

THÈSE
PRÉSENTÉE À
L'UNIVERSITÉ BORDEAUX I
ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE
Par **Philippe ANDOUARD**
Pour obtenir le grade de
DOCTEUR
SPÉCIALITÉ : INFORMATIQUE

**Outils d'aide à la recherche de vulnérabilités dans
l'implantation d'applications embarquées sur carte à puce**

Soutenue le : 18 décembre 2009

Après avis des rapporteurs :

M. Louis Goubin Professeur des Universités
M. Belhassen Zouari . Professeur des Universités

Devant la commission d'examen composée de :

M. Serge Chaumette .	Professeur des Universités	Président
M. Louis Goubin	Professeur des Universités	Rapporteur
M. Ly Olivier	Maître de conférence	Directeur
M. Mosbah Mohamed	Professeur des Universités	Directeur
M. Davy Rouillard . . .	Ingénieur de Recherche . .	Examineur
M. Belhassen Zouari .	Professeur des Universités	Rapporteur

Remerciements

Je tiens à remercier très chaleureusement Olivier Ly mon directeur de thèse sans qui cette aventure n'aurait pas vu le jour. Je souhaite lui exprimer ma plus profonde gratitude pour m'avoir accepté comme élève puis guidé et soutenu tout au long de ces dernières années.

Je remercie Louis Goubin ainsi que Belhassen Zouari pour m'avoir fait l'honneur d'avoir accepté d'être mes rapporteurs de thèse.

Je voudrais remercier très chaleureusement Céline Thuillet avec qui j'ai eu le privilège de travailler au cours de cette thèse. Son efficacité, son soutien et son amitié m'ont été très précieuses.

Je remercie mes grand-mères car elles m'ont toujours encouragé à poursuivre dans la voie qui me plaisait.

Je tiens aussi à remercier Pascale, Luc, Guillaume et Pierre-Victor pour leur soutien, leur amour et les liens forts qui nous unissent.

Je voudrais remercier ma maman pour m'avoir épauler et parce qu'elle a toujours su être là quand j'en avais besoin. D'ailleurs, si elle ne m'avait pas poussé aux études, ce manuscrit n'aurait sans doute jamais vu le jour . . .

Ma dernière pensée sera pour Laure. Je la remercie pour sa compréhension, son soutien et son amour qui ont été sans faille durant ces derniers mois difficiles.

Résumé : Les travaux présentés dans cette thèse ont pour objectif de faciliter les évaluations sécuritaires des logiciels embarqués dans les cartes à puce. En premier lieu, nous avons mis au point un environnement logiciel dédié à l'analyse de la résistance d'implémentations d'algorithmes cryptographiques face à des attaques par analyse de la consommation de courant. Cet environnement doit être vu comme un outil pour rechercher des fuites d'information dans une implémentation en vue d'évaluer la faisabilité d'une attaque sur le produit réel. En second lieu, nous sommes intéressés à l'analyse de programmes écrits en langage d'assemblage AVR dans le but de vérifier s'ils sont vulnérables aux *timing attacks*. Nous avons donc développé un outil qui consiste à décrire des chemins du flot de contrôle d'un programme grâce à des expressions régulières qui seront par la suite interprétées par notre outil afin de donner leur temps exact d'exécution (en terme de cycles d'horloge). Enfin, nous avons étudié comment faciliter la compréhension de programmes écrits en langage C dans le but de vérifier si des politiques de sécurité sont correctement implémentées. D'une part, nous fournissons des assistants de navigation qui au travers d'informations concernant les variables et procédures rencontrées, facilitent la compréhension du programme. D'autre part, nous avons au point une manière de vérifier les politiques de sécurité sans modélisation préalable (e.g. avec un automate à états finis) au moyen de requêtes exprimées dans la logique CTL.

Mots-clefs : Carte à puce, sécurité, microcontrôleurs, *side channel attacks*, langage d'assemblage AVR, langage C.

Abstract : The work presented in this thesis aims at easing the evaluation process of smartcards embedded software. On one hand, we set up a software environment dedicated to analyze the implementation resistance of cryptographic to power analysis attacks. This environment must be seen as a tool that facilitates a real attack by giving a way to find information leakages in an implementation. On the other hand, we focused on analyzing program written in AVR assembly language in order to check whether they are vulnerable to timing attacks. To achieve this goal we have developed a tool that makes possible the description of a path in the control flow of the program thanks to regular expressions. Those regular expressions will be interpreted by our tool in order to give the exact execution timing (expressed in clock cycles). Finally, we studied how to ease the global comprehension of a program written in C language in order to check whether security policies are well implemented. First, we provide graphical navigation assistants that helps to understand the program being analyzed by giving information on variables and procedures. Then, we provide a way to check the security policies through the use of requests expressed with the CTL logic. This approach does not need prior modelisation of the program.

Keywords : Smartcard, security, microcontrollers, side channel attacks, AVR assembly language, C language.

Table des matières

Introduction	1
1 La carte à puce	5
1.1 Historique de la carte à puce	5
1.1.1 Les bandes magnétiques	6
1.1.2 Les cartes à mémoire	6
1.1.3 Les cartes intelligentes	7
1.1.4 Applications des cartes à puce	7
1.2 Caractéristiques techniques des <i>smart cards</i>	8
1.2.1 Le micromodule	8
1.2.2 Les microprocesseurs dédiés aux cartes	9
1.2.3 Les différents types de mémoires	10
1.2.4 Communication	12
1.2.5 Systèmes d'exploitation	14
1.3 La sécurité des cartes à puce	16
1.4 La certification de la sécurité	17
1.5 Processus d'évaluation	19
1.5.1 Les Critères Communs	19
1.5.2 L'évaluation	21
1.6 Conclusion	22
2 Les microcontrôleurs	25
2.1 Architecture d'un microcontrôleur	26
2.1.1 Les registres	27
2.1.2 La mémoire	29
2.1.3 Les bus	30
2.1.4 Fonctionnement	31
2.1.5 Jeu d'instructions	31
2.2 Conclusion	33
3 Les attaques	35
3.1 Les attaques par modification permanente	36
3.1.1 Le micro-sondage	36

3.1.2	La modification de circuits	36
3.2	Les attaques par injection de fautes	36
3.2.1	Injection par les paramètres fonctionnels d'une puce	37
3.2.2	Injection par rayonnement	37
3.2.3	Analyse différentielle de fautes	38
3.3	Les attaques par canaux auxiliaires	38
3.3.1	Les attaques basées sur le temps d'exécution	38
3.3.2	Les attaques par analyse de la consommation de courant	39
3.3.3	Les attaques par analyse des émissions électromagnétiques	41
3.4	Les attaques par mauvaise utilisation	42
3.5	L'arrachage	42
3.6	Contre-mesures	43
3.6.1	Micro-sondage	43
3.6.2	Injection de fautes	43
3.6.3	<i>Timing attacks</i>	45
3.6.4	Analyse du courant et des émissions électromagnétiques	45
3.7	Conclusion	48
4	Simulateur d'un microcontrôleur de carte à puce	49
4.1	Contexte	50
4.2	Approche retenue	50
4.3	Relation entre résistance théorique et résistance réelle	51
4.4	Étude de l'existant	52
4.4.1	Les simulateurs classiques	52
4.4.2	Les simulateurs d'attaques	53
4.5	Présentation du simulateur	54
4.5.1	Fonctionnement du simulateur	54
4.5.2	Scénario d'attaque physique	56
4.5.3	Performances	57
4.6	Modèles de consommation de courant	57
4.6.1	Le modèle distance de Hamming	57
4.6.2	Le modèle poids de Hamming	58
4.7	Modèle de consommation abstrait	58
4.8	Attaque différentielle sur le DES	59
4.8.1	Le système de chiffrement DES	60
4.8.2	Differential Power Analysis sur le DES	61
4.9	DPA théorique sur le DES	62
4.9.1	DPA théorique	62
4.9.2	Introduction du concept de « DPA chirurgicale »	63
4.9.3	« DPA chirurgicale » appliquée aux boîtes-S et aux bus	65
4.9.4	« DPA chirurgicale » et permutation	65
4.9.5	Commentaires sur la DPA théorique	69
4.10	Conclusion	71

