftrightsquigarrow_P state_{[]}(s : fresh)$.

By Definition 18, $[\![P]\!]_{=[]} = [\![P, [], []]\!]_{=[]}$. We see from Figure 14 that for every P we have that $state_{[]}(s : fresh) \in prems(R\theta)$, for $R \in$

$[\![P, [], []]\!]_{=[]}$ and $\theta = \emptyset$. This induces $\tau = \emptyset$ and $\rho = \emptyset$. Since $P|_{[]}\tau\rho = P$, we have $P \leftrightsquigarrow_P \mathsf{state}_{[]}()$, and therefore $\mathcal{P}_0 \leftrightsquigarrow_P S_1$.

Condition 1, Condition 2, Condition 3, Condition 5, Condition 6, and Condition 7 hold trivially.

Inductive step. Assume the invariant holds for $n-1 \geqslant 1$. We have to show that the lemma holds for n transitions, i. e., we assume that

$$\emptyset \xrightarrow{E_1}_{[\![P]\!]} S_1 \xrightarrow{E_2}_{[\![P]\!]} \dots \xrightarrow{E_n}_{[\![P]\!]} S_n \in \mathit{exec}^{msr}([\![P]\!])$$

is normal and $[E_1, \dots, E_n] \vDash \alpha$. Then it is to show that there is

$$(\mathcal{E}_0, \mathcal{S}_0, \mathcal{S}_0^{MS}, \mathcal{P}_0, \sigma_0, \mathcal{L}_0) \xrightarrow{F_1} (\mathcal{E}_1, \mathcal{S}_1, \mathcal{S}_1^{MS}, \mathcal{P}_1, \sigma_1, \mathcal{L}_1) \xrightarrow{F_2} \dots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}_{n'}^{MS}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'})$$

fulfilling Conditions 1 to 8.

Assume now for the following argument, that there is not fact with the symbol Ack in $S_{n-1} \setminus^{\#} S_n$. This is the case for all cases except for the case where rule instance applied from S_{n-1} to S_n has the form $ri = [\mathsf{state}_p^{\mathsf{semi}}(\tilde{s}), \mathsf{Ack}(t_1, t_2)] \dashv\!\vdash [\mathsf{state}_{p \cdot 1}(\tilde{s})]$. This case will require a similiar, but different argument, which we will present when we come to this case.

Since $\emptyset \xrightarrow{E_1}_{[\![P]\!]} \cdots \xrightarrow{E_n}_{[\![P]\!]} S_n \in \mathit{exec}^{msr}([\![P]\!])$ is normal and $n \geqslant 2$, there exists an $m < n$ such that $S_m \to_R^* S_n$ for $R = \{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl}, \text{Fresh}\}$ and $\emptyset \xrightarrow{E_1}_{[\![P]\!]} \cdots \xrightarrow{E_m}_{[\![P]\!]} S_m \in \mathit{exec}^{msr}([\![P]\!])$ is normal, too. This allows us to apply the induction hypothesis on $\emptyset \xrightarrow{E_1}_{[\![P]\!]} \cdots \xrightarrow{E_m}_{[\![P]\!]} S_m \in \mathit{exec}^{msr}([\![P]\!])$. Hence there is a monotonically increasing function from $\mathbb{N}_m \to \mathbb{N}_{n'}$ and an execution such that Conditions 1 to 8 hold. Let f_p be this function and note that $n' = f_p(m)$.

In the following case distinction, we will (unless stated otherwise) extend the previous execution by one step from $(\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}_{n'}^{MS}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'})$ to $(\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}_{n'+1}^{MS}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$, showing Conditions 1 to 7 for $n'+1$. By induction hypothesis, they hold for all $i \leqslant n'$. We define a function $f\colon \mathbb{N}_n \to \mathbb{N}_{n'+1}$ as follows:

$$f(i) := \begin{cases} f_p(i) & \text{if } i \in \mathbb{N}_m \\ n' & \text{if } m < i < n \\ n'+1 & \text{if } i = n \end{cases} \tag{5}$$

Since, $S_m \to_R^* S_n$ for $R = \{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl}, \text{Fresh}\}$, only $S_n \setminus^{\#} S_m$ contains only Fr-facts and !K-facts, and $S_m \setminus^{\#} S_n$ contains only Fr-facts and Out-facts. Therefore, Condition 3,4 and 5 hold for all $i \leqslant n-1$. Since $E_{m+1}, \dots, E_{n-1} = \emptyset$, Condition 1, 2,6 and 7 hold for all $i \leqslant n-1$.

Fix a bijection such that $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$. We will abuse notation by writing $P \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$, if this bijection maps P to $\mathsf{state}_p(\tilde{t})$.

We now proceed by case distinction over the last type of transition from S_{n-1} to S_n. Let $l_{linear} =_E S_{n-1} \setminus S_n$ and $r =_E S_n \setminus S_{n-1}$. l_{linear} can only contain linear facts, while r can contain linear as well as persistent facts. The rule instance ri used to go from S_{n-1} to S_n has the following form:

$$[l_{linear}, l_{persistent}] \dashv[E_n]\rightarrow r$$

for some $l_{persistent} \subset^{\#}_E S_{n-1}$.

Note that l_{linear}, E_n and r uniquely identify which rule in $[\![P, [], []]\!]$ ri is an instance of – with exactly one exception: $[\![[] \dashv[a]\rightarrow []; P, p, \tilde{x}]\!] = [\![\text{event } a; P, p, \tilde{x}]\!]$. Luckily, we can treat the last as a special case of the first.

If R is uniquely determined, we fix some $ri \in ginsts(R)$.

Case: $R = \text{INIT}$ **or** $R \in MD \setminus \{\text{MDIN}\}$. In this case, $\emptyset \xrightarrow{E_1} \dots \xrightarrow{E_n} S_n$ is not a well-formed MSR execution.

Case: $R = \text{MDIN}$. Let $t \in \mathcal{M}$ such that $ri = R\tau = !K(t) \dashv[K(t)]\rightarrow In(t)$.

From the induction hypothesis, and since $E_{m+1}, \dots, E_n = \emptyset$, we have that

$$\mathcal{E}_{n'} = \{\, a \in FN \mid ProtoNonce(a) \in_E \bigcup_{1 \leqslant j \leqslant n} E_j \,\}.$$

From the induction hypothesis, and since no rule producing Out-facts is applied between step m and step n, we have that

$$\{\, x\sigma_{n'} \mid x \in \mathbf{D}(\sigma_{n'}) \,\}^{\#} =_E \{\, Out(t) \in \cup_{k \leqslant n} S_k \,\}^{\#}.$$

Let $\tilde{r} = \{\, a \in FN \mid RepNonce(a) \in_E \bigcup_{1 \leqslant j \leqslant n} F_j \,\}$. Then, by Lemma 2 and Lemma 15, we have that $\nu\mathcal{E}_{n'}, \tilde{r}.\sigma_{n'} \vdash t$. Therefore, $\nu\mathcal{E}_{n'}.\sigma_{n'} \vdash t$. This allows us to chose the following transition:

$$\dots \xrightarrow{F_{n'}} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \xrightarrow{K(t)} (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $(\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$ equal to $(\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'})$.

Conditions 1 to 8 hold trivially.

Case: $ri = [state_p(\tilde{t})] \dashv[] \rightarrow []$ **(for some p and $\tilde{t}$).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P state_p(\tilde{t})$. Let θ be a grounding substitution for $state_p(\tilde{x}) \in prems([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = 0$.

We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \xrightarrow{K(t)} (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{0\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

Condition 4 holds since $Q \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{0\}^{\#}$ and $S_n = S_{n-1} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\}^{\#}$. Conditions 1, 2, 3, 5 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \dashv\!\vdash [\mathsf{state}_{p\cdot 1}(\tilde{t}), \mathsf{state}_{p\cdot 2}(\tilde{t})]$ **(for some p and $\tilde{t}$).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = Q_1|Q_2$, for some processes $Q_1 = P|_{p\cdot 1}\tau\rho$ and $Q_2 = P|_{p\cdot 2}\tau\rho$.

We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{Q_1 \mid Q_2\}^{\#} \cup^{\#} \{Q_1, Q_2\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q_1 \leftrightarrow \mathsf{state}_{p\cdot 1}(\tilde{t})$ and $Q_2 \leftrightarrow \mathsf{state}_{p\cdot 2}(\tilde{t})$. Therefore, and since $Q \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{Q_1 \mid Q_2\}^{\#} \cup^{\#} \{Q_1, Q_2\}^{\#}$, and $S_n = S_{n-1} \setminus^{\#} \{\mathsf{state}_p \tilde{t}\}^{\#} \cup^{\#} \{\mathsf{state}_{p\cdot 1}(\tilde{t}), \mathsf{state}_{p\cdot 2}(\tilde{t})\}^{\#}$, Condition 4 holds.

Conditions 1, 2, 3, 5 and 7 hold trivially.

Case: $ri = [!\mathsf{state}_p(\tilde{t})] \dashv\!\vdash [\mathsf{state}_{p\cdot 1}(\tilde{t})]$ **(for some $p, \tilde{t}$).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = !Q'$for a process $Q' = P|_{p\cdot 1}\tau\rho$..

We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \\ \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \cup^{\#} \{ Q' \}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q' \leftrightsquigarrow_p \mathsf{state}_{p.1}(\tilde{t})$. Therefore, and since $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \cup^{\#} \{ Q' \}^{\#}$, while $S_n = S_{n-1} \cup^{\#} \{ \mathsf{state}_{p.1}(\tilde{t}) \}^{\#}$, Condition 4 holds.

Conditions 1, 2, 3, 5 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t}), \mathsf{Fr}(a' : \mathit{fresh})] \; -[\mathit{ProtoNonce}(a' : \mathit{fresh})]\rightarrow \; [\mathsf{state}_{p.1}(\tilde{t}, a' : \mathit{fresh})]$ **(for some** $p, \tilde{t}$ **and** $a' \in \mathsf{FN}$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_p S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_p S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_p \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p \tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p \tau\rho$, we can deduce that $Q = \nu\, a;\, Q'$ for a name $a \in \mathsf{FN}$ and a process $Q' = P|_{p.1}\tau\rho$.

By definition of exec^{msr}, the fact $\mathsf{Fr}(a')$ can only be produced once. Since this fact is linear it can only be consumed once. Every rule in $[\![P]\!]$ that produces a label $\mathit{ProtoNonce}(x)$ for some x consumes a fact $\mathsf{Fr}(x)$. Therefore,

$$a' \notin \{ a \in \mathit{FN} \mid \mathit{ProtoNonce}(a) \in_E \bigcup_{1 \leqslant j \leqslant n-1} E_j \}.$$

The induction hypothesis allows us to conclude that $a' \notin \mathcal{E}_{n'}$,i. e., a' is fresh. We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \\ \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'} \cup a'$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{ \nu\, a; Q' \}^{\#} \cup^{\#} \{ Q'\{^{a}/_{a'}\} \}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$, $\mathsf{state}_{p.1}(\tilde{x}, a) \in \mathit{prems}(R')$ for an $R' \in [\![P]\!]_{=p.1}$. We can choose $\theta' := \theta[n_a \mapsto a']$. Let $\mathsf{state}_{p.1}(\tilde{t}, a') = \mathsf{state}_{p.1}(\tilde{x}, a)\theta'$. Since $Q = P|_p \tau\rho$ for τ and ρ induced by θ, $Q'\{^{a'}/_{a}\} = P|_p \tau'\rho'$ for τ' and ρ' induced by θ', i. e., $\tau' = \tau$ and $\rho' = \rho[a \mapsto a']$. Therefore, $Q'\{^{a'}/_{a}\} \leftrightsquigarrow_p \mathsf{state}_{p.1}(\tilde{t}, a')$.

Condition 4 holds, since furthermore

$$\nu\ a'; Q' \leftrightarrow \mathsf{state}_p(\tilde{t}),$$
$$\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\nu\ a'; Q'\}^{\#} \cup^{\#} \{Q'\{^{a'}/_{a}\}\}^{\#}$$

and

$$S_n = S_{n-1} \setminus^{\#} \{\mathsf{Fr}(a), \mathsf{state}_p(\tilde{t})\}^{\#} \cup^{\#} \mathsf{state}_{p\cdot 1}(\tilde{t}, a\colon \mathit{fresh}).$$

Condition 1, holds since $\mathcal{E}_{n'+1} = \mathcal{E}_{n'} \cup a'$ and $E_n = \mathit{ProtoNonce}(a')$. Condition 7 holds since $\mathit{ProtoNonce}(a) \in \mathcal{F}_{\mathit{res}}$.

Conditions 2, 3 and 5 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t}), \mathsf{In}(t_1)] -[\mathsf{InEvent}(t_1)]\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{t}), \mathsf{Out}(t_2)]$ **(for some** $p, \tilde{t}$ **and** $t_1, t_2 \in \mathcal{M}$**).** Since the MSR execution is normal, we have that $S_{n-2} \xrightarrow{K(t_1)}_{\text{MDIN}} S_{n-1}$. Since $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_n}_{[\![P]\!]} S_n$ is normal, so is $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_{n-1}}_{[\![P]\!]} S_{n-1}$, and therefore $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_{n-2}}_{[\![P]\!]} S_{n-2}$. Hence there is an $m < n-2$ such $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_m}_{[\![P]\!]} S_m$ is a normal trace and $S_m \rightarrow_R^* S_{n-1}$ for $R = \{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}, \text{FRESH}\}$.

By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, since the rules in $\{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}, \text{FRESH}\}$ neither add nor remove state-facts, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-2}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$ and θ be a grounding substitution for $\mathsf{state}(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = \mathsf{out}\ (t_1, t_2); Q'$ for a process $Q' = P|_{p\cdot 1}\tau\rho$.

From the induction hypothesis, and since $E_{m+1}, \ldots, E_{n-2} = \emptyset$, we have that

$$\mathcal{E}_{n'} = \{a \in \mathit{FN} \mid \mathit{ProtoNonce}(a) \in_E \bigcup_{1 \leqslant j \leqslant n-2} E_j\}.$$

From the induction hypothesis, and since no rule producing Out-facts is applied between step m and step $n-2$, we have that

$$\{x\sigma_{n'} \mid x \in \mathbf{D}(\sigma_{n'})\}^{\#} =_E \{\mathsf{Out}(t) \in \cup_{k \leqslant n-2} S_k\}^{\#}. \tag{6}$$

Let $\tilde{r} = \{a : \mathit{fresh} \mid \mathit{RepNonce}(a) \in \bigcup_{1 \leqslant j \leqslant n-2} F_j\}$. Since $!\mathsf{K}(t_1) \in \mathit{prems}(\text{MDIN}\sigma)$ for $\sigma(x) = t_1$, we have $!\mathsf{K}(t) \in_E S_{n-2}$. By Lemma 2 and Lemma 15, we have $\nu\mathcal{E}_{n'}, \tilde{r}.\sigma_{n'} \vdash t$. Therefore, $\nu\mathcal{E}_{n'}.\sigma_{n'} \vdash t$. We chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{\text{MS}}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \xrightarrow{K(t_1)} (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{\text{MS}}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\mathsf{out}(t_1, t_2); Q'\}^{\#} \cup^{\#} \{Q'\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'} \cup \{{}^{t_2}/_{x}\}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$ for a fresh x.

We define f as follows:

$$f(i) := \begin{cases} f_p(i) & \text{if } i \in \mathbb{N}_m \\ n' & \text{if } m < i < n-1 \\ n'+1 & \text{if } i = n \end{cases}$$

Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

Condition 7 holds since $E_{n-1} = K(t_1)$ which implies $\mathit{hide}([E_1, \ldots, E_m]) =_E [F_1, \ldots, n']$ and $[E_{m+1}, \ldots, E_{n-1}] =_E [F_{n'+1}]$.

Condition 5 holds since $\sigma_{n'+1} = \sigma_{n'} \cup \{{}^{t_2}/_{x}\}$, and therefore:

$$\begin{aligned} \{x\sigma_{n'+1} \mid x \in \mathbf{D}(\sigma_{n'+1})\}^{\#} &= \{x\sigma_{n'} \mid x \in \mathbf{D}(\sigma_{n'})\}^{\#} \cup^{\#} \{t_2\}^{\#} \\ &=_E \{\mathsf{Out}(t) \in \cup_{k \leqslant n-2} S_k\}^{\#} \cup^{\#} \{t_2\}^{\#} \qquad \text{(by (6))} \\ &= \{\mathsf{Out}(t) \in \cup_{k \leqslant n} S_k\}^{\#} \end{aligned}$$

By definition of $[\![P]\!]$ and $[\![P]\!]_{=_p}$, we have that $Q' \leftrightsquigarrow_p \mathsf{state}_{p.1}(\tilde{t})$. Therefore, and since $\mathsf{out}(t_1, t_2); Q' \leftrightsquigarrow_p \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\mathsf{out}(t_1, t_2); Q'\}^{\#} \cup^{\#} \{Q'\}^{\#}$, and $S_n =_E S_{n-1} \setminus^{\#} \{\mathsf{In}(a), \mathsf{state}_p(\tilde{t})\}^{\#} \cup^{\#} \{\mathsf{state}_{p.1}(\tilde{t}), \mathsf{Out}(t_2)\}$, Condition 4 holds.

Conditions Condition 1, 2 and 3 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t}), \mathsf{In}(< t_1, t_2 >)] -[\mathsf{InEvent}(\langle t_1, t_2 \rangle)]\rightarrow [\mathsf{state}_{p.1}(\tilde{t}, \tilde{t}')]$ **(for some** $p, \tilde{t}, \tilde{t}'$ **and** $t_1, t_2 \in \mathcal{M}$**).** Since the MSR execution is normal, we have that $S_{n-2} \xrightarrow{K(t_1)}_{\text{MDIN}} S_{n-1}$. Since $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_n}_{[\![P]\!]} S_n$ is normal, so is $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_{n-1}}_{[\![P]\!]} S_{n-1}$, and therefore $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_{n-2}}_{[\![P]\!]} S_{n-2}$. Hence there is an $m < n-2$ such that $S_0 \xrightarrow{E_1}_{[\![P]\!]} \ldots \xrightarrow{E_m}_{[\![P]\!]} S_m$ is a normal trace and $S_m \rightarrow^*_R S_{n-1}$ for $R = \{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}, \text{FRESH}\}$.

By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_p S_m$. Since $\{\text{MDOUT}, \text{MDPUB}, \text{MDFRESH}, \text{MDAPPL}\}$, FRESH and MDIN do not add or remove state-facts, $\mathcal{P}_{n'} \leftrightsquigarrow_p S_{n-2}$.

Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_p \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=_p})$ such that $\tilde{t} =_E \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Definition 20). From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = \mathsf{in}(t_1, N); Q'$, for N a term that is not necessarily ground, and a process $Q' = P|_{p.1}\tau\rho$. Since $ri \in_E \mathit{ginsts}(R)$, we have that there is a substitution τ' such that $N\tau' =_E t_2$.

From the induction hypothesis, and since $E_{m+1}, \ldots, E_{n-2} = \emptyset$, we have that

$$\mathcal{E}_{n'} = \{\, a \mid \mathit{ProtoNonce}(a) \in \bigcup_{1 \leqslant j \leqslant n-2} E_j \,\}.$$

From the induction hypothesis, and since no rule producing Out-facts is applied between step m and step $n-2$, we have that

$$\{\, x\sigma_{n'} \mid x \in \mathbf{D}(\sigma_{n'}) \,\}^{\#} = \{\, \mathsf{Out}(t) \in \cup_{k \leqslant n-2} S_k \,\}^{\#}. \tag{7}$$

Let $\tilde{r} = \{\, a : \mathit{fresh} \mid \mathit{RepNonce}(a) \in \bigcup_{1 \leqslant j \leqslant n-2} F_j \,\}$. Since $!K(\langle t_1, t_2 \rangle) \in \mathit{prems}(\text{MDIN}\sigma)$ for $\sigma(x) = \langle t_1, t_2 \rangle$, we have $!K(\langle t_1, t_2 \rangle)_E \in S_{n-2}$. By Lemma 2 and Lemma 15, we have $\nu\mathcal{E}_{n'}, \tilde{r}.\sigma_{n'} \vdash \langle t_1, t_2 \rangle$. Therefore, $\nu\mathcal{E}_{n'}.\sigma_{n'} \vdash \langle t_1, t2 \rangle$. Using DEQ and DAPPL with the function symbols *fst* and *snd*, we have $\nu\mathcal{E}_{n'}.\sigma_{n'} \vdash t_1$ and $\nu\mathcal{E}_{n'}.\sigma_{n'} \vdash t2$. Therefore, we chose the following transition:

$$\cdots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \xrightarrow{K(t_1)} (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\, \mathsf{in}(t_1, N); Q' \,\}^{\#} \cup^{\#} \{\, Q'\tau' \,\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as follows:

$$f(i) := \begin{cases} f_p(i) & \text{if } i \in \mathbb{N}_m \\ n' & \text{if } m < i < n-1 \\ n'+1 & \text{if } i = n \end{cases}$$

Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

Condition 7 holds since $\mathit{hide}([E_1, \ldots, E_m]) = [F_1, \ldots, n']$ and $[E_{m+1}, \ldots, E_{n-1}] = [F_{n'+1}]$, since $E_{n-1} = K(t_1)$.

Let θ' such that $ri = \theta' R$. As established before, we have τ' such that $N\tau' =_E t_2$. By definition of $[\![P]\!]_{=p}$, we have that $\mathsf{state}_{p.1}(\tilde{t}, \tilde{t}') \in_E \mathit{ginsts}(P_{=p.1})$, and that $\theta' = \theta \cdot \tau'$. Since τ and ρ are induced by θ, θ' induces $\tau \cdot \tau'$ and the same ρ. We have that $Q'\tau' = (P|_{p.1}\tau\rho)\tau' = P|_p\tau\tau'\rho$ and therefore $Q'\tau \leftrightsquigarrow_p \mathsf{state}_{p.1}(\tilde{t}, \tilde{t}')$. Thus, and since $\mathsf{in}(t_1, N); Q' \leftrightsquigarrow_p \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\, \mathsf{in}(t_1, N); Q' \,\}^{\#} \cup^{\#} \{\, Q'\tau' \,\}^{\#}$ and $S_n = S_{n-1} \setminus^{\#} \{\, \mathsf{In}(< t_1, t_2 >), \mathsf{state}_p(\tilde{t}) \,\}^{\#} \cup^{\#} \{\, \mathsf{state}_{p.1}(\tilde{t}, \tilde{t}') \,\}^{\#}$, Condition 4 holds.

Conditions Condition 1, 2, 3 and 5 hold trivially.

Case: $ri = [\mathsf{state}^{\mathsf{semi}}_p(\tilde{s}), \mathsf{Ack}(t_1, t_2)] \dashv\!\vdash [\mathsf{state}_{p.1}(\tilde{s})]$ **(for some** $p, \tilde{t}$ **and** $t_1, t_2 \in \mathcal{M}$**).** Since the MSR execution is normal, we have that there p,q,$\tilde{x}, \tilde{y}, \tilde{y}'$ such that:

$$S_{n-3} \to_{R_1} S_{n-2} \to_{R_2} S_{n-1} \to_{R_3} S_n \quad \text{, where:}$$

- $R_1 = [\mathsf{state}_p(\tilde{x})] \to [\mathsf{Msg}(t_1, t_2), \mathsf{state}_p^{\mathsf{semi}}(\tilde{x})]$
- $R_2 = [\mathsf{state}_q(\tilde{y}), \mathsf{Msg}(t_1, t_2)] \to [\mathsf{state}_{q.1}(\tilde{y} \cup \tilde{y}'), \mathsf{Ack}(t_1, t_2)]$
- $R_3 = [\mathsf{state}_p^{\mathsf{semi}}(\tilde{x}), \mathsf{Ack}(t_1, t_2)] \to [\mathsf{state}_{p.1}(\tilde{x})]$

.

Since in this case, there is a fact with symbol Ack removed from S_{n-1} to S_n, we have to apply a different argument to apply the induction hypothesis.

Since $\emptyset \xrightarrow{E_1}_{[\![P]\!]} \cdots \xrightarrow{E_n}_{[\![P]\!]} S_n \in \mathit{exec}^{msr}([\![P]\!])$ is normal, $n \geqslant 2$, and $t_1, t_2 \in \mathcal{M}$, $\mathsf{Ack}(t_1, t_2) \in (S_{n-1} \setminus^{\#} S_n)$, there exists $m \leqslant n-3$ such that $S_m \to_R^* S_{n-3}$ for $R = \{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl}\} \cup$ Fresh and $\emptyset \xrightarrow{E_1}_{[\![P]\!]} \cdots \xrightarrow{E_m}_{[\![P]\!]} S_m \in \mathit{exec}^{msr}([\![P]\!])$ is normal. This allows us to apply the induction hypothesis on $\emptyset \xrightarrow{E_1}_{[\![P]\!]} \cdots \xrightarrow{E_m}_{[\![P]\!]} S_m \in \mathit{exec}^{msr}([\![P]\!])$. Hence there is a monotonically increasing function from $\mathbb{N}_m \to \mathbb{N}_{n'}$ and an execution such that Conditions 1 to 8 hold. Let f_p be this function and note that $n' = f_p(m)$.

In the following case distinction, we extend the previous execution by one step from $(\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{\text{MS}}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'})$ to $(\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{\text{MS}}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$, and prove that Conditions 1 to 7 hold for $n'+1$. By induction hypothesis, they hold for all $i \leqslant n'$. We define a function $f\colon \mathbb{N}_n \to \mathbb{N}_{n'+1}$ as follows:

$$f(i) := \begin{cases} f_p(i) & \text{if } i \in \mathbb{N}_m \\ n' & \text{if } m < i \leqslant n-3 \\ n'+1 & \text{if } i = n \end{cases}$$

Since, $S_m \to_R^* S_n$ for $R = \{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl},$ Fresh$\}$, only $S_n \setminus^{\#} S_m$ contains only Fr-facts and !K-facts, and $S_m \setminus^{\#} S_n$ contains only Fr-facts and Out-facts. Therefore, Condition 3, 4 and 5 hold for all $i \leqslant n-3$. Since $E_{m+1}, \ldots, E_{n-1} = \emptyset$, Condition 1, 2, 6 and 7 hold for all $i \leqslant n-3$.

Fix a bijection such that $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$. We will abuse notation by writing $P \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$, if this bijection maps P to $\mathsf{state}_p(\tilde{t})$. Since $\{\text{MDOut}, \text{MDPub}, \text{MDFresh}, \text{MDAppl}\}$ and Fresh do not add or remove state-facts, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-3}$. Let $P \in^{\#} \mathcal{P}_{n'}$ such that $P \leftrightsquigarrow_P \mathsf{state}_p(\tilde{s})$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_q(\tilde{t})$.

Let θ' be a grounding substitution for $\mathsf{state}_q(\tilde{y}) \in \mathit{prems}([\![P]\!]_{=q})$ such that $\tilde{t} =_E \tilde{y}\theta'$. Then θ' induces a substitution τ' and a bijective renaming ρ' for fresh, but not bound names (in Q) such that $P|_q\tau'\rho' = Q$ (see Definition 20).

From the form of the rules R_1 and R_3, and since $P =_E P|_p\tau\rho$, for τ and ρ induced by the grounding substitution for $\mathsf{state}_p(\tilde{x})$, we can deduce that $P =_E \text{out } t_1, t_2; P'$ for a process $P' = P|_{p.1}\tau\rho$. Similarly, from the form of R_2, we can deduce $Q =_E \text{in } (t_1, N); Q'$, for N a

term that is not necessarily ground, and a process $Q' = P|_{q\cdot 1}\tau'\rho'$. Since $S_{n-2} \to_{R_2} S_{n-1}$, we have that there is a substitution τ^* such that $N\tau'\rho'\tau^* =_E t_2$ and $((\tilde{y} \cup \mathit{vars}(N)) \setminus \tilde{y})\tau^* =_E \tilde{t}'$, where $\tilde{t}'$ such that $\mathsf{state}_{q\cdot 1}(\tilde{t}, \tilde{t}') \in S_{n-1} \setminus^{\#} S_{n-2}$.

Given that $Q =_E \mathsf{in}(t_1, N); Q'$ and $P =_E \mathsf{out}\, t_1, t_2; P'$, have that $\mathcal{P}_{n'} = \mathcal{P}' \cup^{\#} \{\mathsf{out}\, t_1, t_2; P', \mathsf{in}(t_1', N); Q'\}^{\#}$ with $t_1 =_E t_1'$ and $t_2 =_E N\tau^*$. Therefore, we chose the following transition:

$$\cdots \xrightarrow{F_n'} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \xrightarrow{K(t_1)} (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}' \cup^{\#} \{P', Q'\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

Conditions 1 to 8 hold for $i \leqslant n-3$. It is left to show that Conditions 1 to 8 hold for n.

As established before, we have τ^* such that $N\tau'\rho'\tau^* =_E t_2$. Let $\mathsf{state}_q(\tilde{t}, \tilde{t}')$ be the state variable added to S_{n-1}. Then, $((\tilde{y} \cup \mathit{vars}(N)) \setminus \tilde{y})\tau^* = \tilde{t}'$. By definition of $[\![P]\!]_{=q}$, we have that $\mathsf{state}_{q\cdot 1}(\tilde{t}, \tilde{t}') \in \mathit{prems}(\mathit{ginsts}(P_{=p\cdot 1}))$ for a grounding substitution $\theta_{q\cdot 1} = \theta' \cdot \tau^*$. Since τ' and ρ' are induced by θ', $\theta_{q\cdot 1}$ induces $\tau \cdot \tau'$ and the same ρ. We have that $Q'\tau' = (P|_{q\cdot 1}\tau'\rho')\tau* = P|_{q\cdot 1}\tau\tau'\rho$ and therefore $Q'\tau^* \leftrightsquigarrow_P \mathsf{state}_{q\cdot 1}(\tilde{t}, \tilde{t}')$. Similarly, we have $P' \leftrightsquigarrow_P \mathsf{state}_{q\cdot 1}(\tilde{s})$. We conclude that Condition 4 holds.

Conditions Condition 1, 2, 3, 5, 6 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \dashv [\mathrm{Eq}(t_1, t_2)] \to [\mathsf{state}_{p\cdot 1}(\tilde{t})]$ **(for some** $p, \tilde{t}$ **and** $t_1, t_2 \in \mathcal{M}$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = \mathtt{if}\ t_1 = t_2\ \mathtt{then}\ Q_1\ \mathtt{else}\ Q_2$ for a process $Q' = P|_{p\cdot 1}\tau\rho$.

Since, $[E_1, \ldots, E_n \vDash \alpha$, and thus $[E_1, \ldots, E_m \vDash \alpha_{eq}$, we have that $t_1 =_E t_2$. We therefore chose the following transition:

$$\cdots \xrightarrow{F_n'} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \to (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\mathtt{if}\ t_1 = t_2 \mathtt{then}\ Q_1\ \mathtt{else}\ Q_2\}^{\#} \cup^{\#} \{Q_1\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q_1 \leftrightarrow \mathsf{state}_{p\cdot 1}(\tilde{t})$. Therefore, and since $\mathtt{if}\ t_1 = t_2\ \mathtt{then}\ Q_1\ \mathtt{else}\ Q_2 \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} =$

$\mathcal{P}_{n'} \setminus^{\#} \{\, \texttt{if } t_1 = t_2 \texttt{ then } Q_1 \texttt{ else } Q_2 \,\}^{\#} \cup^{\#} \{\, Q_1 \,\}^{\#}$, and $S_n = S_{n-1} \setminus^{\#} \{\, \mathsf{state}_p(\tilde{t}) \,\}^{\#} \cup^{\#} \{\, \mathsf{state}_{p.1}(\tilde{t}) \,\}^{\#}$, Condition 4 holds. Conditions 1, 2, 3, 5, 6 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\, \mathrm{NotEq}(t_1, t_2)] \rightarrow [\mathsf{state}_{p.1}(\tilde{t})]$ **(for some** $p, \tilde{t}$ **and** $t_1, t_2 \in \mathcal{M}$**).** In this case, the proof is almost the same as in the previous case, except that α_{noteq} is the relevant axiom, Q_2 is chosen instead of Q_1 and $S_n = S_{n-1} \setminus^{\#} \{\, \mathsf{state}_p(\tilde{t}) \,\}^{\#} \cup^{\#} \{\, \mathsf{state}_{p.2}(\tilde{t}) \,\}^{\#}$.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\, F, \mathrm{Event}()] \rightarrow [\mathsf{state}_{p.1}(\tilde{t})]$ **(for some** $p, \tilde{t}$**).** This is a special case of the case where $ri = [\mathsf{state}_p(\tilde{t}), l] \dashv[\, a] \rightarrow [\mathsf{state}_{p.1}(\tilde{t}), r]$ for $l = r = \emptyset$ and $a = F$.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\mathit{Insert}(t_1, t_2)] \rightarrow [\mathsf{state}_{p.1}(\tilde{t})]$ **(for some** $p, \tilde{t}$ **and** $t_1, t_2 \in \mathcal{M}$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p \tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p \tau\rho$, we can deduce that $Q = \texttt{insert } t_1, t_2; Q'$ for a process $Q' = P|_{p.1} \tau\rho$.

We therefore chose the following transition:

$$\cdots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}[t_1 \mapsto t_2]$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\, \texttt{insert } t_1, t_2; Q' \,\}^{\#} \cup^{\#} \{\, Q' \,\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q' \leftrightarrow \mathsf{state}_{p.1}(\tilde{t})$. Therefore, and since $\texttt{insert } t_1, t_2; Q' \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\, \texttt{insert } t_1, t_2; Q' \,\}^{\#} \cup^{\#} \{\, Q' \,\}^{\#}$, as well as $S_n = S_{n-1} \setminus^{\#} \{\, \mathsf{state}_p(\tilde{t}) \,\}^{\#} \cup^{\#} \{\, \mathsf{state}_{p.1}(\tilde{t}) \,\}^{\#}$, we have that Condition 4 holds.

Condition 2 holds, since $E_n = \mathit{Insert}(t_1, t_2)$ is the last element of the trace.

Conditions 1, 3, 5, 6 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\mathit{Delete}(t_1, t_2)] \rightarrow [\mathsf{state}_{p.1}(\tilde{t})]$ **(for some** $p, \tilde{t}$ **and** $t_1, t_2 \in \mathcal{M}$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p \tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p \tau\rho$, we can deduce that $Q = \texttt{delete } t_1; Q'$ for a process $Q' = P|_{p.1} \tau\rho$.

We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \\ \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}[t_1 \mapsto t_2]$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\!|\, \texttt{delete}\ t_1; Q' \,|\!\}^{\#} \cup^{\#} \{\!|\, Q' \,|\!\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q' \leftrightarrow \mathsf{state}_{p\cdot 1}(\tilde{t})$. Therefore, and since $\texttt{delete}\ t_1; Q' \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\!|\, \texttt{delete}\ t_1; Q' \,|\!\}^{\#} \cup^{\#} \{\!|\, Q' \,|\!\}^{\#}$, as well as $S_n = S_{n-1} \setminus^{\#} \{\!|\, \mathsf{state}_p(\tilde{t}) \,|\!\}^{\#} \cup^{\#} \{\!|\, \mathsf{state}_{p\cdot 1}(\tilde{t}) \,|\!\}^{\#}$, we have that Condition 4 holds.

Condition 2 holds, since $E_n = \mathit{Delete}(t_1, t_2)$ is the last element of the trace.

Conditions 1, 3, 5, 6 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \multimap\!\![\, \mathit{IsIn}(t_1, t_2)] \rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{t}, t_2)]$ **(for some** $p, \tilde{t}$ **and** $t_1, t_2 \in \mathcal{M}$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_p S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_p S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_p \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p \tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = \texttt{lookup}\ t_1\ \texttt{as}\ v\ \texttt{in}\ Q_1\ \texttt{else}\ Q_2$ for some variable V, and two processes $Q_1 = P|_{p\cdot 1}\tau\rho$ and $Q_2 = P|_{p\cdot 2}\tau\rho$.

Since $[E_1, \ldots, E_n] \vDash \alpha_{\mathit{in}}$, there is an $i < n$ such that $\mathit{Insert}(t_1, t_2) \in_E E_i$ and there is no j such that $i < j < n$ and $\mathit{Delete}(t_1) \in_E E_j$ or and $\mathit{Insert}(t_1, t_2) \in_E TE_j$. Since $E_m, \ldots, E_n = \emptyset$, we know that $i < m$. Hence, by induction hypothesis, $\mathcal{S}_{n'}(t_1) = t_2$. We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \\ \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\!|\, \texttt{lookup}\ t_1\ \texttt{as}\ v\ \texttt{in}\ Q_1\ \texttt{else}\ Q_2 \,|\!\}^{\#} \cup^{\#} \{\!|\, Q_1\{^{t_2}/_v\} \,|\!\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q_1\{^{v}/_{t_2}\} \leftrightarrow \mathsf{state}_{p\cdot 1}(\tilde{t}, t_2)$ (for $\tau' = \tau[v \mapsto t_2]$ and $\rho' = \rho$). Therefore, and since $\texttt{lookup}\ t_1\ \texttt{as}\ v\ \texttt{in}\ Q_1\ \texttt{else}\ Q_2 \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\!|\, \texttt{lookup}\ t_1\ \texttt{as}\ v$

$\texttt{in } Q_1 \texttt{ else } Q_2\}^{\#} \cup^{\#} \{Q'\}^{\#}$, as well as $S_n = \S_{n-1} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\}^{\#} \cup^{\#} \{\mathsf{state}_{p\cdot 1}(\tilde{t}, t_2)\}^{\#}$, Condition 4 holds.