5	Analyse de programmes écrits en langage d'assemblage	73
5.1	Approche retenue	74
5.2	Étude de l'existant	75
5.3	Représentation visuelle de langages d'assemblage	75
5.3.1	Quelques définitions	75
5.3.2	Graphe de flot de contrôle	77
5.3.3	Graphe d'appel	78
5.3.4	Limitations à la reconstruction de graphes pour le langage AVR	79
5.3.5	Exemple de résultats obtenus pour la représentation visuelle .	79
5.4	Évaluation de chemins	80
5.4.1	Expressions régulières	81
5.4.2	Système d'équations de langage	82
5.4.3	Instanciation d'expressions régulières	83
5.4.4	Méthode d'évaluation	83
5.5	Ergonomie	87
5.5.1	Blocs de base et nombre de cycles associés	87
5.5.2	Affichage du code inatteignable	89
5.5.3	Visualisation d'impacts d'attaques par injection de fautes . . .	89
5.6	Conclusion	91
6	Analyse de programmes écrits en langage de haut niveau	93
6.1	Analyse sécuritaire de code source	94
6.1.1	Fonctionnalités pour un outil d'aide à l'analyse de code	94
6.2	Étude de l'existant	95
6.3	Approche retenue	96
6.4	CIL	96
6.4.1	Analyses fournies par CIL	96
6.4.2	CIL et graphe de flot de contrôle	97
6.4.3	Conclusion sur CIL	97
6.5	Architecture	98
6.6	Recherche de vulnérabilités	98
6.6.1	Les déclarations dangereuses de tableaux	99
6.6.2	Les valeurs faibles	100
6.7	Les assistants de navigation	103
6.7.1	Variables et procédures	103
6.7.2	Les rapports	104
6.8	Graphes d'appel et précision	105
6.8.1	Navigation dans le flot de contrôle	105
6.8.2	Application à la sécurité	106
6.9	Conclusion	107

7	Model checking sur des programmes écrits en C	109
7.1	Approche retenue	109
7.1.1	Exemples de requêtes	110
7.2	Étude de l'existant	111
7.2.1	MOPS	111
7.2.2	Propriétés de sécurité et graphe de flot de contrôle	112
7.2.3	Aoraï	112
7.2.4	Goanna	113
7.2.5	Choix technique	113
7.3	Vérification et <i>model-checking</i>	113
7.3.1	Modélisation	114
7.3.2	Spécification et logiques temporelles	114
7.4	Modèle de programme	116
7.5	Sémantique, notion d'exécution et CTL	118
7.5.1	Validité	118
7.6	Programme et satisfaction de formules logiques	119
7.6.1	Satisfaction de formules logiques	120
7.6.2	Précision	122
7.7	Présentation de l'outil	124
7.8	Conclusion	126
	Conclusion	127

Table des figures

1	La première carte intelligente.	7
2	Micromodule d'une carte à puce avec contact.	9
3	Caractéristiques des différents types de mémoires.	11
4	Architecture d'un microprocesseur de carte à puce.	12
5	Structure d'une commande <i>APDU</i>	13
6	Structure d'un <i>APDU</i> de réponse.	13
7	Architecture des plates-formes multi-applicatives et ouvertes.	15
8	Les différents intervenants et leurs rôles dans le schéma français.	18
9	Échelle d'assurance des Critères Communs.	21
10	Architecture de type Harvard.	27
11	Architecture détaillée des familles AT90S8515 et ATmega128.	28
12	Cartographie de la mémoire Flash.	29
13	Espace d'adressage des données.	30
14	Format d'une instruction 16 bits et sa mnémonique correspondante.	33
15	Principe d'une <i>timing attack</i>	38
16	Architecture de l'environnement de simulation	54
17	Exemple de scénario visant à attaquer le registre de travail R1 et le compteur ordinal.	56
18	Un tour de DES.	60
19	La fonction f du DES.	61
20	Exemple d'une trace de consommation théorique pour l'algorithme DES.	63
21	Code C de la fonction de permutation.	68
22	Ordre de mise à jour des bits permutés.	68
23	Découpage en blocs de base.	77
24	Ajout des arêtes entre les blocs de base.	78
25	Exemple de graphe d'appel.	78
26	Décrémenter d'un registre.	80
27	Représentation des graphes orientés.	81

28	Expression régulière de tous les chemins du graphe de flot de contrôle du schéma 27a.	82
29	Instanciation d'une expression régulière.	83
30	Procédure de comparaison de tableaux de caractères.	84
31	Graphe de flot de contrôle de la procédure <code>verify_pin</code>	85
32	Architecture client-serveur.	87
33	Coloration syntaxique de code AVR dans Eclipse.	88
34	Affichage des blocs de base et de leurs nombres de cycles d'horloge.	89
35	Marquage du code inatteignable.	90
36	Syntaxe des déclarations de CIL.	97
37	Architecture client-serveur.	98
38	Tableau de taille variable.	99
39	Expression conditionnelle faible.	101
40	Graphe d'appel.	103
41	Graphe d'appel inverse.	104
42	Vérification des éléments générés par une analyse.	105
43	Graphes d'appel obtenus par navigation dans le flot de contrôle.	106
44	Exemple d'une analyse intra-procédurale de graphe d'appel.	107
45	Graphe de flot de contrôle et structure de Kripke.	118
46	Exécution et chemin dans le graphe de flot de contrôle.	119
47	Programme à analyser.	124
48	Interpréteur de formules CTL.	125
49	Réponse du <i>model-checker</i>	125
50	Chemin calculé par l'outil.	126

Introduction

La carte à puce connaît un essor retentissant depuis une vingtaine d'année et au fil du temps elle a su se rendre indispensable dans notre vie quotidienne. En effet, que ce soit dans des domaines aussi variés que les transports, la santé ou la monétique, la carte à puce est devenue incontournable. Son usage s'est tellement banalisé qu'on en oublie presque que ce petit bout de plastique surmonté d'un microprocesseur renferme des quantités d'informations confidentielles.

Considérons par exemple la carte vitale, notre carte d'assurance maladie, qui contient quantités d'informations sur nos antécédents médicaux. Personne ne souhaiterait que ces informations soient facilement accessibles et tombent entre les mains d'individus peu scrupuleux. Par exemple, une compagnie ayant eu accès illégalement au passé médical d'une personne pourrait refuser d'embaucher cette dernière, jugeant qu'elle présente un risque trop grand d'être fréquemment malade. Ou encore des compagnies d'assurance pourraient tenter d'exploiter ces informations dans le but de définir des profils à risques plus précis, leur permettant d'augmenter leurs tarifs ou de refuser un contrat d'assurance à une personne. Il en va de même pour la carte bancaire, nous nous attendons à ce qu'elle présente un niveau de sécurité suffisant pour que le code confidentiel (PIN) qu'elle abrite ne soit connu que de nous. Ainsi, le marché grandissant des cartes à puce doit faire face à des enjeux sécuritaires importants.