Conditions 1, 2, 3, 5, 6 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\mathit{IsNotSet}(t_1)]\rightarrow [\mathsf{state}_{p\cdot 2}(\tilde{t})]$ **(for some** $p, \tilde{t}$ **and** $t_1 \in \mathcal{M}$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = \texttt{lookup } t_1 \texttt{ as } v \texttt{ in } Q_1 \texttt{ else } Q_2$ for a variable v and two processes $Q_1 = P|_{p\cdot 1}\tau\rho$ and $Q_2 = P|_{p\cdot 2}\tau\rho$.

Since $[E_1, \ldots, E_n] \vDash \alpha_{\mathit{notin}}$, there is no $i < n$ with $\mathit{Insert}(t_1, t_2) \in_E E_i$ and there is no j such that $i < j < n$ and $\mathit{Delete}(t_1) \in_E E_j$ or and $\mathit{Insert}(t_1, t_2) \in_E TE_j$. Since $E_m, \ldots, E_n = \emptyset$, we know that holds $j < m$. Hence, by induction hypothesis, $\S_{n'}(t_1)$ is undefined. We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \S_{n'}, \S^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \rightarrow (\mathcal{E}_{n'+1}, \S_{n'+1}, \S^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\S_{n'+1} = \S_{n'}$, $\S^{MS}_{n'+1} = \S^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\texttt{lookup } t_1 \texttt{ as } v \texttt{ in } Q_1 \texttt{ else } Q_2\}^{\#} \cup^{\#} \{Q_2\}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q_2 \leftrightarrow \mathsf{state}_{p\cdot 2}(\tilde{t})$. Therefore, and since $\texttt{lookup } t_1 \texttt{ as } v \texttt{ in } Q_1 \texttt{ else } Q_2 \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{\texttt{lookup } t_1 \texttt{ as } v \texttt{ in } Q_1 \texttt{ else } Q_2\}^{\#} \cup^{\#} \{Q_2\}^{\#}$, and $S_n = \S_{n-1} \setminus^{\#} \{\mathsf{state}_p(\tilde{t})\}^{\#} \cup^{\#} \{\mathsf{state}_{p\cdot 2}(\tilde{t})\}^{\#}$, Condition 4 holds.

Conditions 1, 2, 3, 5, 6 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t}), \mathsf{Fr}(\mathit{lock}_l)] \dashv[\mathit{Lock}(\mathit{lock}_l, t)]\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{t}, \mathit{lock}_l)]$ **(for some** $p, \tilde{t}$**,** $\mathit{lock}_l \in \mathit{FN}$ **and** $t \in \mathcal{M}$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p\tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p\tau\rho$, we can deduce that $Q = \texttt{lock}^l\ t;\ Q'$ for $Q' = P|_{p\cdot 1}\tau\rho$.

Since $[E_1, \ldots, E_n] \vDash \alpha_{\mathit{lock}}$, for every $i < n$ such that $\mathit{Lock}(l_p, t) \in_E E_i$, there a j such that $i < j < n$ and $\mathit{Unlock}(l_p, t) \in_E E_j$, and in between i and j, there is no lock or unlock, i. e., for all k such that $i < k < j$, and all l_i, $\mathit{Lock}(l_i, t) \notin_E E_k$ and $\mathit{Unlock}(l_i, t) \notin_E E_k$.

Since $E_m, \ldots, E_n = \emptyset$, we know that this holds for $i < m$ and $j < m$ as well. By induction hypothesis, Condition 6, this implies that $t \notin_E \mathcal{L}_{n'}$. We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \\ \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{ \mathsf{lock}^l\ t;\ Q' \}^{\#} \cup^{\#} \{ Q' \}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'} \cup \{ t \}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q' \leftrightarrow \mathsf{state}_{p.1}(\tilde{t})$. Therefore, and since $\mathsf{lock}^l\ t;\ Q' \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{ \mathsf{lock}^l\ t;\ Q' \}^{\#} \cup^{\#} \{ Q' \}^{\#}$, and $S_n = \mathcal{S}_{n-1} \setminus^{\#} \{ \mathsf{state}_p(\tilde{t}), \mathsf{Fr}(\mathit{lock}_l) \}^{\#} \cup^{\#} \{ \mathsf{state}_{p.1}(\tilde{t}, \mathit{lock}_l) \}^{\#}$, Condition 4 holds.

Condition 6 holds since $E_n = \{ \mathrm{Lock}(\mathit{lock}_l, t) \}^{\#}$ is added to the end of the trace.

Conditions 1, 2, 3, 5 and 7 hold trivially.

Case: $ri = [\mathsf{state}_p(\tilde{t})] \dashv[\, \mathit{Unlock}(n_l, t)] \rightarrow [\mathsf{state}_{p.1}(\tilde{t})]$ **(for some** $p, \tilde{t}$**,** $n_l \in \mathit{FN}$ **and** $t \in \mathcal{M}$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_p S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_p S_{n-1}$. Let $Q \in^{\#} \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_p \mathsf{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathsf{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p \tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p \tau\rho$, we can deduce that $Q = \mathsf{unlock}^l\ t;\ Q'$ for $Q' = P|_{p.1} \tau\rho$.

We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \\ \rightarrow (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'}$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{ \mathsf{unlock}^l\ t;\ Q' \}^{\#} \cup^{\#} \{ Q' \}^{\#}$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'} \setminus \{ t \}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

By definition of $[\![P]\!]$ and $[\![P]\!]_{=p}$, we have that $Q' \leftrightarrow \mathsf{state}_{p.1}(\tilde{t})$. Therefore, and since $\mathsf{unlock}^l\ t;\ Q' \leftrightarrow \mathsf{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^{\#} \{ \mathsf{unlock}^l\ t;\ Q' \}^{\#} \cup^{\#} \{ Q' \}^{\#}$, as well as $S_n = S_{n-1} \setminus^{\#} \{ \mathsf{state}_p(\tilde{t}) \}^{\#} \cup^{\#} \{ \mathsf{state}_{p.1}(\tilde{t}) \}^{\#}$, Condition 4 holds.

We show that Condition 6 holds for $\mathcal{L}_{n'+1} = \mathcal{L}_{n'} \setminus \{ t \}$: For all $t' \neq_E t$, $t' \in_E \mathcal{L}_{n'} \Leftrightarrow t' \in_E \mathcal{L}_{n'+1}$ by induction hypothesis. If $t \notin_E \mathcal{L}_{n'}$, then $\forall j \leqslant m, u.\mathrm{Lock}(u, t) \in_E E_j \rightarrow \exists j < k \leqslant n.\mathrm{Unlock}(u, t) \in_E E_k$.

Since we have $E_m, \ldots, E_{n-1} = \emptyset$ and $E_n = \{\mathrm{Unlock}(n_l, t)\}^\#$, we can strengthen this to $\forall j \leqslant n, u.\mathrm{Lock}(u,t) \in_E E_j \to \exists j < k \leqslant n.\mathrm{Unlock}(u,t) \in_E E_k$, which means that the condition holds in this case. If $t \in_E \mathcal{L}_{n'}$, then $\exists j \leqslant n, u.\mathrm{Lock}(u,t) \in_E E_j \wedge \forall j < k \leqslant n.\mathrm{Unlock}(u,t) \notin_E E_k$ and since $E_n = \{\mathrm{Unlock}(n_l, t)\}^\#$ and $E_m, \ldots, E_{n-1} = \emptyset$, a contradiction to Condition 6 would constitute of j and $u \neq_E n_l$ such that $\mathrm{Lock}(u,t) \in_E E_j$ and $\forall j < k \leqslant n.\, \mathrm{Unlock}(u,t) \notin_E E_k$.

We will show that this leads to a contradiction with $[E_1, \ldots, E_n] \vDash \alpha$. Fix j and u. By definition of $[\![P]\!]$ and well-formedness of P, there is a p_l that is a prefix of p such that $P|_{q_l} = \mathtt{lock}^l t; Q''$ for the same annotation l and parameter t. The form of the translation guarantees that if $\mathrm{state}_p(\tilde{t}) \in S_n$, then for some $\tilde{t}'$ there is $i \leqslant n$ such that $\mathrm{state}_{p'}(\tilde{t}') \in S_i$, if p' is a prefix of p. We therefore have that there is $i < n$ such that $E_i =_E \{\mathrm{Lock}(n_l, t)\}^\#$. We proceed by case distinction: Case 1: $j < i$ (see Figure 21). Since $\forall j < k \leqslant n.\mathrm{Unlock}(u,t) \notin_E E_k$, $[E_1, \ldots, E_n] \nvDash \alpha_{lock}$.

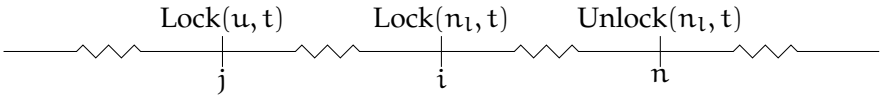


Figure 21: Visualisation of Case 1.

Case 2: $i < j$ (see Figure 22). By definition of $\overline{P}$, there is no parallel and no replication between p_l and p. Note that any rule in $[\![P]\!]$ that produces a state named state_q for a non-empty q is such that it requires a fact with name $\mathrm{state}_{q'}$ for $q = q' \cdot 1$ or $q = q' \cdot 2$ (in case of the translation of out, it might require $\mathrm{state}^{\mathsf{semi}}_{q'}$, which in turn requires $\mathrm{state}_{q'}$). Therefore, there cannot be a second $k \neq n$ such that $\mathrm{Unlock}(n_l, t) \in_E E_k$ (since n_l was added in a Fr-fact in to S_i). This means in particular that there is not k such that $i < k < n$ and $\mathrm{Unlock}(n_l, t) \in_E E_k$. Therefore, $[E_1, \ldots, E_n] \nvDash \alpha_{lock}$.

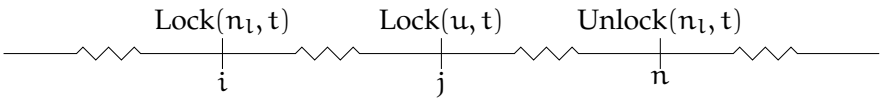


Figure 22: Visualisation of Case 2.

Conditions 1, 2, 3, 5 and 7 hold trivially.

Case: $ri = [\mathrm{state}_p(\tilde{t}), l'] \dashv[\, a', \mathrm{Event}()] \to [\mathrm{state}_{p\cdot 1}(\tilde{t}, \tilde{t}'), r']$ **(for some** $p, \tilde{t}, \tilde{t}'$ **and** $l', r', a' \in \mathcal{G}^*$**).** By induction hypothesis, we have $\mathcal{P}_{n'} \leftrightsquigarrow_P S_m$, and thus, as previously established, $\mathcal{P}_{n'} \leftrightsquigarrow_P S_{n-1}$. Let $Q \in^\# \mathcal{P}_{n'}$ such that $Q \leftrightsquigarrow_P \mathrm{state}_p(\tilde{t})$. Let θ be a grounding substitution for $\mathrm{state}_p(\tilde{x}) \in \mathit{prems}([\![P]\!]_{=p})$ such that $\tilde{t} = \tilde{x}\theta$. Then θ induces a substitution τ and a bijective renaming ρ for fresh, but not bound names (in Q) such that $P|_p \tau\rho = Q$ (see Definition 20).

From the form of the rule R, and since $Q = P|_p \tau\rho$, we can deduce that $Q = l \multimap[a]\to r; Q'$ for a process $Q' = P|_{p\cdot 1}\tau\rho$.

Since $ri \in_E ginsts(R)$, we have that there is a substitution τ^* such that $(l \multimap[a]\to r)\tau^* = l' \multimap[a]\to' r'$, $lfacts(l') \subset^\# S_{n-1}$, $pfacts(l') \subset_E S_{n-1}$ and, from the definition of $[\![P]\!]_{=p}$ $vars(l)\tau^* = \tilde{t}'$. Since P is well-formed, no fact in $\mathcal{F}_{res}$ appears in neither l nor r, so from Condition 3 in the induction hypothesis, we have that $lfacts(l') \subset^\# \mathcal{S}^{MS}_m = \mathcal{S}^{MS}_{n'}$ and $pfacts(l') \subset \mathcal{S}^{MS}_m = \mathcal{S}^{MS}_{n'}$. We therefore chose the following transition:

$$\ldots \xrightarrow{F'_n} (\mathcal{E}_{n'}, \mathcal{S}_{n'}, \mathcal{S}^{MS}_{n'}, \mathcal{P}_{n'}, \sigma_{n'}, \mathcal{L}_{n'}) \xrightarrow{a'} (\mathcal{E}_{n'+1}, \mathcal{S}_{n'+1}, \mathcal{S}^{MS}_{n'+1}, \mathcal{P}_{n'+1}, \sigma_{n'+1}, \mathcal{L}_{n'+1})$$

with $\mathcal{E}_{n'+1} = \mathcal{E}_{n'}$, $\mathcal{S}_{n'+1} = \mathcal{S}_{n'}$, $\mathcal{S}^{MS}_{n'+1} = \mathcal{S}^{MS}_{n'} \setminus lfacts(l') \cup^\# r'$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^\# \{ l \multimap[a]\to r; Q' \}^\# \cup^\# \{ Q'\tau^* \}^\#$, $\sigma_{n'+1} = \sigma_{n'}$ and $\mathcal{L}_{n'+1} = \mathcal{L}_{n'}$.

We define f as in equation 5 on page 226. Therefore, Conditions 1 to 8 hold for $i < n-1$. It is left to show that Conditions 1 to 8 hold for n.

Condition 7 holds since $E_{m+1}, \ldots, E_{n-1} = \emptyset$ and since $hide([E_1, \ldots, E_m]) = [F_1, \ldots, n']$, while $E_n = F'_n \setminus \text{Event}() = a'$ (note that $\text{Event}() \in \mathcal{F}_{res}$).

As established before, we have τ^* such that $(l \multimap[a]\to r)\tau^* =_E l \multimap[a]\to r$. By the definition of $[\![P]\!]_{=p}$, we have that $\text{state}_{p\cdot 1}(\tilde{t}, \tilde{t}') \in_E ginsts(P_{=p\cdot 1})$, and a $\theta' = \theta \cdot \tau^*$ that is grounding for $\text{state}_{p\cdot 1}(\tilde{t}, \tilde{t}')$. Since τ and ρ are induced by θ, θ' induces $\tau \cdot \tau^*$ and the same ρ. We have that $Q'\tau^* = (P|_{p\cdot 1}\tau\rho)\tau^* = P|_p\tau\tau^*\rho$ and therefore $Q'\tau \leftrightsquigarrow_P \text{state}_{p\cdot 1}(\tilde{t}, \tilde{t}')$. Thus, and since $l \multimap[a]\to r; Q' \leftrightsquigarrow_P \text{state}_p(\tilde{t})$, $\mathcal{P}_{n'+1} = \mathcal{P}_{n'} \setminus^\# \{ l \multimap[a]\to r; Q' \}^\# \cup^\# \{ Q'\tau^* \}^\#$ and $S_n = S_{n-1} \setminus^\# lfacts(l') \cup^\# r' \setminus^\# \{ \text{state}_p(\tilde{t}) \}^\# \cup^\# \{ \text{state}_{p\cdot 1}(\tilde{t}, \tilde{t}') \}^\#$, Condition 4 holds.

Condition 3, holds since

$$\begin{aligned} S_n \setminus^\# \mathcal{F}_{res} &= (S_{n-1} \setminus^\# lfacts(l') \cup^\# r' \\ &\qquad \setminus^\# \{ \text{state}_p(\tilde{t}) \}^\# \cup^\# \{ \text{state}_{p\cdot 1}(\tilde{t}, \tilde{t}') \}^\#) \setminus^\# \mathcal{F}_{res} \\ &= (S_{n-1} \setminus^\# lfacts(l') \cup^\# r') \setminus^\# \mathcal{F}_{res} \\ &= S_{n-1} \setminus^\# \mathcal{F}_{res} \setminus^\# lfacts(l') \cup^\# r' \\ &= \mathcal{S}^{MS}_{n'} \setminus^\# lfacts(l') \cup^\# r' \\ &= \mathcal{S}^{MS}_{n'+1} \end{aligned}$$

Conditions 1, 2, 5, 6 and 7 hold trivially.

□

C

INITIALISATION AND SETUP IN $\mathcal{F}_{KM}$

C.1 INITIALISATION PHASE

All regular protocol machines that shall accept messages from $\mathcal{F}_{KM}$ need to send a message to $\mathcal{F}_{KM}$ first [51, § 4.5]. A similar behaviour needs to be emulated by $\mathcal{F}_{setup}$ in the network with the actual tokens. The involved protocol machines are $\mathcal{M} := \mathcal{U} \cup \mathcal{ST} \cup \{ \mathcal{F}_1, \ldots, \mathcal{F}_l \}$, where $\mathcal{F}_i$ denotes the regular protocol machine that makes subroutine calls to $\mathcal{F}_i$, identified with the machine ID «reg, $\mathcal{F}_i$>, sid >.

We add the following part to the definition of $\mathcal{F}_{KM}$:

Listing 29: Initilisation ($\mathcal{F}_{KM}$).

```
ready-P: accept <ready> from P ∈ M
        send <ready•,P> to A
ready [ready-P ∀P ∈ M ]
```

We add the following part to the definition of ST:

Listing 30: Initilisation (ST_i).

```
ready [¬ ready]:
accept <ready> from parentId
        call F_setup with <ready>
```

C.2 HANDLING OF THE SETUP PHASE IN $\mathcal{F}_{KM}$ AND ST_i

The following listings describe the setup phase introduced on p. 120.