Une carte à puce est facilement transportable et c'est justement ce caractère portable qui l'expose à diverses menaces. En effet, elle évolue généralement dans un environnement qui n'est pas de confiance, ce qui fait qu'elle est la cible privilégiée de toute une gamme d'attaques visant à mettre à mal sa sécurité. Aussi pratique soit-elle, il faut que l'utilisateur final puisse avoir une confiance absolue dans la sécurité intrinsèque de la carte qu'il utilise quotidiennement.

Afin de garantir le niveau de sécurité souhaité, des laboratoires indépendants appelés Centres d'Évaluation de la Sécurité des Technologies de l'Information ont été créés. Ces laboratoires sont composés d'experts dont les compétences vont de l'électronique à l'informatique en passant par la cryptographie et dont le travail consiste à évaluer la résistance d'une carte donnée face à des attaques très pointues.

Dans ce contexte, notre objectif est de fournir des outils qui permettent à ces experts d'une part d'améliorer et d'automatiser la recherche de vulnérabilités et d'autre part de mettre en place des attaques spécifiques.

Objet des travaux

L'« intelligence » des cartes à puce repose entièrement sur le microcontrôleur qu'elles embarquent. Il s'agit en fait d'un microprocesseur associé à de la mémoire dont l'architecture et le fonctionnement sont en tout point comparables à un micro-ordinateur classique. Néanmoins, sans être exactement soumises aux mêmes menaces qu'un micro-ordinateur, il existe des attaques qui visent à déterminer quelles sont les données secrètes ou sensibles que le microcontrôleur manipule (e.g. clef de chiffrement). Notre attention s'est portée sur deux types d'attaques en particulier : les attaques par canaux auxiliaires et les attaques par injection de fautes. Les attaques par canaux auxiliaires (ou *side channel attacks*) sont des attaques non intrusives qui consistent à observer les effets physiques liés à des calculs tels que le temps d'exécution (*timing attack*) ou la consommation de courant (e.g. *Differential Power Analysis* (DPA)) pour retrouver les données secrètes manipulées par un algorithme lors de son exécution. La seconde catégorie concerne les attaques par injection de fautes qui visent à modifier (souvent à l'aide de lasers) des parties du microcontrôleur telles que la mémoire ou les registres dans le but de dérouter le flot de contrôle du programme et de passer outre des procédures de sécurité.

Travaux

Nos travaux ont consisté à développer une plate-forme qui simule le comportement d'un microcontrôleur de carte à puce. En premier lieu, cette plate-forme permet de conduire des attaques physiques qui consistent à perturber le fonctionnement du microprocesseur en injectant des fautes durant le déroulement d'un programme. En second lieu, elle permet d'établir des profils de consommation théoriques des programmes exécutés rendant possible la mise en place d'attaques par DPA. Ces travaux ont permis de faire émerger la notion de « DPA chirurgicale ». Ainsi, grâce à notre plate-forme de simulation, il est possible de vérifier si la consommation de composants très précis du microcontrôleur (e.g. registres, mémoires, etc.) peut mener au secret manipulé.

Puis, toujours dans le but de découvrir des vulnérabilités, nous avons proposé un outil de recherche de *timing attacks* dans le code source de programmes écrits en langage d'assemblage. Il s'agit d'un outil d'analyse semi-automatique (e.g. guidé par l'évaluateur) qui reconstruit le graphe de flot de contrôle d'un programme pour faciliter sa compréhension et fournit un moyen d'analyser le temps d'exécution de chemins d'exécution spécifiques au travers d'un interpréteur d'expressions régulières.

Enfin, nos travaux s'intéressent à faciliter l'analyse de programmes écrits en langage C. Notre approche consiste à fournir un outil ergonomique qui soit facilement utilisable et qui s'intègre facilement dans l'un des outils de navigation utilisés par les évaluateurs. Au vu de ces contraintes, nous nous sommes orientés vers un outil qui vient se greffer dans l'environnement de développement Eclipse. Plus exactement,

cet outil permet de simplifier la navigation dans le code source, d'automatiser la recherche de vulnérabilités liées aux attaques physiques et enfin d'aider à vérifier si les politiques de sécurité ont été correctement implémentées. En premier lieu, la navigation dans le code est simplifiée en donnant des informations sur des variables, des procédures et propose une façon de naviguer dans le flot de contrôle dans le but de créer un graphe d'appel plus précis. En second lieu, il existe certaines valeurs de constantes présentes dans des expressions conditionnelles qui peuvent être facilement perturbées par une attaque physique; notre outil se propose d'automatiser leur recherche. Enfin, les évaluateurs sont souvent amenés à vouloir vérifier des propriétés de sécurité telles que :

- dans la procédure f , existe-t-il un chemin d'exécution sur lequel la procédure de vérification de code PIN n'est pas appelée ?
- le buffer ayant manipulé une clef est-il systématiquement effacé ?

Pour les aider dans cette analyse, notre outil se propose de formaliser ces requêtes au moyen de la logique temporelle CTL et de vérifier ces propriétés en se basant sur le graphe de flot de contrôle du programme en cours d'analyse.

Organisation de la thèse

Le **Chapitre 1** présente le contexte de la carte à puce. Le lecteur y trouvera un rapide historique ainsi qu'une description des principales caractéristiques des matériels et architectures dédiés. Il introduit aussi la notion de sécurité présente dans les cartes à puce et explique comment cette sécurité est évaluée.

Le **Chapitre 2** traite des microcontrôleurs. L'objectif de ce chapitre est de présenter tous les éléments nécessaires concernant l'architecture et le langage d'assemblage utilisés par deux familles de microcontrôleurs généralistes qui ont servi de bases à nos travaux.

Le **Chapitre 3** dépeint quelles sont les principales attaques qui peuvent mettre à mal la sécurité des cartes à puce. Il aborde aussi les contre-mesures qui peuvent être mises en place pour faire face à ces attaques, aussi bien au niveau matériel que logiciel.

Le **Chapitre 4** présente un simulateur de microcontrôleur, qui permet d'établir des profils de consommations théoriques des programmes qu'il exécute en vue de simuler des attaques physiques comme par exemple une *Differential Power Analysis*.

Le **Chapitre 5** décrit un outil de recherche de *timing attacks* dans des programmes écrits en langage d'assemblage. Cet outil reconstruit le graphe de flot de contrôle d'un programme pour faciliter sa compréhension et fournit un moyen d'analyser le temps d'exécution de chemins au travers d'un interpréteur d'expressions régulières.

Le **Chapitre 6** présente un outil destiné à faciliter l'analyse de code écrit en langage C. Cet outil permet de simplifier la navigation dans le code source, d'automatiser la recherche de vulnérabilités liées aux attaques physiques et enfin de vérifier si des politiques de sécurité ont été correctement implémentées.

Le **Chapitre 7** se focalise sur la vérification de propriétés de sécurité sur le graphe de flot de contrôle d'un programme écrit en langage C via des requêtes exprimées au moyen de la logique temporelle CTL.