Listing 31: The setup phase: sharing keys ($\mathcal{F}_{KM}$).

```
share[¬finish_setup∧ready]: accept <share,h_1,U_2> from U_1 ∈ U;
        if U_i,U_2 ∈ Room
                create h_2; Store[U_2,h_2]=Store[U_1,h_1]; send <
                    share•,h_2> to U_2
```

Listing 32: The setup phase: sharing keys (ST_i).

```
share[¬finish_setup∧ready]: accept <share,h_1,U_2> from F_setup;
        if Store[U,h_1]=s call F_setup with <send,s,U_2>
import[¬finish_setup∧ready]:accept <deliver,s,U_1> from F_setup
        if U_1 ∈ Room create h_2; Store[U_i,h_2]=s; send <share•,h_2>
             to F_setup
```

Listing 33: The setup phase: terminating the setup phase ($\mathcal{F}_{KM}$).

```
finish_setup[¬finish_setup∧ready]: accept <finish_setup> from
    U ∈ 𝒰;
        send <finish_setup•> to A
```

Listing 34: The setup phase: terminating the setup phase (ST_i).

```
finish_setup[¬finish_setup∧ready]: accept <close> from ℱsetup;
        send <close•> to ℱsetup
```

C.3 SETUP ASSUMPTIONS FOR THE IMPLEMENTATION

The setup assumption used for $\mathcal{ST}$ is subsumed in the setup functionality $\mathcal{F}_{setup}$, which is defined as follows:

```
ready-Ui: accept <ready,STi> from Ui ∈ 𝒰
                send <ready•,Ui> to A
ready-P: accept <ready> from P ∈ ℳ \ 𝒰
                send <ready•,P> to A
ready [ready-P ∀P ∈ ℳ]
share[ready∧ ¬finish_setup]: accept <send,x,STj> from STi
        if Ui,Uj ∈ Room
                send <deliver,x,STi> to STj
        else
                send <⊥,STi> to Ui
finish_setup[ready∧¬finish_setup]:
        accept <finish_setup> from U ∈ 𝒰
        from i:=1 to n
                send <close> to STi; accept <close•> from STi
        send <finish_setup•> to A
relay_receive[ready]: accept <x,STi> from Ui; send <x> to STi
relay_send[ready]: accept <x> from STi;send <x,STi> to Ui
```

(The sixth line signifies that `ready` is considered true when `ready-P` is true for all $P \in \mathcal{M}$.)

D

THE COMPLETE PROOF OF EMULATION

The static call graph has only an edge from `prot-fkm` to `prot-fsetup`, $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ is thus rooted at `prot-fkm`.

Lemma 16: For all KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ is a poly-time protocol.

Proof. By Definition 2 in [51, § 6], we need to show that there exists a polynomial p such that for every well-behaved environment Z that is rooted at `prot-fkm`, we have:

$$\begin{aligned}&\Pr[\mathrm{Time}_{\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta)\\ &\quad > p(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))]\\ &\quad = \mathrm{negl}(\eta).\end{aligned}$$

Let $p_{\max}$ be a polynomial such that for all $\mathcal{F}_i$ and $C \in \mathcal{C}$, the algorithm $impl_C$ terminates in a running time smaller than $p_{\max}(n)$, where n is the length of the input. $impl^F_{\mathsf{new}}$ is always called on input of length η, thus all keys have a length smaller $p_{\max}(\eta)$. In step `new`, F and a are provided by the environment (as input to U_i, which then asks $\mathcal{F}_{\text{setup}}$ to relay the request to ST_i). `Store` grows at most by some polynomial $p_{\text{growth}-\text{new}}$ in the length of the environment's input. Similarly, an `<unwrap,...>` query cannot grow the Store by more than $p_{\text{growth}-\text{unwrap}}$. Therefore, at any point in time t (we simply count the number of epochs, i. e., activations of the environment), the store is smaller than $p'(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))$ for a polynomial p'.

We observe that there is not a single activation of a machine in $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$, neither a U_i, an ST_i nor $\mathcal{F}_{\text{setup}}$, where the running time is not polynomial in the environment's input and the length of the `Store`. A_D might corrupt user $U \in \mathcal{U} \cup \mathcal{U}^{\text{ext}}$, but they do not have any state. Thus, we have for the running time of $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}$ at point t, i. e., $\mathrm{Time}_{\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},t}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta)$,

$$\begin{aligned}&\mathrm{Time}_{\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},t}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta)\\ &\quad = \mathrm{Time}_{\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},t-1}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta) +\\ &\qquad p'(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))\\ &\quad \leqslant t \cdot p'(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))\\ &\quad \leqslant p''(\mathrm{Flow}_{Z\to\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}},A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta))\end{aligned}$$

for another polynomial p'', because

$$t < \mathrm{Flow}_{Z \to \pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z](\eta)).$$

□

The proof that π implements $\mathcal{F}_{KM}$ proceeds in several steps: Making use of the composition theorem, the last functionality $\mathcal{F}_l$ in $\mathcal{F}_{KM}$ can be substituted by its key-manageable implementation $\hat{I}_L$. Then, $\mathcal{F}_{KM}$ can simulate $\hat{I}$ instead of calling it. Let $\mathcal{F}_{KM}^{\{\mathcal{F}_l/\hat{I}_l\}}$ be the resulting functionality. In the next step, calls to this simulation are substituted by calls to the functions used in $\hat{I}$, $impl_C$ for each $C \in \mathcal{C}_l$. The resulting, partially implemented functionality $\mathcal{F}_{KM}^{\{\mathcal{F}_l/\mathtt{Impl}_{\mathcal{F}_l}\}}$ saves keys rather than credentials (for $\mathcal{F}_l$). We repeat the previous steps until $\mathcal{F}_{KM}$ does not call any KU functionalities anymore, i. e., we have $\mathcal{F}_{KM}^{\{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_n/\mathtt{Impl}_{\mathcal{F}_n}\}}$, which we abbreviate to $\mathcal{F}_{KM}^{\mathtt{impl}}$. Then we show that the network of distributed token π emulates the monolithic block $\mathcal{F}_{KM}^{\mathtt{impl}}$, which does not call KU functionalities anymore, using a reduction to the security of the key-wrapping scheme.

The first four steps will be the subject of Lemma 6, the last step is Lemma 7. But before we come to this, the following definition expresses partial implementations of $\mathcal{F}_{KM}$. In fact, the formal definition of $\mathcal{F}_{KM}$ is the special case in which the set of substituted functionalities is empty:

Definition 34 ($\mathcal{F}_{KM}$ with partial implementation)*:* Given the KU parameters $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$, and functions $(impl_{\mathtt{new}}^{\mathtt{KW}}, \mathtt{wrap}, \mathtt{unwrap})$, let $\mathtt{Impl}_{\mathcal{F}_i}$ be the algorithms defining the keymanageable implementation $\hat{I}_i$ of $\mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_p\} \subset \overline{\mathcal{F}}$. We define the partial implementation of $\mathcal{F}_{KM}$ with respect to the KU functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_p$, denoted

$\mathcal{F}_{KM}^{\{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_p/\mathtt{Impl}_{\mathcal{F}_p}\}}$, as follows:

Let the ideal protocols $\mathcal{F}_{p+1}, \ldots, \mathcal{F}_l$ be rooted at $\mathtt{prot}\text{-}\mathcal{F}_{p+1}, \ldots,$ $\mathtt{prot}\text{-}\mathcal{F}_l$. In addition, $\mathcal{F}_{KM}^{\{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_p/\mathtt{Impl}_{\mathcal{F}_p}\}}$, defines the protocol name `prot-fkm`. For `prot-fkm`, the protocol defines the following behaviour: A regular protocol machine with machine ID $\langle\langle \mathtt{reg}, \mathcal{F}_i\rangle, sid\rangle$ for $\mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$ runs the following code:

```
ready: accept <ready> from parentId
       send <ready> to <ideal,sid> (= F_KM)
relay_to: accept <m> from <ideal,sid> (= F_KM)
       send <m> to <<reg,F_i>,<sid,<prot-F_i,<>>>> (= F_i)
relay_from: accept <m> from <<reg,F_i>,<sid,<prot-F_i,<>>>
       send <m> to <ideal,sid> (= F_KM)
```

The ideal party runs the logic for $\mathcal{F}_{KM}$ described in Section-8.4, with the following alteration of the corrupt macro used in the `corrupt` and `wrap` step:

Listing 35: procedure for corrupting a credential c.

```
$\mathcal{K}_{cor} \leftarrow \mathcal{K}_{cor} \cup \{c\}$
for any Store[U,h]=⟨F,a,c⟩
        if F ∈ {KW, $\mathcal{F}_1,\ldots,\mathcal{F}_p$}
                key[c]← c
                send <corrupt•,h,c> to A
        else
                call F with <corrupt,c>
                accept <corrupt•,k> from F
                key[c]← k
                send <corrupt•,h,k> to A
```

and in the `new`, `command`, `public_command` and `unwrap` steps:

```
new[ready]: accept <new,F,a> from U ∈ 𝒰
if <F,new,a,*> ∈ Π then
        if F ∈ {$\mathcal{F}_1,\ldots,\mathcal{F}_p$}
                (k, public) ← $impl^{F}_{new}(1^\eta)$
                create h; Store[U,h] ← <F,a,k>
                $\mathcal{K} \leftarrow \mathcal{K} \cup \{k\}$
                send <new•,h,public> to U
        else if F = KW
                (k,public) ← $impl^{KW}_{new}(1^\eta)$
                if k ∈ $\mathcal{K} \cup \mathcal{K}_{cor}$
                        send <error> to A
                else
                        create h; Store[U,h] ← <F,a,k>
                        $\mathcal{K} \leftarrow \mathcal{K} \cup \{k\}$
                        send <new•,h,public> to U
        else
                call F with <new>
                accept <new•,c,public> from F
                if c ∈ $\mathcal{K} \cup \mathcal{K}_{cor}$
                        send <error> to A
                else
                        create h; Store[U,h] ← <F,a,c>
                        $\mathcal{K} \leftarrow \mathcal{K} \cup \{c\}$
                        send <new•,h,public> to U
```

```
command[finish_setup]:
        accept <C,h,m> from U ∈ 𝒰
        if Store[U,h]=<F,a,c> and <F,C,a,*>∈ Π
                if F ∈ {$\mathcal{F}_1,\ldots,\mathcal{F}_p$}
                        send <C•,$impl_C(c,m)$> to U
                else if F ≠ KW
                        call F with <C,c,m>
                        accept <C•,r> from F
                        send <C•,r> to U
```

```
public_command:
        accept <C,public,m> from U ∈ 𝒰
        if C ∈ $\mathcal{C}_{i,pub}$
```

```
                if F ∈ {F_1,...,F_p}
                        send <C•,impl_C(public,m)> to U
                else
                        call F_i with <C,public,m>
                        accept <C•,r> from F_i
                        send <C•,r> to U
```

```
unwrap[finish_setup]:
accept <unwrap,h_1,w,a_2,F_2,id> from U ∈ 𝒰
if Store[U,h_1]=<KW,a_1,c_1> and
   <KW,unwrap,a_1,a_2>∈ Π, F_2 ∈ F̄
        if c_1 ∈ K_cor
                c_2 = unwrap^<F_2,a_2,id>(c_1,w)
                if c_2 ≠ ⊥ and c_2 ∉ K
                        K_cor ← K_cor ∪ {c_2}
                        if F_2 ∈ {KW,F_1,...,F_p}
                                create h_2
                                Store[U,h_2] ← <F_2,a_2,c_2>
                                key[c_2]=c_2
                        else
                                call F_2 with <inject,c_2>
                                accept <inject•,c'>
                                if c' ∉ K ∪ K_cor
                                        create h_2
                                        Store[U,h_2] ← <F_2,a_2,c'>
                                        key[c']=c_2
                        send <unwrap•,h> to U
                else if c_2 ≠ ⊥, c_2 ∈ K and c_2 ∈ K_cor
                        create h_2
                        Store[U,h_2] ← <F_2,a_2,c_2>
                        send <unwrap•,h> to U
                else // (c_2 = ⊥ ∨ c_2 ∈ K \ K_cor)
                        send <error> to A
        else if ( c_1 ∉ K_cor and
                  ∃!c_2.<c_2,<F_2,a_2,id>,w>∈encs[c_1])
                create h_2
                Store[U,h_2] ← <F_2,a_2,c_2>
                send <unwrap•,h> to U
```

Note that the partial implementation of $\mathcal{F}_{\text{KM}}$ is not an ideal protocol in the sense of [51, § 8.2], since not every regular protocol machine runs the dummy party protocol – the party $\texttt{<reg,}\mathcal{F}_i\texttt{>}$ relays the communication with the KU functionalities.

Lemma 6: Let $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ be ku parameters such that all $F \in \overline{\mathcal{F}}$ are key-manageable. Let further $\texttt{Impl}_{\mathcal{F}_i}$ be the set of functions defining the key-manageable implementation $\hat{\Gamma}_i$ of $\mathcal{F}_i$. Then $\mathcal{F}_{\text{KM}}^{\mathcal{F}_1/\texttt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_l/\texttt{Impl}_{\mathcal{F}_l}}$ emulates $\mathcal{F}_{\text{KM}}$. Furthermore, it is poly-time.

Proof. Induction on the number of substituted KU functionalities.

BASE CASE: $\mathcal{F}_{\mathrm{KM}}^{\emptyset}$ actually equals $\mathcal{F}_{\mathrm{KM}}$. Since emulation is reflexive, $\mathcal{F}_{\mathrm{KM}}^{\emptyset}$ emulates $\mathcal{F}_{\mathrm{KM}}$. It is left to show that $\mathcal{F}_{\mathrm{KM}}$ is poly-time. The argument is actually the same as for $\pi_{\overline{\mathcal{F}},\overline{C},\Pi,\overline{\mathtt{Impl}}}$ (see proof to Lemma 16), after we have established five things: 1. instead of calling implementation functions $impl_C^F$ for $\mathcal{F} \neq \mathtt{KW}$, $\mathcal{F}_{\mathrm{KM}}$ is calling the function $\mathcal{F}$ with the same value. Since $\mathcal{F}$ is key-manageable, it is also poly-time. 2. The implementation function for wrapping is applied on a different value, but this value has the same length, therefore the same upper bound holds for its running time. 3. Graph reachability is linear in the number of credentials, which in turn is polynomial, because the flow from the environment is polynomial, and thus the number of `new` queries is polynomial, too. Therefore, only a number of credentials that is polynomial in the flow from the environment is added. 4. The relaying of messages from U_i via $\mathcal{F}_{\mathrm{setup}}$ does not add more than linearly in η to the running time, 5. similarly, for the distribution of the `<finish_setup>` message.

INDUCTION STEP: Assume $i \geqslant 1$ and $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_{i-1}/\mathtt{Impl}_{\mathcal{F}_{i-1}}}$ (in the following: $\mathcal{F}_{\mathrm{KM}}^{i-1}$) emulates $\mathcal{F}_{\mathrm{KM}}$. Since emulation is transitive, it suffices to show that $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_i/\mathtt{Impl}_{\mathcal{F}_i}}$ emulates $\mathcal{F}_{\mathrm{KM}}^{i-1}$. We will proceed in three steps: First, we will substitute $\mathcal{F}_i$ by its key-manageable implementation $\hat{I}_i$. Then, we will alter $\mathcal{F}_{\mathrm{KM}}$ to simulate $\hat{I}_i$ inside. The main part of the proof is showing that $\hat{I}_i$ can be emulated by calling $\mathtt{Impl}_{F_i}$ inside $\mathcal{F}_{\mathrm{KM}}$, storing keys instead of credentials.

The first step is a consequence of composition theorem [51, Theorem 7]. The induction hypothesis gives us that $\mathcal{F}_{\mathrm{KM}}^{i-1}$ is a polytime protocol which is rooted in `fkm`. By assumption, $\hat{I}_i$ is the key-manageable implementation of $\mathcal{F}_i$, i. e., $\hat{I}_i$ is a polytime protocol that emulates $\hat{I}_i$. Furthermore, $\hat{I}_i$ defines only `F-i`, therefore $\hat{I}_F$ is substitutable for $\mathcal{F}_i$ in $\mathcal{F}_{\mathrm{KM}}^{i-1}$. Therefore, $F^{i-1}[\mathcal{F}_i/\hat{I}_i]$ is poly-time and emulates $\mathcal{F}_{\mathrm{KM}}^{i-1}$.

In the second step, we alter $F^{i-1}[\mathcal{F}_i/\hat{I}_i]$ (in the following: $\mathcal{F}_{\mathrm{KM}}^{i-1\prime}$) such that the ideal functionality defined in $\mathcal{F}_{\mathrm{KM}}^{i-1\prime}(\mathtt{prot-fkm})$ simulates $\hat{I}_i$ locally, and calls this simulation whenever $\langle\langle \mathtt{reg}, \mathcal{F}_i\rangle, sid\rangle$ would be addressed in $\mathcal{F}_{\mathrm{KM}}$. $\hat{I}_i$ might send a message to A, in which case this message is indeed relayed to A. Since the simulation will only be called by $\mathcal{F}_{\mathrm{KM}}$, it will only respond to $\mathcal{F}_{\mathrm{KM}}$. We will call this protocol $\mathcal{F}_{\mathrm{KM}}^{i-1\prime\prime}$. To show that $\mathcal{F}_{\mathrm{KM}}^{i-1\prime\prime}$ emulates $\mathcal{F}_{\mathrm{KM}}^{i-1\prime}$, we have to make sure that in $\mathcal{F}_{\mathrm{KM}}^{i-1\prime}$, $\hat{I}_i$ can only be addressed by $\mathcal{F}_{\mathrm{KM}}$, via the relay mechanism implemented in $\langle\langle \mathtt{reg}, \mathcal{F}_i\rangle, sid\rangle$ (which consequently is not present in $\mathcal{F}_{\mathrm{KM}}^{i-1\prime\prime}$, since any call to $\langle\langle \mathtt{reg}, \mathcal{F}_i\rangle, sid\rangle$ is substituted by calls to the simulation of $\hat{I}_i$). If this is the case, then the observable output to the environment is exactly the same. First, $\hat{I}_i$ never adresses `<adv>`, so by C5, it cannot be adressed by the adversary. Second, since the environment is rooted at `proto-fkm`, it cannot address $\hat{I}_i$. Third, there is no other regular party than $\langle\langle \mathtt{reg}, \mathcal{F}_i\rangle, sid\rangle$ in `sid` that calls $\hat{I}_i$.

By C8, there cannot be other regular machines addressing $\hat{I}_i$. Therefore, $\mathcal{F}_{KM}^{i-1''}$ emulates $\mathcal{F}_{KM}^{i-1'}$. Since $\hat{I}_i$ is poly-time, $\mathcal{F}_{KM}^{i-1''}$ can simulate it and is still poly-time.

In the third step, we show that F_i emulates $\mathcal{F}_{KM}^{i-1''}$. We claim that in fact, with overwhelming probability, F_i provides a perfect simulation of $\mathcal{F}_{KM}^{i-1''}$, namely, when the list L maintained in $\mathcal{F}_{KM}^{i-1}$ describes a function from credentials to keys. Whenever a pair (c, k) is added to L (this happens only in steps `new` and `inject`), $impl_{new}^{\mathcal{F}_i}$ is used to draw c. Since for all k, $\Pr[k' = k | k' \leftarrow impl_{new}^{\mathcal{F}_i}(1^\eta)]$ is negligible by assumption (see Definition 24), for every c there is (with overwhelming probability) exactly one k such that $(c, k) \in L$, hence L describes a function from credentials to keys. So we can assume that, with overwhelming probability, this list describes a function. We will inspect the steps `new, command, wrap, unwrap` and `corrupt`, since they are the only steps that produce an output depending on the value of the credential. Note first that, since $\Pr[k' = k | k' \leftarrow impl_{new}^{\mathcal{F}_i}(1^\eta)]$ is negligible for all k, and since both $\mathcal{K}$ and $\mathcal{K}_{cor}$ are only polynomial in size, the checks for $c \in \mathcal{K} \cup \mathcal{K}_{cor}$ in step `new` pass only with negligible probability. Therefore, we can assume those checks to be non-operations. The steps `new, corrupt` and `command` are trivial to verify. Each credential is substituted by the corresponding key. The corruption macro used in both steps `corrupt` and `wrap` makes sure that for each credential $c \in \mathcal{K} \cap \mathcal{K}_{cor}$, `key[c]` contains the same key that the bijection defined by L in $\mathcal{F}_{KM}^{i-1}$ assigns to it. Furthermore, for each $c \notin \mathcal{K}, c \in \mathcal{K}_{cor}$, the step `unwrap` gives the same guarantee (by definition of step `inject` in Definition 24). Therefore, the step `wrap` correctly substitutes corrupted credentials by keys. Since $\hat{I}_i$ is key-manageable, and both credential and keys are drawn using $impl_{new}^{\mathcal{F}_i}$, with overwhelming probability, the substitution is correct for uncorrupted credentials, too. The last step to check is `unwrap`. Unless $c_1 \in \mathcal{K}_{cor}$ and $c_2 \notin \mathcal{K}$, this step restores only a previously created credential in the Store, so no substitution necessary. In case that $c_1 \in \mathcal{K}_{cor}$ and $c_2 \notin \mathcal{K}$, a credential that is freshly created and linked to the content of the wrapping (see the `inject` step) is stored, whereas in $\mathcal{F}_{KM}^{i-1''}$, it is the content of the wrapping itself that is stored.

By transitivity of emulation, we have that F_i emulates $\mathcal{F}_{KM}$. By the fact that F_i actually computes less than $\mathcal{F}_{KM}^{i-1''}$, we know it is poly-time. □

Definition 35 (guaranteeing environment)*:* Suppose Z is an environment that is rooted at r, and p is a predicate on sequences of (id_0, id_1, m). Let $S_p(Z)$ be a sandbox that runs Z but checks at the end of each activation if the predicate holds true on the list of messages sent and received by the environment (including the message about to be send). If the predicate does not hold true, S_p aborts Z_p and outputs some error symbol *fail* $\in \Sigma$. We say that Z *guarantees* a predicate p, if

there exists such a sandbox $S_p(Z)$, and for every protocol Π rooted at r, for every adversary A, we have that:

$$\Pr[\text{Exec}[\Pi, A, Z] = \mathit{fail}]$$

is negligible in η.

Let us denote a list of messages m_i from a_i to b_i, as $M^t = ((a_0, b_0, m_0), \ldots, (a_t, b_t, m_t))$. We will denote the i-prefix of this list by M^i. We can filter messages by their session ID: $M^i_{|SP}$ denotes a messages (a_i, b_i, m_i) where either $a_i = <\texttt{env}>$ and b_i is of the form

$$\langle\langle \texttt{reg}, \mathit{basePID}\rangle, \langle \alpha_1, \ldots, \alpha_{k-1}, \langle \texttt{prot} - \texttt{fkm}, \langle SP\rangle\rangle\rangle\rangle,$$

or vice versa. We say (a_j, b_j, m_j) is a response to (a_i, b_i, m_i) if (a_j, b_j, m_j) is the earliest message such that $i < j$, $a_i = b_j$, $b_j = a_i$, and that no other message at an epoch $k < i$ exists such that (a_i, b_i, m_i) is a response to (a_k, b_k, m_k). This assumes that there is a response to every query. (In case of an error, $\mathcal{F}_{KM}$ responds with $\perp$ rather than ignoring the query.) In order to tell which handles are corrupted, we need to define which handles point to the same key a given moment t.

Given $M_{|NP} = M_{|\mathcal{U}, \mathcal{U}^{ext}, \mathcal{ST}, \mathit{Room}} = ((a_0, b_0, m_0), \ldots, (a_n, b_n, m_n))$, we define $\equiv^0$ to be the empty relation and for all $1 \leqslant t \leqslant n$, we define $\equiv^t$ as the least symmetric transitive relation such that

1. $\equiv^t \subset \equiv^{t-1} \cup \{(U, h), (U, h)\}$, if $m_t = <\texttt{new}^\bullet, h, \mathit{public}>$, $a_t = U$ and $\exists s < t, F, a : m_s = <\texttt{new}, F, a>$ and (a_t, b_t, m_t) is a response to (a_s, b_s, m_s)
2. $\equiv^t \subset \equiv^{t-1} \cup \{(U_1, h_1), (U_2, h_2)\}$, if $m_t = <\texttt{share}^\bullet>$, $a_t = U_1$ and $\exists s < t :\ m_s = <\texttt{share}, (U_1, h_1), (U_2, h_2)>$ and (a_t, b_t, m_t) is a response to (a_s, b_s, m_s)
3. $\equiv^t \subset \equiv^{t-1} \cup \{(U_1, h_1), (U_2, h_2)\}$, if $m_t = <\texttt{unwrap}^\bullet, h_2>$, $a_t = U_2$ and $\exists q, r, s$: such that (a_t, b_t, m_t) is a response to (a_s, b_s, m_s), and (a_r, b_r, m_r) is a response to (a_q, b_q, m_q), and $r < s$. Furthermore:
   $m_q = <\texttt{wrap}, h_1, h_2, \texttt{id}>$, $b_q = U_1$, $m_r = <\texttt{wrap}^\bullet, w>$, $a_r = U_1$ and $m_s = \texttt{unwrap}, h_1', w, a, F, \texttt{id}>$, $b_s = U_1$ and $(U_2, h_1') \equiv^{t-1} (U_1, h_1)$.
4. $\equiv^t = \equiv^{t-1}$, otherwise.

Using this relation, we can define the predicate $\mathit{corrupted}_{M_{|NP}}(U, h)$, which holds iff either some (U^*, h^*), $((U^*, h^*) \equiv^t (U, h))$ were corrupted directly, via wrapping with a corrupted key, or injected via unwrapping, i. e., if there are $m_i, m_j \in M_{|NP}$, m_j is a response to m_i and:

- $m_j = (\mathsf{adv}, \mathsf{env}, \langle \mathtt{corrupt}^\bullet, h^*, c\rangle), m_i = (\mathsf{env}, U^*, \langle \mathtt{corrupt}, h\rangle)$ (for some c), or
- $m_j = (U^*, \mathsf{env}, \langle \mathtt{wrap}^\bullet, w\rangle)$, $m_i = (\mathsf{env}, U^*, \langle \mathtt{wrap}, h_1, h^*, \mathrm{id}\rangle)$, with $\mathit{corrupted}_{M_{|NP}}(U^*, h_1)$, or
- $m_j = (U^*, \mathsf{env}, \langle \mathtt{unwrap}^\bullet, h^*\rangle), m_i = (\mathsf{env}, U^*, \langle \mathtt{unwrap}, h_1, w, a_2, F_2, \mathit{id}\rangle)$ with $\mathit{corrupted}_{M_{|NP}}(U^*, h_1)$.

Finally, let *corrupt-before-wrap* be the following predicate on a list of messages $M^t = ((a_0, b_0, m_0), \ldots, (a_t, b_t, m_t))$: For all $i \leqslant t$ and network parameters $NP = \mathcal{U}, \mathcal{U}^{\mathrm{ext}}, \mathcal{ST}, \mathit{Room}$, we have

$$\mathit{corrupted}_{M_{|NP}}(U, h) \wedge (\mathsf{env}, U, < \mathtt{wrap}, h, h' >) \in M^i_{|NP} \Rightarrow \mathit{corrupted}_{M^i_{|NP}}(U, h).$$

Lemma 7: For any ku parameter $\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi$ and set of sets of PPT algorithms $\overline{\mathtt{Impl}}$, let $\mathcal{F}_{\mathrm{KM}}^{\mathcal{F}_1/\mathtt{Impl}_{\mathcal{F}_1},\ldots,\mathcal{F}_l/\mathtt{Impl}_{\mathcal{F}_l}}$ be the partial implementation of $\mathcal{F}_{\mathrm{KM}}$ with respect to all ku functionalities in $\overline{\mathcal{F}}$. If $KW = (\mathit{impl}^{\mathsf{KW}}_{\mathtt{new}}, \mathtt{wrap}, \mathtt{unwrap})$ is a secure and correct key-wrapping scheme (Definition 31) then $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ emulates $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ for environments that guarantee *corrupt-before-wrap*.

Proof. Proof by contradiction: Assuming that there is no adversary *Sim* such that for all well-behaved environments Z that are rooted at `prot-fkm` and guarantee *corrupt-before-wrap*, both networks are indistinguishable, i. e.,

$$\mathrm{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z] \approx \mathrm{Exec}[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}, \mathit{Sim}, Z]$$

we chose a *Sim* that basically simulates $\mathcal{F}_{\mathrm{setup}}$ for corrupted users in $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$, and a Z that is indeed able to distinguish $\mathrm{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$ and $\mathrm{Exec}[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}, \mathit{Sim}, Z]$. Then, we use it to construct an attacker $\mathcal{B}_Z$ against the key-wrapping challenger. B_Z will be carefully crafted, such that *a*) it is a valid adversary *b*) it has the same output distribution in the fake key-wrapping experiment as Z has when interacting with $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ and *Sim c*) it has the same output distribution in the real key-wrapping experiment as Z in interaction with $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ and A_D.

Sim defines the same code as the dummy adversary (see [51, §4.7]), but when instructed by the environment to instruct a corrupted party to call $\mathcal{F}_{\mathrm{setup}}$, it simulates $\mathcal{F}_{\mathrm{setup}}$ (because $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ does not define `prot-fsetup`). This means: *Sim* waits for $\langle \mathtt{ready}^\bullet, P\rangle$ from $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl}}$ for all parties $P \in \mathcal{U} \cup \mathcal{ST} \cup \hat{\mathcal{F}}$ before operating - for corrupted parties $U_i \in \mathcal{U}$ (security tokens are incorruptible, $U^{\mathrm{ext}} \in \mathcal{U}^{\mathrm{ext}}$ are ignored), it waits until instructed to send `ready` and simulates the reception of $\langle \mathtt{ready}^\bullet, P\rangle$ itself. Afterwards, it accepts instructions to send $\langle m\rangle$ as U_i to $\mathcal{F}_{\mathrm{setup}}$–

in this case, *Sim* instructs U_i to send m to $\mathcal{F}_{KM}^{impl}$. Similarly, the response from $\mathcal{F}_{KM}^{impl}$ is simulated to be transmitted via $\mathcal{F}_{setup}$. When U_i is instructed to send `finish_setup` to $\mathcal{F}_{setup}$, *Sim* sends `finish_setup` to $\mathcal{F}_{KM}^{impl}$ instead (and relays the answer), but only if it received ready from all parties $P \in \mathcal{U} \cup \mathcal{ST} \cup \hat{\mathcal{F}}$ before (as we already mentioned), and only the first time.

Given Z, we will now construct the attacker $\mathcal{B}_Z$ against the key-wrapping game in Definition 31. Recall that Z is rooted at `prot-fkm`. This means that Z only calls machines with the same SID and the protocol name `prot-fkm`. In particular, the session parameters $(\overline{\mathcal{F}}, \overline{\mathcal{C}}, \Pi)$ are the same (see [51, §5.3]), so from now on, we will assume them to be arbitrary, but fixed. The construction of $\mathcal{B}_Z$ aims at simulating Z in communication with the simulator *Sim* given above and the key-management functionality $\mathcal{F}_{KM}^{impl}$, but instead of performing wrapping and unwrapping in $\mathcal{F}_{KM}^{impl}$ itself, $\mathcal{B}_Z$ queries the challenger in the wrapping experiment. In case of the fake experiment, the simulation is very close to the network $[\mathcal{F}_{KM}^{impl}, Sim, Z]$, for the case of the real experiment, we have to show that the output is indistinguishable from a network of distributed security tokens and a dummy adversary $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$. This will be the largest part of the proof. $\mathcal{B}_Z$ is defined as follows: $\mathcal{B}_Z$ simulates the network $[\mathcal{F}_{KM}^{impl,KW}, Sim, Z]$, where $\mathcal{F}_{KM}^{impl,KW}$ is defined just as $\mathcal{F}_{KM}^{impl}$, but `new,wrap,unwrap` and `corrupt` are altered such that they send queries to the experiment instead. Note that for this reason, $\mathcal{F}_{KM}^{impl,KW}$ is not a valid machine in the GNUC model - we just use it as a convenient way to describe the simulation that $\mathcal{B}_Z$ runs. We assume that the place-holder symbols $K_1, \ldots$ from Definition 31 are distinguishable from other credentials and that there is some way to select those symbols such that each of them is distinct. Furthermore, they should be difficult to guess. One way to achieve this is to implement a pairing of i and j, for $U = U_i$ and $j \leftarrow \{0,1\}^\eta$, using Cantor's pairing function.

```
new[ready]: accept <new,F,a> from U ∈ 𝒰
if <F,new,a,*> ∈ Π then
        if F ∈ {F_1,...,F_l}
                (k, public) ← impl^F_new(1^η)
                create h; Store[U,h] ← <F,a,k>
                K ← K ∪ {k}
                send <new•,h,public> to U
        else if F = KW
                create K_i, h
                query NEW(K_i)
                K ← K ∪ {K_i}
                Store[U,h] ← <KW,a,K_i>
                send <new•,h,> to U
```

```
wrap[finish_setup]:
accept <wrap,h_1,h_2,id> from U ∈ 𝒰
if Store[U,h_1]=<KW,a_1,c_1> and Store[U,h_2]=<F_2,a_2,c_2>
```

```
        and <KW,wrap,a1,a2>∈ Π
        if ∃w.<c2,<F2,a2,id>,w>∈encs[c1]
                send <wrap•,w> to U
        else
                W ← W ∪{(c1,c2)}
                if c1 ∈ Kcor
                 for all c3 reachable from c2 in W
                        corrupt c3
                 w ← wrap^<F2,a2,id>(key[c1],key[c2])
                else
                        w = TENC(c1,< F2,a2,id >,c2)
                encs[c1] ← encs[c1] ∪{ <c2,<F2,a2,id>,w>}
                send <wrap•,w> to U
```

```
unwrap[finish_setup]:
accept <unwrap,h1,w,a2,F2,id> from U ∈ U
if Store[U,h1]=<KW,a1,c1> and <KW,unwrap,a1,a2>∈ Π,F2 ∈ F̄
        if c1 ∈ Kcor
                c2 = unwrap^<F2,a2,id>(key[c1],w)
                if c2 ≠ ⊥ and c2 ∉ K \ Kcor
                        Kcor ← Kcor ∪{c2}
                        create h2
                        Store[U,h2] ← <F2,a2,c2>
                        key[c2]=c2
                        send <unwrap•,h> to U
                else // first bad event
                        send <error> to A
        else
                if ∃!c2.<c2,<F2,a2,id>,w>∈encs[c1]
                        create h2
                        Store[U,h2] ← <F2,a2,c2>
                        send <unwrap•,h> to U
                else //c1 ∉ Kcor ∧ ¬∃c2.⟨c2,⟨F2,a2,id⟩,w⟩ ∈encs[c1])
                        query c2 = DEC(c1,< F2,a2,id >,w);
                        if c2 ≠ ⊥ //second bad event
                                halt and output 0.
```

Listing 36: Procedure for corrupting a credential c.

```
if c ∈ Kcor
        send <corrupt•,h,key[c]> to A
else
        Kcor ← Kcor ∪{c}
        for any Store[U,h]=<F,a,c>
                if F ∈ KW
                        query k = CORR(c)
                        key[c]← k
                else
                        key[c]← c
                send <corrupt•,h,key[c]> to A
```

$\mathcal{B}_Z$ IS A VALID ADVERSARY We will argue about each condition on the behaviour of the adversary from Definition 31:

1. For all i, the query $\mathsf{NEW}(\mathsf{K}_i)$ is issued at most once, because we specified $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ to select a new K_i for each such query
2. All queries issued by $\mathcal{B}_Z$ contain keys that have already been generated by the experiment. Observe that all queries are preceeded by a conditional that checks if the argument to the query is in the third position of the store, i. e., there are U, h, a such that $\mathtt{Store}[U, h] = \langle \mathsf{KW}, a, k\rangle$ at some point of the execution of $\mathcal{B}_Z$. We claim that each such k has either been generated using NEW or is in $\mathcal{K}_{\mathrm{cor}}$ (in which case no query is made). Proof by contradiction: Assume we are at the first point of the execution where such a key is added to the store. The store is only written in the `new` and `unwrap` step. In `new`, a new K_i is created. In `unwrap`, there are three cases in which the store is written to: *a*) If $c_1 \in \mathcal{K}_{\mathrm{cor}}$, then $c_2 \in \mathcal{K}_{\mathrm{cor}}$. Once something is marked as corrupted, it stays corrupted. *b*) If $c_1 \notin \mathcal{K}_{\mathrm{cor}}$, but $\exists c_2.\ \langle c_2, \langle \mathsf{F}_2, a_2, \mathit{id}\rangle, w\rangle \in \mathtt{encs}[c_1]$. Only `wrap` can write to `encs`, so c_2 must have been in the store before.
3. The adversary never makes a test query $\mathsf{TENC}(\mathsf{K}_i, a, \mathsf{K}_j)$ if K_i is corrupted at the end of the experiment, because a TENC query is only output in the step `wrap` if $c_1 \notin \mathcal{K}_{\mathrm{cor}}$. The condition *corrupt-before-wrap* enforces that if c_1 is not corrupted at that point, it will never be corrupted. (A detailed analysis about how *corrupt-before-wrap* is correct with respect to the definition if $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ is left to the reader.)
4. If $\mathcal{B}_Z$ issues a test query $\mathsf{TENC}(\mathsf{K}_i, a, m)$ then $\mathcal{B}_Z$ neither issues $\mathsf{TENC}(\mathsf{K}_j, a', m')$ nor $\mathsf{ENC}(\mathsf{K}_j, a', m')$ with $(\mathsf{K}_i, a, m) = (\mathsf{K}'_j, a, m')$, since $\mathcal{B}_Z$ never issues ENC queries at all and only issues TENC queries if the same combination of (K_i, a, m) was not stored in `encs` before. Every time TENC is called, `encs` is updated with those parameters.
5. $\mathcal{B}_Z$ never queries $\mathsf{DEC}(\mathsf{K}_i, a, c)$ if c was the result of a query $\mathsf{TENC}(\mathsf{K}_i, a, m)$ or of a query $\mathsf{ENC}(\mathsf{K}_i, a, m)$ or K_i is corrupted, because *a*) TENC queries are stored in `encs` and the step `unwrap` checks this variable before querying DEC, *b*) *enc* queries are not issued at all, *c*) if a credential c_1 inside the `unwrap` step is corrupted, the query DEC is not issued.