Chapitre 1

La carte à puce

Depuis son invention il y a maintenant trente ans, la carte à puce n'a cessé d'évoluer et d'occuper une place de plus en plus importante dans notre vie quotidienne. Son premier succès industriel porta sur la production de Télécartes destinées à donner accès aux cabines téléphoniques publiques. Depuis, elle a tellement su se fondre dans notre vie que nous ne prêtons même plus attention à tous les services qu'elle nous rend au quotidien. Pourtant, elle a remplacé la monnaie, permet de se connecter à un réseau de téléphonie mobile et Internet ou encore d'utiliser les transports en commun. Son succès est dû à deux principaux facteurs que sont sa taille réduite qui permet de la glisser dans une poche ou un portefeuille mais surtout à l'ensemble des protections qui assurent la sécurité des données qu'elle stocke.

Ce chapitre a pour vocation de donner au lecteur les clefs permettant d'appréhender les principales caractéristiques des cartes à puces. Nous souhaitons présenter l'évolution de la carte à puce depuis sa création jusqu'à ce qu'elle est devenue aujourd'hui. Cette rétrospective nous permettra d'appréhender quelle est sa place dans l'économie actuelle et de mieux comprendre pourquoi sa sécurisation constitue un enjeu important.

1.1 Historique de la carte à puce

Des cartes en plastique permettant d'effectuer des paiements voient le jour aux États Unis, dans les années cinquante. La toute première carte de crédit autorisant les paiements inter-régionaux a été émise par le *Diners' Club*. Ce club très fermé permettait à ses membres d'accéder à une sélection de restaurants et de payer avec une simple signature, le paiement étant différé. Utilisée par 200 privilégiés dans 14 restaurants lors de sa création, elle compte 20 000 clients et est acceptée dans 1000 restaurants l'année d'après.

Cependant, les protections proposées par les premières cartes en plastiques pour se protéger des contrefaçons étaient rudimentaires car elles reposaient sur des caractéristiques visuelles tels que le champ de signature et une empreinte de sécurité. Par conséquent, la sécurité du système reposait entièrement sur le souci du détail

du personnel acceptant la carte. Avec la prolifération croissante des cartes de crédit, ces protections rudimentaires ne suffisaient plus, il était nécessaire d'améliorer les mécanismes de sécurité de la carte.

1.1.1 Les bandes magnétiques

La première amélioration a consisté à ajouter une bande magnétique au dos des cartes de crédit. Cette avancée permettait de stocker des informations qui ne pourraient être lues qu'avec une machine adéquate, ajoutant une couche de sécurité supplémentaire aux données visuelles gravées en relief. En France, la première carte comportant une piste magnétique a été mise en service en 1971 par l'organisation Carte Bleue, en même temps que les premiers distributeurs de billets automatiques.

Néanmoins, la technologie des bandes magnétiques souffrait d'une faiblesse cruciale : leur contenu pouvait être facilement lu, reproduit voire même effacé. Par conséquent, elles ne pouvaient servir à stocker des données confidentielles et les systèmes qui les employaient se virent dans l'obligation d'être connectés en ligne avec l'ordinateur du système hôte. Les coûts élevés inhérents aux transmissions de données, poussèrent à chercher des solutions alternatives.

1.1.2 Les cartes à mémoire

Les énormes progrès réalisés par la microélectronique dans les années soixante ont rendu possible l'intégration d'une mémoire et d'une unité arithmétique et logique sur un même composant mesurant quelques millimètres carrés. Par ailleurs, l'idée d'utiliser un circuit intégré pour des cartes d'identités a fait l'objet d'un dépôt de brevet en 1968 par deux ingénieurs allemands, Jürgen Dethloff et Helmut Grötrupp. Peu de temps après, une application similaire fait elle aussi l'objet d'un dépôt de brevet au Japon par Kunikata Arimura en 1970. Toujours la même année, l'américain Jules K. Ellingboe décrit de façon concrète un moyen de paiement électronique sur une carte à contact.

Cependant, c'est en 1974 que le français Roland Moreno réalise le premier prototype *d'objet portatif à mémoire intelligente*. Son prototype rappelle l'objet que décrit René Barjavel dans son roman intitulé «La Nuit des Temps». Il s'agissait d'un anneau pourvu de moyens de mémorisation et de communication, utilisé par une civilisation vieille de milliers d'années : les Gondas. Plus précisément, le prototype de Roland Moreno apparaît sous la forme d'une bague surmontée d'une mémoire PROM (*Programmable Read-Only Memory*) protégée par des moyens inhibiteurs assurant entre autre l'intégrité du contenu de certaines parties de la mémoire. Il dépose alors le brevet couvrant cette technique ce qui fera de lui l'inventeur de la carte à puce.

1.1.3 Les cartes intelligentes

Il y a très peu voire pas de sécurité dans une carte à mémoire car elle n'embarque aucune intelligence capable de réagir en cas d'intrusion. Le besoin d'un produit flexible, sûr, fiable et économique mais qui soit en plus capable d'évoluer et d'être normalisé s'est très rapidement fait sentir. En d'autres termes une carte à base de microprocesseur. C'est ainsi qu'en 1977, le français Michel Ugon, de Bull, dépose le brevet de la première carte *intelligente* (voir [1]). Il s'agit d'une carte à microprocesseur comprenant de la mémoire non volatile programmable permettant à la carte de fonctionner comme un véritable micro-ordinateur.

L'année suivante, en 1978, Michel Ugon continue sur sa lancée et brevète la SPOM (*Self Programmable One Chip Micro-Computer*), une nouvelle architecture pour un microprocesseur capable de traiter de façon sûre des applications multiples grâce aux opérations d'auto-programmation à l'intérieur d'une mémoire non volatile. Cet aspect d'auto-programmation permet à un microprocesseur de modifier son comportement suivant certaines données voire même de s'autodétruire en cas d'alerte.

Finalement, le 21 mars 1979, la première carte à microprocesseur voyait le jour, fruit de la collaboration entre CII-Honeywell Bull et Motorola (voir figure 1). Cette



FIG. 1: La première carte intelligente.

carte embarquait deux puces : une mémoire EPROM¹ 2716 et un microprocesseur 8 bits 3870 fourni par Motorola : la carte intelligente (*smart card* en anglais) était née (voir [2]).

1.1.4 Applications des cartes à puce

Un intense travail de standardisation impliquant l'ISO (*International Organization of Standardization*) et l'IEC (*International Electrotechnical Commission*) ont donné lieu à de nombreuses normes spécifiant les caractéristiques physiques et les protocoles de transmission des données des cartes à puce (voir par exemple [3, 4]). Cet énorme travail de standardisation a contribué au déploiement massif des cartes à puce dans différents pays et dans des domaines d'applications très variés.

Aujourd'hui, le principal domaine d'utilisation reste la téléphonie mobile où la carte à puce, appelée carte *SIM* (*Subscriber Identify Module*) regroupe toutes les

¹*Erasable PROM* : type de mémoire autorisant une ou plusieurs re-programmation ultérieures.

fonctionnalités concernant l'accès au réseau global de communications mobiles *GSM* (*Global System for Mobile Communication*). Elle sert à authentifier l'utilisateur sur le réseau et contient les clefs de chiffrements permettant d'assurer la confidentialité des communications.