We conclude that $\mathcal{B}_Z$ fulfills the assumptions on the behaviour of the adversary expressed in Definition 31.

$\mathcal{B}_Z$ SIMULATES $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ IN THE FAKE EXPERIMENT $\mathcal{B}_Z$ is defined to be a simulation of the network $\left[\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}, \mathit{Sim}, \mathsf{Z}\right]$, where

$\mathcal{F}_{\text{KM}}^{\text{impl,KW}}$ is $\mathcal{F}_{\text{KM}}^{\text{impl}}$, except for the altered steps `new`,`wrap`,`unwrap` and `corrupt`. We claim that, in the fake experiment, those alterations do not change the input/output behaviour. First, we will simplify the `unwrap` step in $\mathcal{F}_{\text{KM}}^{\text{impl}}$:

```
1  unwrap[finish_setup]:
2  accept <unwrap,h1,w,a2,F2,id> from U ∈ 𝒰
3  if Store[U,h1]=<KW,a1,c1> and
4     <KW,unwrap,a1,a2>∈ Π,F2 ∈ F̄
5          if c1 ∈ 𝒦cor
6                  c2 = unwrap^<F2,a2,id>(c1,w)
7                  if c2 ≠ ⊥ and c2 ∉ 𝒦
8                          𝒦cor ← 𝒦cor ∪ {c2}
9                          create h2
10                         Store[U,h2] ← <F2,a2,c2>
11                         key[c2]=c2
12                         send <unwrap•,h> to U
13                 else if c2 ≠ ⊥, c2 ∈ 𝒦 and c2 ∈ 𝒦cor
14                         create h2
15                         Store[U,h2] ← <F2,a2,c2>
16                         send <unwrap•,h> to U
17                 else // (c2 = ⊥ ∨ c2 ∈ 𝒦 \ 𝒦cor)
18                         send <error> to A
19         else if ( c1 ∉ 𝒦cor and
20                  ∃!c2.<c2,<F2,a2,id>,w>∈encs[c1])
21                 create h2
22                 Store[U,h2] ← <F2,a2,c2>
23                 send <unwrap•,h> to U
```

We first observe that it would not make a difference if the code in the branch at Line 13 would execute the same code as in the branch at Line 7, i. e., additionally perform the computations in Line 8 and 11, since from the definition of the steps `unwrap` and the corruption procedure, if $c_2 \in \mathcal{K}_{\text{cor}}$, then already $\mathsf{key}[c_2] = c_2$. This means that Lines 7 to 12 are executed if $c_2 \neq \bot \wedge c_2 \notin \mathcal{K} \setminus \mathcal{K}_{\text{cor}}$, otherwise a bad event is produced, i. e., an error is send to the adversary. For reference, this is the equivalent, simpler code:

```
   unwrap[finish_setup]:
2  accept <unwrap,h1,w,a2,F2,id> from U ∈ 𝒰
   if Store[U,h1]=<KW,a1,c1> and
      <KW,unwrap,a1,a2>∈ Π,F2 ∈ F̄
           if c1 ∈ 𝒦cor
                   c2 = unwrap^<F2,a2,id>(c1,w)
7                  if c2 ≠ ⊥ and c2 ∉ 𝒦 \ 𝒦cor
                           𝒦cor ← 𝒦cor ∪ {c2}
                           create h2
                           Store[U,h2] ← <F2,a2,c2>
                           key[c2]=c2
12                         send <unwrap•,h> to U
                   else // (c2 = ⊥ ∨ c2 ∈ 𝒦 \ 𝒦cor)
                           send <error> to A
```

```
        else if ( c1 ∉ 𝒦cor and
                  ∃!c2.<c2,<F2,a2,id>,w>∈encs[c1])
17              create h2
                Store[U,h2] ← <F2,a2,c2>
                send <unwrap•,h> to U
```

We claim that the following invariant holds: For all Z, and at the end of every epoch [51, §5.3], i. e., after each activation of Z:

- the distribution of the input received by Z (the view of Z) is the same for Z in $[\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl}}, \mathit{Sim}, \mathsf{Z}]$ as well as in the network $[\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}, \mathit{Sim}, \mathsf{Z}]$ simulated by $\mathcal{B}_{\mathsf{Z}}$.
- in the same two execution, the entry $\mathtt{Store}[\mathsf{U}, \mathsf{h}] = \langle \mathtt{KW}, \mathsf{a}, \mathsf{c} \rangle$ exists in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}$ if, and only if, the entry $\mathtt{Store}[\mathsf{U}, \mathsf{h}] = \langle \mathtt{KW}, \mathsf{a}, \overline{\mathsf{c}} \rangle$ exists in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl}}$, and the following holds for c and $\overline{\mathsf{c}}$
  - $\mathsf{c} \in \mathcal{K} \leftrightarrow \overline{\mathsf{c}} \in \mathcal{K}$ and $\mathsf{c} \in \mathcal{K}_{\mathrm{cor}} \leftrightarrow \overline{\mathsf{c}} \in \mathcal{K}_{\mathrm{cor}}$
  - if $\mathsf{c} \in \mathcal{K}$, then $\overline{\mathsf{c}}$ is the key drawn for c in the NEW step. Furthermore, if $\mathsf{c} \in \mathcal{K}_{\mathrm{cor}}$, in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}$ $\mathtt{key}[\mathsf{c}] = \overline{\mathsf{c}}$ and in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl}}$ $\mathtt{key}[\overline{\mathsf{c}}] = \overline{\mathsf{c}}$
  - if $\mathsf{c} \in \mathcal{K}_{\mathrm{cor}} \setminus \mathcal{K}$ (i. e., c was injected), $\mathsf{c} = \overline{\mathsf{c}}$ and $\mathtt{key}[\mathsf{c}] = \mathsf{c}$ in both $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}$ and $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl}}$.

The only relevant steps are the altered steps `new`,`wrap`,`unwrap`, and the corruption macro.

- `corrupt`: For honestly generated wrapping keys $\mathsf{c} \in \mathcal{K}$, by induction hypothesis, the corruption procedure outputs the key generated with NEW in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}$, and the equally drawn key in case of $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl}}$. Assume $\mathsf{c} \notin \mathcal{K}$. If $\mathsf{c} \notin \mathcal{K}_{\mathrm{cor}}$, by induction hypothesis, both $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl}}$ and $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}$ produce an error and leave their respective store unchanged. For dishonestly generated keys – that is, for the missing case where $\mathsf{c} \in \mathcal{K}_{\mathrm{cor}} \setminus \mathcal{K}$ – both output the same keys by induction hypothesis. The second condition holds by definition of this step and induction hypothesis.
- `new`: From the definition of the fake experiment follows that the first condition holds. The second condition holds since the freshly created c is added to $\mathcal{K}$ and since c is the argument to the NEW step.
- `wrap`: By definition of the fake experiment, ENC and TENC substitute credentials c in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}$. If $\mathsf{c}_1 \notin \mathcal{K}_{\mathrm{cor}}$, then $\mathsf{c}_1 \in \mathcal{K}$ (as otherwise it would not be in the database). Thus, in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}$, TENC will perform this substitution for both c_1 and c_2, resulting in the same output that $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl}}$ produces. If $\mathsf{c}_1 \in \mathcal{K}_{\mathrm{cor}}$, then, by definition of this step in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl}}$, $\mathsf{c}_2 \in \mathcal{K}_{\mathrm{cor}}$ at the point in the time where the unwrap function is computed. Therefore, and by induction hypothesis, $\mathtt{key}[\mathsf{c}_1]$ in $\mathcal{F}_{\mathrm{KM}}^{\mathsf{impl},\mathsf{KW}}$ contains $\overline{\mathsf{c}}_1$. The

same holds for c_2). Hence, the same value is computed and output in $\mathcal{F}_{KM}^{impl,KW}$ and $\mathcal{F}_{KM}^{impl}$ and the first condition holds. The second condition holds by induction hypothesis and the fact that both functions call the `corrupt` macro on c_2 if $c_1 \in \mathcal{K}_{cor}$.

- `unwrap`: Since for $c_1 \notin \mathcal{K}_{cor}$ (by definition of the fake experiment), DEC produces $\perp$, i. e., the conditional marked "second bad event" never evaluates. Note furthermore, that this step is the same in $\mathcal{F}_{KM}^{impl,KW}$ and $\mathcal{F}_{KM}^{impl}$, only that the call to the `unwrap` function in case $c_1 \in \mathcal{K}_{cor}$ substitutes c_1 by $\mathsf{key}(c_1)$ in $\mathcal{F}_{KM}^{impl,KW}$. This is correct by the second condition of the induction hypothesis. Therefore, both conditions of the induction hypothesis are preserved for the next step.

We can conclude that

$$\mathbf{Exp}_{KW,\mathcal{B}_Z}^{wrap,fake}(\eta) = \mathrm{Exec}[\mathcal{F}_{KM}^{impl}, Sim, Z](\eta).$$

$\mathcal{B}_Z$ SIMULATES $\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\mathtt{Impl}}}$ IN THE REAL EXPERIMENT In the fake experiment, it is not possible that $\mathcal{B}_Z$ halts at the end of the `unwrap` step (marked "second bad event"), since DEC always outputs $\perp$. Thus, the probability that $\mathcal{B}_Z$ halts at the "second bad event" mark whilst in the real experiment must be negligible, as this would contradict the assumption that KW is a secure wrapping scheme right here. The representation of the this part of the proof benefits from altering $\mathcal{B}_Z$ such that instead of halting, $\mathcal{B}_Z$ continues to run $\mathcal{F}_{KM}^{impl,KW}$, by running the following code:

```
1 create h2; Store[U,h2]← <F2,a2,c2>
  send <unwrap•,h> to U
```

We further modify the code of $\mathcal{B}_Z$ by removing the conditional in front of the comment "first bad event". Here as well, the probability that this conditional evaluates to true is negligible, following from the assumption that KW is a secure wrapping scheme. Proof by contradiction: If $\mathcal{B}_Z$ could produce this conditional, then she knows a wrapping w such that $unwrap^a(k_1, w) = c_2$ for $c_2 \in \mathcal{K}$ and $c_2 \notin \mathcal{K}_{cor}$. Since $c_1 \in \mathcal{K}_{cor}$ and $k_1 = \mathsf{key}[c_1]$, $\mathcal{B}_Z$ either knows k_1 (since it injected it, i. e., $c_1 \notin \mathcal{K}$), or can learn by corrupting it (if $c_1 \in \mathcal{K}$). If there was a path in $\mathcal{W}$ from c_1 to c_2, it would already be corrupted, so the attacker can learn k_1 while $c_2 \notin \mathcal{K}_{cor}$. She can use k_1 as to decrypt w himself and learn c_2. But she should not be able to learn c_2 in the fake experiment, since it is randomly drawn and did never appear in any output[1]. Since this happens only with neglible probability, it can only happen with negligible probability in the real experiment, too,

1 The adversary can check if it guessed c_2 correctly, for example, by requesting a wrapping of a third, corrupted wrapping key under c_2 via $\mathcal{F}_{KM}^{impl,KW}$ and then calling DEC to check if the received wrapping (using the same argument) decrypts to the corrupted and thus known key.

as otherwise the assumption that *KW* is a secure wrapping scheme would be contradicted.

Therefore, we perform those modifications and call the sightly different attacker $\mathcal{B}'_Z$. Since those parts of the code are only executed with negligible probability, we have that

$$\Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},\mathcal{B}_Z}(\eta) : b = 1] - \Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},\mathcal{B}'_Z}(\eta) : b = 1]$$

is negligible in η.

Fix an arbitrary security parameter η. Then, let $\mathit{View}_{\mathcal{B}'_Z}(t)$ be the view of Z, i. e., the distribution of messages it is simulated to send to the protocol machine or the adversary, in the tth step of the simulation of $\mathcal{B}'_Z$ in the real experiment. Furthermore, let $\mathit{Store}_{\mathcal{B}'_Z}(t)$ be the distribution of the variable `Store` within the simulated machine $\langle \mathtt{ideal}, \mathit{sid} \rangle$, i. e., $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$, but with the following substitution that affects the wrapping keys: Every entry $\langle \mathtt{KW}, a, K_i \rangle$ in the variable `Store[U,h]` for some U and h is substituted by an entry $\langle \mathtt{KW}, a, k_i \rangle$, where k_i is the key that the key-wrapping experiment associates to K_i, denoted in the following by $k(K_i)$. If K_i was not created by the key-wrapping experiment, it is left untouched.

We will denote the view of Z in the execution of the network $[\pi_{\overline{\mathcal{F}},\overline{C},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$ by View_π and the distribution of the union of all `Store` variables of all security tokens $ST_1, \ldots, ST_n$ in the network as Store_π. The union of those variables is still a map, because the first element of each key-value of this table is different for all ST. A step t is an epoch [51, §5.3], i. e., it begins with an activation of Z and ends with the next activation.

We define the following invariant, which will allow us to conclude that $\mathcal{B}'_Z$ has the same output distribution as $[\pi_{\overline{\mathcal{F}},\overline{C},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$. For each number of steps t, the following three conditions hold:

- *state consistency (s.c.)*: $\mathit{Store}_{\mathcal{B}'_Z}$ (t) and Store_π are equally distributed
- *output consistency (o.c.)*: $\mathit{View}_{\mathcal{B}'_Z}$ (t) and View_π are equally distributed
- *phase consistency (p.c.)*: The probability that the flag `ready` is set in $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ in $\mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},\mathcal{B}'_Z}$ equals the probability that `ready` is set in $\mathcal{F}_{\mathrm{setup}}$ in $[\pi_{\overline{\mathcal{F}},\overline{C},\Pi,\overline{\mathtt{Impl}}}, A_D, Z]$. Furthermore, the probability that the flag `setup_finished` is set in $\mathcal{F}_{\mathrm{KM}}^{\mathtt{impl},\mathsf{KW}}$ in $\mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},\mathcal{B}'_Z}$ equals the probability that `setup_finished` is set in all $ST \in \overline{\mathcal{ST}}$.
  .

If $t = 0$, the protocol has not been activated, thus there was no output, and not state changes. The invariant holds trivially. If $t > 0$, we can assume that s.c., o.c. and p.c. were true at the end of the preceding epoch. Note that Z is restricted to addressing top-level parties

with the same *sid*. In particular, it cannot address $\mathcal{F}_{\text{setup}}$ directly (but it can corrupt a user to do this). Since the *sid* has to have a specific format that is expected by all machines in both networks, we assume *sid* to encode $\mathcal{U}, \mathcal{U}^{\text{ext}}, \mathcal{ST}, \mathit{Room}$. Case distinction over the recipient and content of the message that Z sends at the beginning of the next epoch:

1. Z sends a message to $ST_i \in \mathcal{ST}$, and
   a) the message is $\langle \texttt{ready} \rangle$: In $\mathbf{Exp}^{\text{wrap},}_{\text{KW},\mathcal{B}'_Z}$, $\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$ records $\texttt{ready-}ST_i$ and sends $\langle \texttt{ready}^\bullet, ST_i \rangle$ to *Sim*, if the message is recorded for the first time. *Sim* behaves just like A_D in this case. If all other $ST_j \in \mathcal{ST}$ and all $U \in \mathcal{U}$ have sent this message before, the flag `ready` is set to true.

      In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, ST_i accepts the message and forwards it (as $\langle \texttt{ready} \rangle$) to $\mathcal{F}_{\text{setup}}$. Then, $\mathcal{F}_{\text{setup}}$ records $\texttt{ready-}ST_i$ and sends $< \texttt{ready}^\bullet, ST_i >$ to A_D, if the message is recorded for the first time. If all other $ST_j \in \mathcal{ST}$ and all $U \in \mathcal{U}$ have sent this message before, the flag `ready` is set to true. We see that p.c. and o.c. hold. S.c. holds trivially, because the Store did change in neither execution.

   b) the message is of some other form: In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, ST accepts no other message from the environment. In $\mathbf{Exp}^{\text{wrap},}_{\text{KW},\mathcal{B}'_Z}$, $\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$ ignores any other message coming from ST_i, too. So p.c., o.c. and s.c. hold.

2. Z sends a message to $U_i \in \mathcal{U}$:

   In $\mathbf{Exp}^{\text{wrap},}_{\text{KW},\mathcal{B}'_Z}$, $\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$ will receive this message, and treat it depending on its form (if its flag `ready` is set). In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, U_i will relay this message m in the form $<m, ST_i>$ to $\mathcal{F}_{\text{setup}}$, who in turn will send m to ST_i, (assuming the steps necessary for the flag `ready` were completed).

   a) Let m be `<ready>`: If the `ready` flag has not been set before, and U_i is the last party in $\mathcal{U} \cup \mathcal{ST}$ that has not sent this message yet, $\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$ in $\mathbf{Exp}^{\text{wrap},}_{\text{KW},\mathcal{B}'_Z}$ will set the `ready` flag, otherwise it will not. The same holds for $\mathcal{F}_{\text{setup}}$ in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$. Therefore, we have p.c. In both cases, *Sim*, respectively A_D, forwards the acknowledgement to the environment (after recording the state change). Thus, s.c. and o.c. hold trivially.

      For the following cases, assume `ready` to be set in both $\mathbf{Exp}^{\text{wrap},}_{\text{KW},\mathcal{B}'_Z}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$. If it is unset in one of them, by induction hypothesis, it is unset in both. If it is not set, any other message will not be accepted by neither $\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$, nor $\mathcal{F}_{\text{setup}}$ (thus never reach ST_i). Therefore, in the following cases we will assume `ready` to be set in

$\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$ and $\mathcal{F}_{\text{setup}}$, i. e., $\mathcal{F}_{\text{setup}}$ delivering commands that U_i receives from the environment to ST_i.

b) Let m=<new,F,a> : If $F \neq \texttt{KW}$, the same code is executed (except for the step adding the freshly created credential to $\mathcal{K}$), p.c.,s.c. and o.c. hold trivially. Assume $F = \texttt{KW}$ and <KW,new,a,*>$\in \Pi$ (otherwise, $\perp$ is output in both executions). $\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$ draws a new K_i and call NEW to create the key. K_i is created, just like handles, in a way that makes sure it is unique. Therefore, since throughout $\mathbf{Exp}_{\text{KW},\mathcal{B}'_Z}^{\texttt{wrap},}$, K_i is always substituted for the same key, there is a function mapping K_i to the key k_i created by the experiment, and this function is injective. Note that, by the definition of $Store_{\mathcal{B}'_Z}$, $Store_{\mathcal{B}'_Z}(t)$ is $Store_{\mathcal{B}'_Z}(t-1)$ with an additional entry $\langle \texttt{KW}, a, k_i \rangle$ at $[U, h]$, where k_i is distributed according to KG. In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, ST_i calls the key-generation directly ($impl_{\texttt{new}}^{KW}$ calls KG, adding nothing but an empty public part). The output in both cases is <new$^\bullet$,h,> for an equally distributed h. Thus, o.c. holds. $Store_\pi$ is $Store_\pi(t-1)$ with an additional entry KW,a,k_i at $[U, h]$ where k_i is distributed according to the same KG as above. Therefore, s.c. holds. (P.c. holds trivially.)

c) Let $m = \langle \texttt{share}, h_i, U_j \rangle$: In $\mathbf{Exp}_{\text{KW},\mathcal{B}'_Z}^{\texttt{wrap},}$, assuming $U_i, U_j \in$ *Room*, $\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$ outputs $\langle \texttt{share}^\bullet, h_j \rangle$, and $Store_{\mathcal{B}'_Z}(t)$ equals $Store_{\mathcal{B}'_Z}(t-1)$ extended by a copy of its entry $[U_i, h_i]$ at $[U_j, h_j]$. In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, ST_i checks the same conditions (which by p.c. have an equal probability of success) implicitly: It sends the content of its store at $[U_i, h_i]$ to $\mathcal{F}_{\text{setup}}$, which verifies $U_i, U_j \in$ *Room*. If this is not the case, $\mathcal{F}_{\text{setup}}$ sends $\perp$ to ST_i, which sends this to the environment (via $\mathcal{F}_{\text{setup}}$), behaving just like $\mathbf{Exp}_{\text{KW},\mathcal{B}'_Z}^{\texttt{wrap},}$. If the condition is met, ST_i sends the content of the store at $[U_i, h_i]$ to $\mathcal{F}_{\text{setup}}$, which delivers this information to ST_j, which in the next step extends $Store_\pi(t-1)$ by a copy of its entry $[U_i, h_i]$ at $[U_j, h_j]$. Thus, s.c. holds. Both output <share$^\bullet$,h_j> upon success, so o.c. holds, too. p.c. holds trivially.

d) Let $m = \langle \texttt{finish\_setup} \rangle$: By p.c., we have that $\mathbf{Exp}_{\text{KW},\mathcal{B}'_Z}^{\texttt{wrap},}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$ either both have finish_setup set, or none has. If both have it set, both output $\perp$ and do nothing.

Assume none has finish_setup set and both ready. In **Exp**, $\mathcal{F}_{\text{KM}}^{\texttt{impl},\text{KW}}$ sets the flag finish_setup and responds. In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, U_i sends <finish_setup> to $\mathcal{F}_{\text{setup}}$, which in turn, instead of forwarding it to ST_i like for the majority of commands, sends <close> to every $ST_j \in \mathbb{ST}$, accepting the response (and thus taking control) after each

of those have set the `finish_setup` flag. By the time $\mathcal{F}_{\text{setup}}$ finishes this step, and hands communication over to U_i, which forwards `finish_setup` to the environment, every ST_i has left the setup phase. We see that p.c. and o.c. are preserved.

e) Let $m = \langle C, h, m \rangle$: $\mathcal{F}_{\text{KM}}^{\text{impl,KW}}$ and ST_i execute the same code on their inputs, so by s.c., the invariant is preserved.

f) Let $m = \langle \texttt{corrupt}, h \rangle$: ST_i outputs the credential. $\mathcal{F}_{\text{KM}}^{\text{impl,KW}}$ does the same, except for wrapping keys, where it substitutes the credential by the output of CORR, i. e., $k = k(c)$. By definition of $Store_\pi$ as s.c., $< \texttt{KW}, a, k > \in Store_\pi[U_i, h]$ with the same probability as $< \texttt{KW}, a, c > \in Store_{\mathcal{B}'_Z}[U_i, h]$, thus the output is equally distributed. S.c. and p.c. hold trivially.

g) Let $m = \langle \texttt{wrap}, h_1, h_2, id \rangle$: For this case and the following case, observe that $\mathcal{F}_{\text{KM}}^{\text{impl,KW}}$ initialises `key`$[c]$ only at steps `wrap`, `unwrap` and `corrupt`.

The variable `key`$[c]$ contains either the output of a query CORR, thus $k(c)$, or the same value as c, or it is defined. It is defined whenever $c \in \mathcal{K}_{\text{cor}} \cap \mathcal{K}$, because if c is added to $\mathcal{K}_{\text{cor}}$ at step `corrupt`, the response is written to `key`$[c]$, and if a $c_3 \notin \mathcal{K}_{\text{cor}}$ is found during step `wrap`, the condition *corrupt-before-wrap* must have been violated by Z: If such a c_3 is reachable from c_1, without loss of generality, assume it to have minimal distance from c_1 in $\mathcal{W}$. Then, the second-before last node on this path is in $\mathcal{K}_{\text{cor}}$, as the distance would not be minimal otherwise. By definition of the step `wrap`, this node could not have been wrapped without adding it to $\mathcal{K}_{\text{cor}}$, therefore this node was corrupted after it was used to create this wrapping. If $c \in \mathcal{K}_{\text{cor}}$, but $c \notin \mathcal{K}$, then `key`$[c] = c$ (see step `unwrap`).

Assume now ST_i and $\mathcal{F}_{\text{KM}}^{\text{impl,KW}}$ both have `finish_setup` set, as otherwise either p.c. was violated in the previous step, or both would output $\perp$ and trivially satisfy the invariant. (This argument is valid for each of the following sub-cases, but the last one).

Both machines check the same conditions on the Store and the policy. ST_i computes $w = \texttt{wrap}^{\langle F_2, a_2, id \rangle}(c_1, c_2)$ on the values $\langle \texttt{KW}, a_1, c_1 \rangle$ and $\langle F_2, a_2, c_2 \rangle$ at $[U_i, h_1]$ and $[U_i, h_2]$ in $Store_\pi$.

$\mathcal{F}_{\text{KM}}^{\text{impl,KW}}$ performs a case distinction, but we will show that in each cases, it outputs the same value. If $\langle c_2, \langle F_2, a_2, id \rangle, w \rangle \in \texttt{encs}[c_1]$, then by observing that `encs` is only written at the end of this function, we see that p.c. would have

been violated in an earlier step, if the output now was differently distributed then the output in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$.

If $c_1 \in \mathcal{K}_{cor}$, then $c_2 \in \mathcal{K}_{cor}$, too. Assume $c_1, c_2 \in \mathcal{K}$. Then, since $\texttt{key}(c_1) = k(c_1)$ (and $\texttt{key}(c_2) = k(c_2)$ in case $F_2 = \texttt{KW}$), the output is $w = \texttt{wrap}^{<F_2,a_2,id>}(k(c_1), c_2))$ (or $w = \texttt{wrap}^{<F_2,a_2,id>}(k(c_1), k(c_2))$), which preserves o.c., since s.c. from the last step guarantees that $\langle \texttt{KW}, a, c\rangle \in \mathit{Store}_\pi[U_i, h]$ which equals $\langle \texttt{KW}, a, k(c)\rangle \in \mathit{Store}_{\mathcal{B}'_Z}[U_i, h]$ for $c = c_1$ and $c = c_2$, in case c_2 is a wrapping key. By definition of the real experiment, it performs the same substitutions in case $c_1 \notin \mathcal{K}_{cor}$, so the same argument can be applied. In case that $c_1 \notin \mathcal{K}$, or $c_2 \notin \mathcal{K}$, the substitution performed is the identity, as $\texttt{key}[c] = c$ for $c \in \mathcal{K}_{cor} \setminus \mathcal{K}$. Therefore, in this case, the same output is produced in both $\mathcal{F}_{KM}^{\texttt{impl},KW}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$.

Therefore, the output is equally distributed in all three cases, assuming that s.c. was true for the previous step. s.c. and p.c. hold trivially.

h) Let m=<unwrap,h_1,w,a_2,F_2,*id*> : In $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, if policy and Store allow, i. e., $\langle F_1, a_1, c_1\rangle \in \mathit{Store}_\pi(U_i, h_1)$, then ST_i writes $\langle F_2, a_2, \texttt{unwrap}^{\langle F_2,a_2,id\rangle}(c_1, w)\rangle$ at a fresh place $[U_i, h]$ in Store_π, unless unwrap returns $\perp$ on this input.

$\mathcal{F}_{KM}^{\texttt{impl},KW}$ chooses U_i and a new h exactly the same way. By s.c., we have, with equal probability, that $\langle F_1, a_1, c_1\rangle \in \mathit{Store}_{\mathcal{B}'_Z}(U_i, h_1)$. Since $F_1 = \texttt{KW}$, we will use $\hat{c}_1$ for the actual value in $\mathcal{B}_Z$'s store before substitution, i. e., given c_1, $\hat{c}_1$ is such that $k(\hat{c}_1) = c_1$.

- If $\hat{c}_1 \in \mathcal{K}_{cor}$ and $\hat{c}_1 \notin \mathcal{K}$, we have that $c_1 = \hat{c}_1$, and that $\texttt{key}[\hat{c}_1] = \hat{c}_1$ (only in step unwrap, a key that is not in $\mathcal{K}$ can be added to the store). Hence, $\mathcal{F}_{KM}^{\texttt{impl},KW}$ writes $\langle F_2, a_2, \texttt{unwrap}^{\langle F_2,a_2,id\rangle}(k(\hat{c}_1), w)\rangle$.
- If $\hat{c}_1 \in \mathcal{K}_{cor}$ and $\hat{c}_1 \in \mathcal{K}$, $k(\hat{c}_1) = c_1$. Since a key in $\mathcal{K}_{cor}$ but not $\mathcal{K}$ must have been added using the corruption procedure, we have that $\texttt{key}[\hat{c}_1] = c_1 = k(\hat{c}_1)$. Thus, $\mathcal{F}_{KM}^{\texttt{impl},KW}$ writes $\langle F_2, a_2, \texttt{unwrap}^{\langle F_2,a_2,id\rangle}(k(\hat{c}_1), w)\rangle$.
- If $\hat{c}_1 \notin \mathcal{K}_{cor}$ and w was recorded earlier, inspection of $\mathcal{F}_{KM}^{\texttt{impl},KW}$ shows that encs is written to only at the wrap step, which implies that $w = \texttt{wrap}^{\langle F_2,a_2,id\rangle}(k(\hat{c}_1), c_2)$ for some c_2. From the correctness of the scheme, we conclude that $\langle F_2, a_2, c_2 = \texttt{unwrap}^{\langle F_2,a_2,id\rangle}(k(\hat{c}_1), w)\rangle$ is written to this position.
- If $\hat{c}_1 \notin \mathcal{K}_{cor}$ and w is not recorded earlier, by definition of DEC, $\langle F_2, a_2, \texttt{unwrap}^{\langle F_2,a_2,id\rangle}(k(\hat{c}_1), w)\rangle$ is writ-

ten. (Same argument as in the first case, follows from s.c.)

i) Let $m = \langle \texttt{attr\_change}, h, a' \rangle$: The same code is executed in both $\mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},\mathcal{B}'_Z}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, thus p.c., s.c., and o.c. hold trivially.

j) Let $m = \langle C, \textit{public}, m \rangle$: The same code is executed in both $\mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},\mathcal{B}'_Z}$ and $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, thus p.c., s.c., and o.c. hold trivially.

3. Z sends a message to $U^{\text{ext}} \in \mathcal{U}^{\text{ext}}$.

   Both $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ and U^{ext} in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$ only accept messages of the form $\langle C, \textit{public}, m \rangle$ for $C \in \mathcal{C}_{i,\textit{pub}}$. Both perform the same computations, thus p.c., s.c., and o.c. hold trivially.

4. Z sends a message to <adv>.

   Both A_D and *Sim* ignore messages that are no instruction. So we can assume that Z instructs the adversary to send a message to some party.

   a) Assume Z instructs <adv> to send a message to a corrupted party, namely:

      i. $U_i \in \mathcal{U}$: U_i can only be addressed by the adversary if it was corrupted before, as otherwise it has never sent a message to the adversary. Note that the code run by U_i in $[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$, as well as in $\mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},\mathcal{B}'_Z}$, does not depend on any internal state. U_i can only talk to the environment, the adversary (*Sim* acts like A_D in this case) and it can call $\mathcal{F}_{\text{setup}}$, which *Sim* has to simulate in $\mathbf{Exp}^{\mathsf{wrap},}_{\mathsf{KW},\mathcal{B}'_Z}$. *Sim* is described above and receives all the information necessary to simulate it, that is: $\langle \texttt{ready}^\bullet, P \rangle$, when a protocol party in $\mathcal{U} \cup ST$ receives $\langle \texttt{ready} \rangle$ from the environment, $\langle \texttt{finish\_setup}^\bullet \rangle$, when a protocol party in $\mathcal{U}$ receives $\langle \texttt{ready} \rangle$ from the environment, and all messages that $\mathcal{F}_{\text{KM}}^{\texttt{impl},\mathsf{KW}}$ sends to the corrupted U_U (and $\mathcal{F}_{\text{setup}}$ would need to relay). Thus, if p.c. holds in the previous step, the invariant is preserved in case that the message is $\langle \texttt{ready}, U_i \rangle$ or $\langle \texttt{finish\_setup} \rangle$. A message of form $\langle \texttt{send}, \ldots \rangle$ is ignored by $\mathcal{F}_{\text{setup}}$ and *Sim*, so the invariant is trivially preserved here. The communication relayed in the steps `relay_receive` and `relay_send` in $\mathcal{F}_{\text{setup}}$ is simulated as described above and thus falls back to case 2.

      ii. $U_i^{\text{ext}} \in \mathcal{U}^{\text{ext}}$: Like in the previous case, only that $\mathcal{F}_{\text{setup}}$ ignores messages from U_i^{ext}, which *Sim* simulates correctly.

iii. other parties cannot become corrupted

b) Assume Z instructs <adv> to send a message to a party that cannot be corrupted, but that addressed <adv> before, i. e., $ST \in \mathcal{ST}$ or $\mathcal{F}_{\text{setup}}$. Since both ST and $\mathcal{F}_{\text{setup}}$ are specified to ignore messages in this case, *Sim* can simply mask their presence by reacting like ST or $\mathcal{F}_{\text{setup}}$ react upon reception of an unexpected message and answer with $\perp$.

We conclude that the invariant is preserved for an arbitrary number of steps. Since output consistency implies that Z has an identical view, the distribution of Z's output is the same in both games. Thus:

$$\Pr[b \leftarrow \text{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta) : b = 1] = \Pr[b \leftarrow \mathbf{Exp}^{\text{wrap},}_{\text{KW},\mathcal{B}'_Z}(\eta) : b = 1]$$

and therefore:

$$\begin{aligned}
&|\Pr[b \leftarrow \text{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z](\eta) : b = 1] \\
&\qquad - \Pr[b \leftarrow \text{Exec}[\mathcal{F}^{\texttt{impl}}_{\text{KM}}, \mathit{Sim}, Z](\eta) : b = 1]| \\
&= |\Pr[b \leftarrow \mathbf{Exp}^{\text{wrap,fake}}_{\text{KW},\mathcal{B}_Z}(\eta) : b = 1] \\
&\qquad - \Pr[b \leftarrow \mathbf{Exp}^{\text{wrap},}_{\text{KW},\mathcal{B}'_Z}(\eta) : b = 1]| \\
&> |\Pr[b \leftarrow \mathbf{Exp}^{\text{wrap,fake}}_{\text{KW},\mathcal{B}_Z}(\eta) : b = 1] \\
&\qquad - \Pr[b \leftarrow \mathbf{Exp}^{\text{wrap},}_{\text{KW},\mathcal{B}_Z}(\eta) : b = 1]| - \epsilon(\eta),
\end{aligned}$$

where ϵ is negligible in η. This contradicts the indistinguishability of $\text{Exec}[\mathcal{F}^{\texttt{impl}}_{\text{KM}}, \mathit{Sim}, Z]$ and $\text{Exec}[\pi_{\overline{\mathcal{F}},\overline{\mathcal{C}},\Pi,\overline{\texttt{Impl}}}, A_D, Z]$ and thus concludes the proof.

□

BIBLIOGRAPHY

[1] Martín Abadi and Véronique Cortier. 'Deciding Knowledge in Security Protocols Under Equational Theories.' In: *Automata, Languages and Programming* (2004), pp. 46–58.

[2] Martín Abadi and Cédric Fournet. 'Mobile values, new names, and secure communication.' In: *Principles of Programming Languages*. ACM, 2001, pp. 104–115.

[3] Ross J. Anderson. *Security engineering - a guide to building dependable distributed systems*. Wiley, 2001.

[4] Ross J. Anderson and Markus G. Kuhn. 'Low Cost Attacks on Tamper Resistant Devices.' In: *International Workshop on Security Protocols*. Springer, 1998, pp. 125–136.

[5] Myrto Arapinis, Eike Ritter, and Mark Dermot Ryan. 'StatVerif: Verification of Stateful Processes.' In: *Computer Security Foundations Symposium*. IEEE Computer Society, 2011, pp. 33–47.

[6] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. 'The AVISPA tool for the automated validation of internet security protocols and applications.' In: *Computer Aided Verification*. Springer, 2005, pp. 281–285.

[7] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra Abad. 'Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps.' In: *Workshop on Formal Methods in Security Engineering*. ACM, 2008, pp. 1–10.

[8] Michael Backes and Dennis Hofheinz. 'How to Break and Repair a Universally Composable Signature Functionality.' In: *Information Security* (2004), pp. 61–72.

[9] Ian Batten, Shiwei Xu, and Mark Ryan. 'Dynamic measurement and protected execution: model and analysis.' In: *Trustworthy Global Computing*. To appear. Spinger, 2013.

[10] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. 'A Concrete Security Treatment of Symmetric Encryption.' In: *Symposium on Foundations of Computer Science*. IEEE Computer Society, 1997, pp. 394–403.

[11] Stefano Bistarelli, Iliano Cervesato, Gabriele Lenzini, and Fabio Martinelli. 'Relating multiset rewriting and process algebras for security protocol analysis.' In: *Journal of Computer Security* 1 (2005), pp. 3–47.