L'autre grand domaine d'application concerne les applications bancaires avec les cartes de crédit et les porte-monnaies électroniques. Dans ce domaine, les trois organisations internationales de cartes de crédit, Europay, MasterCard et Visa ont travaillé conjointement afin de développer des spécifications communes pour l'utilisation des cartes de paiement et ceci en vue d'assurer une compatibilité mutuelle. La première version de cette spécification est sortie en 1994 sous le nom d'*EMV*.

Le domaine de la santé utilise aussi les cartes à puce afin d'améliorer la sécurité et la confidentialité des données du patient. Par exemple, la carte peut contenir l'identification du patient, les informations liées à l'assurance maladie ou encore des données médicales permettant un suivi du patient.

Le secteur des transports a lui aussi grandement bénéficié de l'apparition des cartes sans contacts, qui permettent de rapidement s'identifier sans avoir besoin de la glisser la carte dans un terminal (e.g. le passe Navigo proposé par la RATP).

Le domaine de la sécurité informatique n'est pas en reste et utilise les cartes à puce afin de stocker des données confidentielles. Par exemple, certains navigateurs sont capables d'utiliser des certificats stockés sur des cartes à puce pour faire de l'authentification forte et naviguer de façon plus sûre sur l'Internet. De plus, certains systèmes de chiffrement de disques durs utilisent des cartes à puce pour sauvegarder de façon sécurisée la clef utilisée.

Enfin, les problèmes liés à l'identification ont poussé certains gouvernements à se servir des cartes à puce comme carte d'identité des citoyens. C'est notamment le cas avec le passeport électronique qui renferme le nom, la date de naissance du détenteur ainsi que le bureau de délivrance du passeport et d'identifiants biométriques (empreintes digitales et photos) et qui assure, par des moyens cryptographiques qu'il ne s'agit pas d'un faux ou d'un clone.

1.2 Caractéristiques techniques des *smart cards*

Le matériel embarqué dans une carte à puce est fortement contraint car les normes ISO (voir [3, 5]) imposent une surface de silicium inférieure à 27 millimètres carrés. Cette limite entraîne donc une architecture matérielle minimaliste aussi bien en terme de puissance de calcul que de capacité mémoire.

1.2.1 Le micromodule

Une carte à puce est une carte en plastique flexible (généralement du polychlorure de vinyle (PVC)) dans laquelle est inséré un *micromodule*. Un micromodule (voir figure 2) est constitué d'un circuit intégré, comprenant un microprocesseur et

de la mémoire, sur lequel est posé un ensemble de contacts (*microcontact*). Le microcontact est relié au circuit intégré via des fils d'or, permettant à la carte d'être alimentée en courant et de communiquer avec l'extérieur. La surface du microcontact possède huit points de connexions :

- Vcc qui correspond à la tension d'alimentation de la puce et qui est généralement de 5 volts,
- RST (*Reset*) la ligne de signal utilisée pour initialiser l'état du circuit intégré après sa mise sous tension,
- CLK (*Clock*) le signal d'horloge qui est utilisé pour piloter la vitesse du circuit intégré. Cette horloge sert aussi de référence pour les communications,
- GND (*Ground*) la tension de référence,
- Vpp qui sert à fournir une alimentation suffisante pour programmer de la mémoire de type EPROM. Néanmoins, ce point de contact tend à ne plus trop être utilisé,
- IO (*Input/Output*) qui correspond à la ligne d'entrée/sortie grâce à laquelle la puce reçoit des commandes et échange des données avec le monde extérieur,
- RFU (*Reserved for Future Use*) qui correspond à des contacts servant à étendre les fonctionnalités de la carte. Ces deux derniers contacts sont actuellement utilisés pour communiquer en USB (*Universal Serial Bus*).

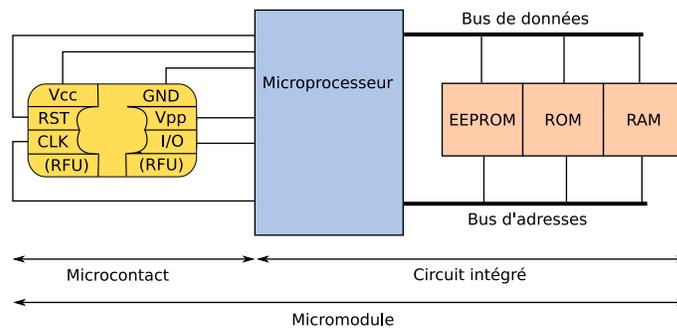


FIG. 2: Micromodule d'une carte à puce avec contact.

1.2.2 Les microprocesseurs dédiés aux cartes

Le microprocesseur constitue le cœur de la carte car aussi bien les échanges avec l'extérieur que le traitement des données passent par lui. Dans cette section, notre but n'est pas de dresser une liste exhaustive de tous les microprocesseurs présents dans les cartes à puce mais plutôt de donner au lecteur une idée des types de microprocesseurs utilisés et de leur puissance. Le tableau 1 présente les caractéristiques de trois microprocesseurs massivement utilisés dans les cartes à puce.

En premier lieu, nous retrouvons la famille de microprocesseurs ST19, basée sur une architecture CISC ² huit bits. Proposés par la société ST MICROELECTRONICS,

²Complex Instruction Set Computer

Dénomination	ST19	AVR	ARM7
Architecture	CISC	RISC	RISC
Bus de données	8 bits	8 bits	32 bits
Nb. registres	8	32	16
Fréquence max.	> 20Mhz	16 Mhz	> 40Mhz

TAB. 1: Caractéristiques des microprocesseurs dédiés aux cartes à puce.

ces microcontrôleurs sont très performants et les modèles les plus puissants possèdent huit kilooctets de mémoire RAM et des modules cryptographiques *matériels* permettant d'exécuter les algorithmes DES ou AES (*Advanced Encryption Standard* voir [6]).

Ensuite, viennent les microprocesseurs AVR proposés par la société ATMEL[®]. Il s'agit de microprocesseurs RISC³ huit bits très répandus dans le monde de l'embarqué car de nombreux outils libres existent pour les programmer. De plus, la syntaxe du langage d'assemblage relativement simple et épurée contribue à la facilité de prise en main de ces microprocesseurs. Cette famille de microprocesseurs a servi de base pour nos travaux et pour cette raison le chapitre 2 lui sera consacré.

Enfin, nous avons les microprocesseurs ARM7 qui sont basés sur une technologie RISC 32 bits et conçus par la société ARM. Cette gamme de microprocesseurs présente la particularité de posséder des instructions dont le format est régulier et codé sur 32 bits. Les familles de microprocesseurs basées sur la technologie ARM sont très performantes ce qui explique leur grande diffusion, outre les cartes à puce, dans des périphériques tels que des téléphones mobiles dernière génération, des consoles de jeux ou encore des cartes graphiques.

1.2.3 Les différents types de mémoires

Une carte à puce possède trois types de mémoires différents dont chacun présente des propriétés qui lui sont propres. Nous détaillons les principaux types de mémoires dans ce qui suit.

Random Access Memory. La mémoire vive ou RAM (*Random Access Memory*) est la mémoire de travail du microprocesseur. Cette mémoire est volatile ce qui signifie que lorsque la carte n'est plus alimentée en courant, toutes les données s'y trouvant sont perdues. Ses temps d'accès sont très rapides aussi bien en lecture qu'en écriture et elle sert à stocker de manière temporaire des données obtenues durant l'exécution d'un programme. La mémoire RAM utilisée dans les cartes à puce est dite statique (*Static RAM*), c'est-à-dire que son contenu n'a pas besoin d'être rafraîchi périodiquement. Autrement dit, elle ne dépend pas d'une horloge externe ce qui contraste avec la RAM dynamique (*Dynamic RAM*) qui, si l'horloge

³Reduced Instruction Set Computer

est stoppée, perd toutes les informations qu'elle contient. Cependant, elle reste très coûteuse et pour cette raison sa taille n'excède pas huit kilooctets.

Read Only Memory. La mémoire ROM (*Read Only Memory*) aussi appelée *masked-ROM* est une mémoire persistante et non modifiable. Autrement dit, elle n'a pas besoin d'énergie pour sauvegarder l'information qu'elle contient et elle sert à stocker des données qui n'ont pas besoin d'être modifiées comme par exemple un système d'exploitation. Suivant les cartes, sa taille oscille entre trente-deux et cent vingt-huit kilooctets.

Non-Volatile Memory. Les mémoires de type NVM (*Non-Volatile Memory*) présentent le double avantage de pouvoir modifier les données qu'elles contiennent et de les conserver même lorsque la carte est hors tension. À l'heure actuelle, les mémoires non volatiles les plus utilisées sont l'EEPROM (*Electrical Erasable Programmable Read Only Memory*) et la *Flash*.

L'EEPROM est très pratique puisqu'elle peut être effacée électriquement mais présente deux inconvénients majeurs. D'une part elle s'use rapidement (i.e. elle ne permet que 100 000 cycles d'écritures et d'effacements) et son temps de rétention d'information est limité à 10 ans. D'autre part sa latence en écriture (i.e. son temps d'accès aux données) est mille fois supérieure à celle de la RAM du fait de son effacement adresse par adresse.

La mémoire *Flash* quant à elle possède de grandes capacités de stockage, des temps rapides d'effacement et elle n'occupe pas beaucoup de place sur la puce. Cette mémoire présente ainsi une excellente alternative à la mémoire ROM car elle peut contenir la totalité du système d'exploitation. De ce fait, la ROM ne sert plus qu'à stocker des données nécessaires au bon démarrage de la carte.

Récapitulatif. Le tableau ci-dessous (voir figure 3) résume les principales caractéristiques des mémoires rencontrées dans les cartes à puce. Une cellule RAM prend

	RAM	ROM	EEPROM	Flash
Volatile	oui	non	non	non
Durabilité	∞	-	$\sim 100\ 000$	$\sim 10\ 000$
Coût	très élevé	bas	élevé	moyen
Taille d'une cellule	$\sim 1700\mu m^2$	$\sim 100\mu m^2$	$\sim 400\mu m^2$	$\sim 200\mu m^2$
Cycle de lecture	$60ns$	$150ns$	$150ns$	$80 - 120ns$
Cycle d'écriture	$70ns$	-	$5 - 10ms/octet$	$10 - 17\mu s$

FIG. 3: Caractéristiques des différents types de mémoires.

quatre fois plus de place qu'une cellule EEPROM qui elle même prend quatre fois plus de place qu'une cellule de ROM. Cette contrainte de place associée aux coûts

des mémoires oblige les constructeurs à trouver des compromis pour obtenir des produits performants et abordables.

Nous noterons que certaines cartes possèdent en plus un coprocesseur cryptographique permettant d'optimiser l'exécution d'algorithmes de chiffrement tels que le RSA ou le DES. Ce coprocesseur ne possède pas de mémoire dédiée et utilise la mé-

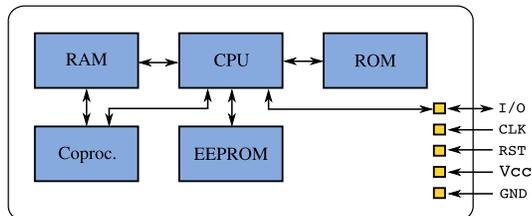


FIG. 4: Architecture d'un microprocesseur de carte à puce.

moire RAM présente sur la carte pour stocker les résultats de calculs intermédiaires (voir figure 4).

1.2.4 Communication

Comme nous l'avons vu, la carte à puce est un périphérique très compact doté d'une puissance de calcul raisonnable et en ce sens, elle peut être comparée à un micro-ordinateur. Toutefois, une carte à puce est un ordinateur un peu particulier qui communique avec le monde extérieur au travers d'un lecteur qui l'alimente en courant électrique. La communication en mode contact entre une carte et un lecteur (aussi appelé terminal ou *Card Acceptance Device (CAD)*), se fait en mode *half-duplex*⁴ via un unique contact celui d'entrée/sortie. Les cartes à puce sont toujours remises à zéro lorsqu'elles sont insérées dans un lecteur. Cette action a pour effet d'obliger la carte à émettre ce qui s'appelle une réponse au démarrage (*Answer-to-reset (ATR)*). Cette réponse indique au lecteur les capacités de la carte à puce concernant les protocoles de communications (e.g. protocoles T=0 ou T=1⁵, vitesse de transmission, etc.). L'ATR contient aussi des informations dites historiques concernant par exemple l'identification du produit, sa version, le type de puce utilisé ou encore l'état de la carte.

La communication se fait alors par échange de paquets de données, appelés *APDU (Application Protocol Data Unit)*. Un *APDU* peut être une commande

⁴À un instant donné, seule la carte ou le lecteur envoie des données mais pas les deux en même temps.

⁵T=0 indique que les données seront transmises octet par octet alors qu'en mode T=1 elles seront transmises par blocs.

provenant d'un lecteur vers une carte (*C-APDU*) ou une réponse d'une carte au lecteur (*R-APDU*). Le mode de discussion instauré entre une carte et un lecteur est dit de type maître-esclave car le lecteur dirige la communication et la carte ne fait qu'exécuter les commandes.

***APDU* de commande**

Un *APDU* d'instruction est constitué d'un en-tête et d'un corps dont la longueur peut être arbitraire voire nulle quand le champ de données est vide (cf. figure 5). L'en-tête est composé des quatre éléments qui sont la classe **CLA**, l'instruction **INS**

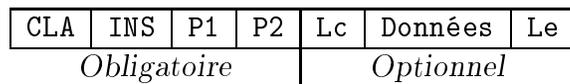


FIG. 5: Structure d'une commande *APDU*.

et les paramètres 1 et 2 (**P1** et **P2**). L'octet de classe est utilisé pour identifier les applications et leur jeu d'instruction spécifique. Par exemple, la téléphonie GSM utilise l'octet de classe **A0** alors que les instructions basées ISO sont encodées par l'octet **0X**. Ensuite, vient l'octet qui identifie l'instruction de la commande. Les deux octets **P1** et **P2** sont utilisés afin de paramétrer l'instruction sélectionnée par l'octet d'instruction.

Le corps d'un *APDU* joue un double rôle. D'une part, il permet de fixer la longueur des données à envoyer à la carte (i.e. champ **Lc**) et de spécifier la longueur des données que le lecteur devra renvoyer à la carte (i.e. champ **Le**). D'autre part, il contient les données à envoyer à la carte et qui sont relatives à l'instruction en cours d'exécution.

***APDU* de réponse**

Un *APDU* de réponse est envoyé à la carte en réponse à un *APDU* de commande et il est constitué d'un corps optionnel mais d'un en-queue obligatoire (cf. figure 6).

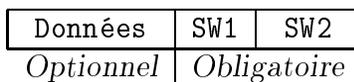


FIG. 6: Structure d'un *APDU* de réponse.