[12] Fredrik Björck. *Security DJ Blog: Increased security for Yubikey*. 2009. URL: http://web.archive.org/web/20100725005817/http://security.dj/?p=154.

[13] Fredrik Björck. *Security DJ Blog: Yubikey Security Weaknesses*. 2009. URL: http://web.archive.org/web/20100203110742/http://security.dj/?p=4.

[14] Bruno Blanchet. 'An Efficient Cryptographic Protocol Verifier Based on Prolog Rules.' In: *Computer Security Foundations Workshop*. IEEE Computer Society, 2001, pp. 82–96.

[15] Bruno Blanchet, Martín Abadi, and Cédric Fournet. 'Automated Verification of Selected Equivalences for Security Protocols.' In: *Journal of Logic and Algebraic Programming* 1 (2008), pp. 3–51.

[16] Bruno Blanchet and Miriam Paiola. 'Automatic Verification of Protocols with Lists of Unbounded Length (long version).' to appear at CCS'13. 2013. URL: https://sites.google.com/site/ccs2013submission/.

[17] M. Bond and R. Anderson. 'API level attacks on embedded systems.' In: *IEEE Computer Magazine* (2001), pp. 67–75.

[18] Mike Bond and Piotr Zieliński. *Decimalisation table attacks for PIN cracking*. Tech. rep. University of Cambridge Computer Laboratory, 2003.

[19] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. *The quest to replace passwords: a framework for comparative evaluation of Web authentication schemes*. Tech. rep. UCAM-CL-TR-817. University of Cambridge, Computer Laboratory, 2012. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-817.pdf.

[20] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. 'Attacking and Fixing PKCS#11 Security Tokens.' In: *Computer and Communications Security*. ACM, 2010, pp. 260–269.

[21] C. Cachin and N. Chandran. 'A Secure Cryptographic Token Interface.' In: *Computer Security Foundations Symposium*. IEEE Computer Society, 2009, pp. 141–153.

[22] Ran Canetti. 'Universally Composable Security: A New Paradigm for Cryptographic Protocols.' In: *Foundations of Computer Science*. IEEE Computer Society, 2001, pp. 136–145.

[23] Ran Canetti. 'Universally Composable Signature, Certification, and Authentication.' In: *Computer Security Foundations Workshop*. IEEE Computer Society, 2004, pp. 219–233.

[24] Ran Canetti and Tal Rabin. 'Universal Composition with Joint State.' In: *Advances in Cryptology*. Springer, 2003, pp. 265–281.

[25] *CCA Basic Services Reference and Guide*. IBM. 2006. URL: http://www-03.ibm.com/security/cryptocards/pdfs/bs327.pdf.

[26] Vincent Cheval and Bruno Blanchet. 'Proving More Observational Equivalences with ProVerif.' In: *Principles of Security and Trust*. Springer, 2013, pp. 226–246.

[27] Jolyon Clulow. 'On the Security of PKCS#11.' In: *International Worshop on Cryptographic Hardware and Embedded Systems*. Springer, 2003, pp. 411–425.

[28] Irving M Copi. *Introduction to logic*. Macmillan, 1982.

[29] V. Cortier, G. Keighren, and G. Steel. 'Automatic Analysis of the Security of XOR-based Key Management Schemes.' In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2007, pp. 538–552.

[30] Véronique Cortier and Graham Steel. 'A generic security API for symmetric key management on cryptographic devices.' In: *European Symposium on Research in Computer Security*. Springer, 2009, pp. 605–620.

[31] Véronique Cortier, Graham Steel, and Cyrille Wiedling. 'Revoke and let live: a secure key revocation api for cryptographic devices.' In: *Computer and communications security*. ACM, 2012, pp. 918–928.

[32] Marion Daubignard, David Lubicz, and Graham Steel. *A Secure Key Management Interface with Asymmetric Cryptography*. Tech. rep. 2013. URL: http://hal.inria.fr/hal-00805987.

[33] Stéphanie Delaune, Steve Kremer, Mark D. Ryan, and Graham Steel. 'A Formal Analysis of Authentication in the TPM.' In: *Formal Aspects in Security and Trust*. Vol. 6561. Springer, 2010, pp. 111–125. URL: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/DKRS-fast10.pdf.

[34] Stéphanie Delaune, Steve Kremer, Mark D. Ryan, and Graham Steel. 'A formal analysis of authentication in the TPM.' In: *Formal Aspects in Security and Trust*. Springer, 2010, pp. 111–125.

[35] Stéphanie Delaune, Steve Kremer, Mark D. Ryan, and Graham Steel. 'Formal analysis of protocols based on TPM state registers.' In: *Computer Security Foundations Symposium*. IEEE Computer Society, 2011, pp. 66–82.

[36] Stéphanie Delaune, Steve Kremer, and Graham Steel. 'Formal Analysis of PKCS#11 and Proprietary Extensions.' In: *Journal of Computer Security* 6 (2010), pp. 1211–1245.

[37] G. Denker, J. Meseguer, and C. Talcott. 'Protocol Specification and Analysis in Maude.' In: *Workshop on Formal Methods and Security Protocols*. Springer, 1998.

[38] Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. 'Searching for Shapes in Cryptographic Protocols.' In: *Tools and Algorithms for the Construction and Analysis of Systems*. The Cryptographic Protocol Shapes Analyzer is available at http://hackage.haskell.org/package/cpsa. Springer, 2007, pp. 523–537.

[39] Danny Dolev and Andrew Chi-Chih Yao. 'On the security of public key protocols.' In: *Transactions on Information Theory* 2 (1983), pp. 198–207.

[40] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. 'Undecidability of Bounded Security Protocols.' In: *Workshop on Formal Methods and Security Protocols*. IEEE Computer Society, 1999.

[41] Santiago Escobar, Catherine Meadows, and José Meseguer. 'Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties.' In: *Foundations of Security Analysis and Design*. Vol. 5705. Springer, 2009, pp. 1–50.

[42] Eurosmart. *Eurosmart General Assembly Confirms Strong Growth*. Accessed: Fr 13 Sep 2013 14:51:39 IST. 2013. URL: http://www.eurosmart.com/index.php/publications/market-overview.html.

[43] FIPS-140-2. *Security Requirements for Cryptographic Modules*. 2004. URL: http://www.nist.gov/itl/upload/fips1402.pdf.

[44] Sibylle B. Fröschle and Nils Sommer. 'Reasoning with Past to Prove PKCS#11 Keys Secure.' In: *Formal Aspects in Security and Trust*. Vol. 6561. Springer, 2010, pp. 96–110.

[45] Sibylle Fröschle and Graham Steel. 'Analysing PKCS#11 Key Management APIs with Unbounded Fresh Data.' In: *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*. Springer, 2009, pp. 92–106.

[46] Juan A. Garay, Markus Jakobsson, and Philip D. MacKenzie. 'Abuse-Free Optimistic Contract Signing.' In: *Advances in Cryptology*. Springer, 1999, pp. 449–466.

[47] Joshua D. Guttman. 'State and Progress in Strand Spaces: Proving Fair Exchange.' In: *Journal of Automated Reasoning* 2 (2012), pp. 159–195.

[48] Thomas Habets. *YubiHSM login helper program*. Accessed: Wed 18 Sep 2013 15:07:42 CEST. 2011. URL: http://code.google.com/p/yhsmpam/.

[49] Jonathan Herzog. 'Applying Protocol Analysis to Security Device Interfaces.' In: *Security and Privacy* 4 (2006), pp. 84–87.

[50] Dennis Hofheinz. *Possibility and impossibility results for selective decommitments*. Cryptology ePrint Archive. 2008. URL: http://eprint.iacr.org/.

[51] Dennis Hofheinz and Victor Shoup. *GNUC: A New Universal Composability Framework*. Cryptology ePrint Archive. 2011. URL: http://eprint.iacr.org/.

[52] ITU Telecommunication Development Bureau. *6.8 billion mobile-cellular subscriptions*. Accessed: Fr 13 Sep 2013 15:08:38 IST. 2013. URL: http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013.pdf.

[53] Horace William Brindley Joseph. *An Introduction to Logic*. Clarendon Press, 1916.

[54] George Hayward Joyce. *Principles of logic*. Green, 1949.

[55] Kamikaze28 et al. *Specification of the Yubikey operation in the Yubico wiki*. 2012. URL: http://wiki.yubico.com/wiki/index.php/Yubikey.

[56] D. Kaminsky. *On The RSA SecurID Compromise*. 2011. URL: http://dankaminsky.com/2011/06/09/securid/.

[57] Steve Kremer, Robert Künnemann, and Graham Steel. 'Universally Composable Key-Management.' In: *European Symposium on Research in Computer Security*. Springer, 2013, pp. 327–344.

[58] Steve Kremer, Graham Steel, and Bogdan Warinschi. 'Security for Key Management Interfaces.' In: *Computer Security Foundations Symposium*. IEEE Computer Society, 2011, pp. 66–82.

[59] Robert Künnemann and Graham Steel. *Source files and proofs for the analysis of the Yubikey protocol*. 2013. URL: http://www.lsv.ens-cachan.fr/~kunneman/yubikey/analysis/yk.tar.gz.

[60] Robert Künnemann and Graham Steel. 'YubiSecure? Formal Security Analysis Results for the Yubikey and YubiHSM.' In: *Workshop on Security and Trust Management*. Springer, 2012, pp. 257–272.

[61] Ralf Küsters and Max Tuengerthal. 'Ideal Key Derivation and Encryption in Simulation-Based Security.' In: *Topics in Cryptology*. Springer, 2011, pp. 161–179.

[62] Ralf Küsters and Max Tuengerthal. 'Joint State Theorems for Public-Key Encryption and Digitial Signature Functionalities with Local Computation.' In: *Computer Security Foundations Symposium*. IEEE Computer Society, 2008, pp. 270–284.

[63] Ralf Küsters and Max Tuengerthal. *The IITM Model: a Simple and Expressive Model for Universal Composability*. Tech. rep. 2013/025. Cryptology ePrint Archive, 2013. URL: http://eprint.iacr.org/.

[64] D. Longley and S. Rigby. 'An Automatic Search for Security Flaws in Key Management Schemes.' In: *Computers & Security* 1 (1992), pp. 75–89.

[65] Gavin Lowe. 'An attack on the Needham-Schroeder public-key authentication protocol.' In: *Information Processing Letters* 3 (1995), pp. 131–133.

[66] Gavin Lowe. 'Breaking and fixing the Needham-Schroeder public-key protocol using FDR.' In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1996, pp. 147–166.

[67] Ueli Maurer and Renato Renner. 'Abstract Cryptography.' In: *Innovations in Computer Science*. Tsinghua University Press, 2011, pp. 1–21.

[68] Simon Meier. *Contract Signing Protocol (Example 2 from [5])*. Accessed: Tue 17 Sep 2013 14:28:06 CEST. 2012. URL: https://github.com/tamarin-prover/tamarin-prover/blob/develop/data/examples/related_work/StatVerif_ARR_CSF11/StatVerif_GM_Contract_Signing.spthy.

[69] Simon Meier. 'Formal Analysis of Key-Exchange Protocols and Physical Protocols.' PhD thesis. ETH Zürich, 2013.

[70] Simon Meier. *Simple security device (Example 1 from [5])*. available in the `example/` directory of the tamarin distribution. Accessed: Tue 17 Sep 2013 14:31:32 CEST. 2012. URL: https://github.com/tamarin-prover/tamarin-prover/blob/develop/data/examples/related_work/StatVerif_ARR_CSF11/StatVerif_Security_Device.spthy.

[71] Simon Meier. *The keyserver example from [74]*. Accessed: Tue 17 Sep 2013 14:23:53 CEST. 2012. URL: https://github.com/tamarin-prover/tamarin-prover/blob/develop/data/examples/related_work/AIF_Moedersheim_CCS10/Keyserver.spthy.

[72] Simon Meier. *The TESLA protocol, scheme 1 and 2*. available in the `example/` directory of the tamarin distribution. Accessed: Thu 19 Sep 2013 14:36:52 CEST. 2012. URL: https://github.com/tamarin-prover/tamarin-prover/blob/develop/data/examples/loops/TESLA_Scheme[1,2].spthy.

[73] Simon Meier, Cas J. F. Cremers, and David A. Basin. 'Strong Invariants for the Efficient Construction of Machine-Checked Protocol Security Proofs.' In: *Computer and Communications Security*. ACM, 2010, pp. 231–245.

[74] Sebastian Mödersheim. 'Abstraction by set-membership: verifying security protocols and web services with databases.' In: *Computer and Communications Security*. ACM, 2010, pp. 351–360.

[75] The Canadian Press – OBJ. *PCs still more popular than tablets, smarpthones: Deloitte*. Accessed: Fr 13 Sep 2013 15:44:46 IST. 2013. URL: http://www.obj.ca/Canada%20-%20World/2013-01-15/article-3156753/PCs-still-more-popular-than-tablets-smarpthones-Deloitte/1.

[76] David Oswald, Bastian Richter, and Christof Paar. 'Side-Channel Attacks on the Yubikey 2 One-Time Password Generator.' In: *Research in Attacks, Intrusions and Defenses*. Springer, 2013.

[77] Catuscia Palamidessi. 'Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculi.' Anglais. In: *Mathematical Structures in Computer Science* 5 (2003), pp. 685–719. URL: http://hal.inria.fr/inria-00201104.

[78] Adrian Perrig, J. D. Tygar, Dawn Song, and Ran Canetti. 'Efficient Authentication and Signing of Multicast Streams over Lossy Channels.' In: *Security and Privacy*. IEEE Computer Society, 2000, pp. 56–73.

[79] Alfredo Pironti, Davide Pozza, and Riccardo Sisto. 'Formally-Based Semi-Automatic Implementation of an Open Security Protocol.' In: *Journal of Systems and Software* (2012), pp. 835–849.

[80] Alfredo Pironti and Riccardo Sisto. 'Provably Correct Java Implementations of Spi Calculus Security Protocols Specifications.' In: *Computers & Security* (2010), pp. 302–314.

[81] *PKCS #11: Cryptographic Token Interface Standard*. RSA Security Inc. 2004.

[82] *PKCS#11 v2.20: Cryptographic Token Interface Standard. Available from http://www.rsa.com/rsalabs*. 2004.

[83] AVISPA project. *Deliverable 2.3: The Intermediate Format*. 2003. URL: http://www.avispa-project.org.

[84] R. Küsters. 'Simulation-Based Security with Inexhaustible Interactive Turing Machines.' In: *Computer Security Foundations Workshop*. IEEE Computer Society, 2006, pp. 309–320.

[85] R. Küsters and T. Truderung. 'Reducing Protocol Analysis with XOR to the XOR-free Case in the Horn Theory Based Approach.' In: *Journal of Automated Reasoning* 3 (2011), pp. 325–352.

[86] P. Rogaway and T. Shrimpton. *Deterministic Authenticated Encryption: A Provable-Security Treatment of the Keywrap Problem*. 2006.

[87] Mark Ryan. *Introduction to the TPM 1.2*. Tech. rep. University of Birmingham, 2009.

[88] Benedikt Schmidt. 'Formal Analysis of Key-Exchange Protocols and Physical Protocols.' PhD thesis. ETH Zürich, 2012.

[89] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. 'Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties.' In: *Computer Security Foundations Symposium*. IEEE Computer Society, 2012, pp. 78–94.

[90] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. 'The TAMARIN Prover for the Symbolic Analysis of Security Protocols.' In: *Computer Aided Verification*. Springer, 2013, pp. 696–701.

[91] D. Shaw. *The OpenPGP HTTP Keyserver Protocol (HKP)*. Internet-Draft. Internet Engineering Task Force, 2003. URL: http://tools.ietf.org/html/draft-shaw-openpgp-hkp-00.

[92] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. 'Strand Spaces: Why is a Security Protocol Correct?' In: *Security and Privacy*. IEEE Computer Society, 1998, pp. 160–171.

[93] *The YubiKey Manual - Usage, configuration and introduction of basic concepts (Version 2.2)*. Yubico AB. 2010. URL: http://www.yubico.com/documentation.

[94] The yubikey-val-server-php project. *Validation Protocol Version 2.0*. 2011. URL: http://code.google.com/p/yubikey-val-server-php/wiki/ValidationProtocolV20.

[95] *Trusted Platform Module Library, Family 2.0*. Committee Draft. Trusted Computing Group, 2013. URL: http://www.trustedcomputinggroup.org/resources/tpm_library_specification.

[96] Loredana Vamanu. 'Formal Analysis of Yubikey.' Master's Thesis. École normale supérieure de Cachan, 2011. URL: http://n.ethz.ch/~lvamanu/download/YubiKeyAnalysis.pdf.

[97] Christoph Weidenbach. 'Combining Superposition, Sorts and Splitting.' In: *Handbook of Automated Reasoning* (2001), pp. 1965–2013.

[98] Christoph Weidenbach. 'Towards an Automatic Analysis of Security Protocols in First-Order Logic.' In: *Automated Deduction*. Vol. 1632. Springer, 1999, pp. 314–328.

[99] D. Whiting, R. Housley, and N. Ferguson. *Counter with CBC-MAC (CCM)*. RFC 3610 (Informational). Internet Engineering Task Force, 2003. URL: http://www.ietf.org/rfc/rfc3610.txt.

[100] *Yubicloud Validation Service - (Version 1.1)*. Yubico AB. 2012. URL: http://www.yubico.com/documentation.

[101] Yubico AB. *Department of Defence: Moving from legacy authentication to Yubico technology and best practice security processes*. Accessed: Wed 17 Jul 2013 11:20:48 CEST. 2013. URL: https://www.yubico.com/about/reference-customers/department-defence/.

[102] Yubico AB. *Yubico customer list*. Accessed: Wed 17 Jul 2013 11:40:50 CEST. 2013. URL: https://www.yubico.com/about/reference-customers/.

[103] Yubico AB. *YubiKey NEO*. Accessed: Wed 17 Jul 2013 13:25:32 CEST. 2013. URL: www.yubico.com/products/yubikey-hardware/yubikey-neo/.

[104] Yubico AB. *YubiKey Security Evaluation: Discussion of security properties and best practices*. v2.0. 2009. URL: http://static.yubico.com/var/uploads/pdfs/Security_Evaluation_2009-09-09.pdf.

[105] Yubico Inc. *YubiHSM 1.0 security advisory 2012-01*. 2012. URL: http://static.yubico.com/var/uploads/pdfs/SecurityAdvisory%202012-02-13.pdf.

[106] *Yubico YubiHSM - Cryptographic Hardware Security Module (Version 1.0)*. Yubico AB. 2011. URL: http://www.yubico.com/documentation.

[107] *YubiKey Authentication Module Design Guide and Best Practices (Version 1.0)*. Yubico AB. 2011. URL: http://www.yubico.com/documentation.