Le corps est constitué d'un champ de données dont la longueur a été déterminée dans l'octet **Le** de la précédente *APDU* de commande. L'en-queue quant à elle contient le code de retour de l'instruction précédemment exécutée. Ce code est stocké dans les deux octets **SW1** et **SW2**. Par exemple, le code de retour **0x9000** signifie que l'instruction s'est complètement déroulée et de façon normale.

1.2.5 Systèmes d'exploitation

Le système d'exploitation est la couche intermédiaire entre les applications et le matériel. Il répond à trois critères qui sont la gestion du matériel, l'abstraction du matériel (en fournissant aux concepteurs d'applications une manière conviviale d'utiliser le support physique) et la sécurité des programmes. Cette partie a pour vocation de donner un aperçu des systèmes d'exploitation (pour plus de détails voir [7]) depuis l'invention des cartes à puce jusqu'à l'invention des plates-formes ouvertes apparues au milieu des années quatre vingt dix.

Les systèmes monolithiques. La toute *première génération* d'architecture logicielle embarquée dans les puces consistait en une unique application mise au point par les fabricants de cartes et qui correspondait d'une part aux exigences du client et d'autre part aux spécifications du microprocesseur. Cette application était ensuite transmise au fondeur afin de la *masquer*⁶ dans la mémoire ROM des cartes. On ne pouvait pas à proprement parler de véritable système d'exploitation.

La *seconde génération* de systèmes était basée sur la réutilisabilité de fragments de code. Les fabricants de cartes avaient rapidement compris qu'en réutilisant le plus possible des portions de code, ils économisaient énormément de temps et réduisaient par la même occasion le temps critique de la mise sur le marché. La seconde génération était caractérisée par un monolithe décomposé en trois parties :

- un ensemble de modules de gestion du matériel,
- un ensemble d'applications modulaires bien connues et utilisées par les applications comme par exemple la gestion du code PIN,
- le code correspondant aux exigences de l'application cible et qui n'ont pas encore été remplis par les précédents modules.

Notons que dans les deux premières générations de logiciels, il était impossible de modifier le programme embarqué dans la carte après sa mise en circulation.

Les environnements d'exécution dédiés. Les plates-formes standards, qui peuvent être considérées comme une combinaison de modules de gestion du matériel et de classes d'applications, sont embarquées dans la puce et peuvent être par la suite spécialisées par les fabricants de cartes en assemblant des composants additionnels appelés *filtres*. Ces filtres sont des fonctions qui n'ont pas encore été incorporées dans la plate-forme standard comme par exemple un mode de gestion spécial du PIN. L'application elle même est alors ajoutée en définissant des structures de données et en les remplissant avec les données adéquates. L'étape finale peut être réalisée par des fournisseurs de services même si traditionnellement, cette étape est laissée aux fabricants de cartes.

⁶Un masque consiste à indiquer l'emplacement des 0 et 1 dans la matrice en silicium qui constitue la mémoire ROM.

Les plates-formes multi-applicatives et ouvertes. Les exigences requises pour une gestion adéquate de la mémoire, de portabilité et d'interopérabilité associées aux avancées de la technologie des microprocesseurs a mené à de nouvelles initiatives. Les principaux objectifs étaient l'indépendance vis-à-vis du système d'exploitation sous-jacent, la capacité de manipuler de façon sécurisée plusieurs applications et aussi de pouvoir modifier le contenu de la carte une fois mise en circulation.

C'est ainsi que les plates-formes *multi-applicatives* et *ouvertes* sont apparues offrant la possibilité d'héberger plusieurs applications qui peuvent provenir de différents fournisseurs et être chargées à n'importe quel moment. Néanmoins, les applications ne s'exécutent pas de façon concurrente car les systèmes d'exploitation ne gèrent pas (encore) plusieurs processus à la fois. Les principaux standards de cartes multi-applicatives sont :

- *Java Card* (voir [8]) développé en 1996 par Schlumberger et qui dérive de la technologie Java,
- *Multos* (voir [9]) mis au point par un consortium industriel nommé MAOSCO composé entre autre d'American Express, MasterCard International, Siemens et Motorola,
- *Basic Card* (voir [10]) développé par ZeitControl CardSystems et qui permet la programmation d'applications en langage BASIC,
- *Windows for Smart Cards* proposée par Microsoft en 1998 mais qui n'est plus utilisée aujourd'hui.

Toutes ces plates-formes reposent sur une *machine virtuelle*, c'est-à-dire une couche logicielle qui interprète un langage intermédiaire d'instructions. Plus exactement, la machine virtuelle permet d'abstraire la couche matérielle en la simulant (voir figure 7). Une application est traduite dans le langage de la machine virtuelle. Cette

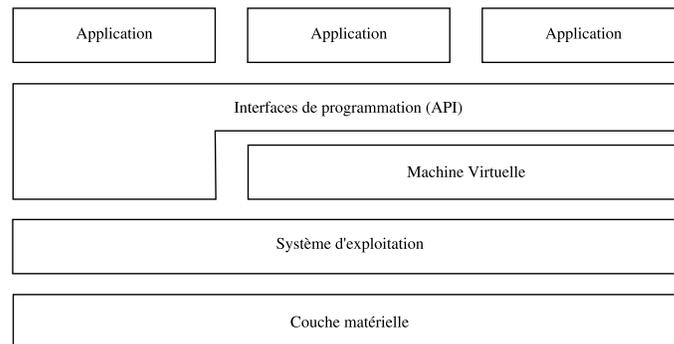


FIG. 7: Architecture des plates-formes multi-applicatives et ouvertes.

dernière va alors *interpréter* (exécuter) ces instructions en les transformant en instructions natives propres au processeur. Il existe deux principaux avantages au fait d'utiliser une machine virtuelle. D'une part, une application peut être programmée indépendamment du microprocesseur sur lequel elle sera exécutée (seule la machine virtuelle devra être portée). D'autre part, les différents contrôles (accès aux données, typages, etc.) réalisés lors de l'interprétation du code intermédiaire renforce la

sécurité du système.

1.3 La sécurité des cartes à puce

Les cartes à puce sont de véritables ordinateurs aux capacités réduites dont le but est d'offrir un niveau de sécurité élevé pour les données et les programmes. Elles incorporent des algorithmes de chiffrements et intègrent des protocoles élaborés en vue d'assurer la protection des données de la carte et des communications. La carte à puce est donc bien un maillon important d'un système de sécurité dont la défaillance peut entraîner l'obtention illégale de biens ou de services. Rendre les cartes plus sûres nécessite de travailler à trois niveaux : au niveau physique (le corps de la carte), au niveau matériel des composants et au niveau logiciel.

Au niveau physique. Le corps de la carte doit répondre à certaines exigences sécuritaires. La carte embarque des protections visuelles pour complexifier la tâche des contrefacteurs. Il peut y avoir différentes couches d'impressions à diverses profondeurs dans le plastique de la carte. De plus, le corps de la carte embarque souvent un hologramme et les données concernant son porteur sont gravées dans le plastique.

Comme nous le verrons dans la section 3, certaines attaques nécessitent un accès direct au circuit intégré de la carte. Afin d'empêcher la manipulation du micro-module, diverses méthodes d'encollage ont été développées visant à rendre la puce inutilisable lors de son extraction.

Au niveau matériel. La qualité de la couche matérielle est primordiale pour assurer le bon fonctionnement du microprocesseur et de la mémoire, mais c'est surtout sur elle que repose la sécurité des cartes à puce.

Certaines attaques consistent à perturber le fonctionnement de la puce. Le but de ces perturbations peut consister à modifier le contenu d'une case mémoire ou à essayer de sauter des instructions dans le programme en cours d'exécution. Afin d'éviter que le circuit intégré soit affecté par des perturbations extérieures, les fabricants ont intégré différents types de détecteurs. Par exemple, nous retrouvons des détecteurs de lumière, des détecteurs de température ou encore des détecteurs de tension et de fréquence.

Comme nous le verrons dans la section 3, certaines attaques se basent sur la consommation de courant ou du rayonnement électromagnétique des circuits intégrés pour en déduire de l'information sur ce qui est manipulé. Si les détecteurs empêchent fortement les circuits intégrés d'être perturbés, ils ne les protègent en rien contre des observations passives. Pour pallier à ces problèmes, les fabricants utilisent principalement deux protections à savoir un mécanisme de pompe à charge qui lisse la consommation globale de courant de la carte à puce et une grille de protection qui limite les émissions électromagnétiques.

Enfin, nous terminerons notre aperçu des mécanismes sécuritaires de la couche matérielle en évoquant la possibilité d'enfourer des bus dans les différents niveaux de métallisation, de créer des pistes qui ne sont reliées à aucune autre ou encore de chiffrer les bus ou les mémoires.

Au niveau logiciel. La confiance accordée à une carte à puce repose fortement sur sa couche logicielle. Les développeurs d'applications sont à l'état de l'art des attaques qu'il est possible de mener et adaptent donc leur façon de programmer en conséquence. Par exemple, les procédures de bas niveau, considérées comme critiques du point de vue de la sécurité (e.g. opérations cryptographiques, comparaison de données, etc.), sont souvent programmées en langage d'assemblage. Utiliser un langage de si bas niveau présente l'intérêt de pouvoir précisément étudier le déroulement des instructions et des données qu'elles manipulent. Ce niveau de précision permet de maîtriser le temps d'exécution du programme en fonction des données qu'il manipule. Par exemple, si un programme s'exécute en temps constant quelles que soient les données qu'il manipule, alors un observateur extérieur ne pourra faire aucune corrélation entre les données manipulées et le temps d'exécution.

Précédemment, nous avons évoqué des attaques qui consistent à sauter des instructions dans le programme. Afin de se prémunir contre ce type d'attaques, certains développeurs d'application parsèment leur code de *balises* spécifiques. Vérifier si une exécution se déroule correctement consiste à s'assurer que le flot d'exécution du programme est bien passé par les balises qu'il devait rencontrer.

Parmi tous les mécanismes sécuritaires logiciels, nous pouvons aussi citer ceux mis en œuvre par le système d'exploitation. D'une part, le système implémente une politique très stricte de contrôle d'accès aux données. En effet, l'accès aux données nécessite la vérification de règles qui dépendent des opérations à réaliser. De plus, le système s'assure de la bonne intégrité des données via des mécanismes de somme de contrôle (*checksum*) sur toute ou partie de la mémoire. Enfin, le système s'occupe aussi des mécanismes de transactions et d'atomicité des opérations.

1.4 La certification de la sécurité

La différenciation entre chaque fabricant de cartes à puce se fait généralement au niveau de la sécurité. Cette partie a pour vocation d'expliquer comment un produit ou un système issus des Technologies de l'Information comme une carte à puce se voit délivrer un certificat permettant d'établir un niveau de confiance dans sa sécurité intrinsèque. Un processus de certification fait intervenir trois tiers indépendants qui sont :

- le commanditaire de l'évaluation (e.g. un développeur) souhaitant obtenir un certificat pour son produit,
- l'organisme de certification qui analyse la pertinence des rapports émis par le centre d'évaluation et qui décide ou non d'accorder un certificat,

- le centre d'évaluation (nommé CESTI⁷) qui vérifie la sécurité du produit en question.

L'organisme de certification est chargé de vérifier la pertinence et la qualité des analyses menées par le centre d'évaluation, garantissant ainsi un niveau de confiance que nous pouvons avoir dans un produit. Ce dernier point est très important car les évaluations sécuritaires sont des activités commerciales : c'est donc au commanditaire de choisir le centre d'évaluation qui établira la sécurité de son produit. Ainsi, le contrôle extérieur assuré par l'organisme de certification garantit la qualité des tests qui ont été menés sur un produit et permet donc d'éviter les fraudes (e.g. qu'un commanditaire achète son certificat). Nous allons à présent décrire l'organisation du schéma français de certification.

Le schéma français

Dans le schéma français (voir figure 8) de certification, nous retrouvons toujours les trois acteurs évoqués précédemment à savoir : le commanditaire, l'organisme de certification et le centre d'évaluation plus un quatrième qui est un organisme d'accréditation.

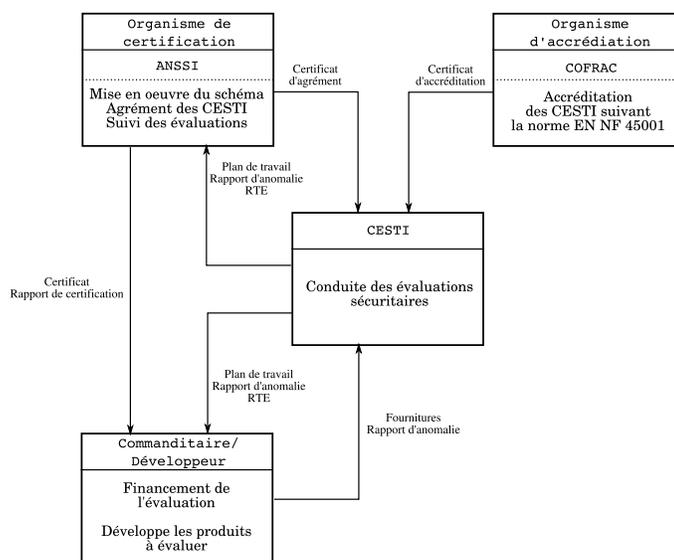


FIG. 8: Les différents intervenants et leurs rôles dans le schéma français.

Le commanditaire. Le commanditaire est la personne qui souhaite s'assurer de la sécurité de son produit et qui finance la certification. Au travers de l'obtention d'un certificat, les commanditaires prouvent leurs compétences et peuvent en faire un argument marketing en vue d'accroître leurs parts de marché.

⁷ Acronyme signifiant Centre d'Évaluation de la Sécurité des Technologies de l'Information.

Comité Français d'Accréditation. L'accréditation est l'attestation de la compétence, de l'impartialité et de l'indépendance d'un organisme certificateur, d'un laboratoire ou d'un organisme d'inspection au regard des normes en vigueur (par exemple, la norme NF EN 45011 pour les organismes certificateurs de produits).

Afin de pouvoir réaliser une évaluation, un CESTI doit être accrédité par le COFRAC (COmité FRançais d'ACcréditation voir [11]) qui est une association loi 1901 créée en 1994 sous l'égide des pouvoirs publics. Au premier juillet 2008, sont accrédités en France environ 1600 laboratoires, 244 organismes d'inspection et 97 organismes de certification de produits, services, entreprises et personnels.

Organisme de certification. Dans le schéma français, l'organisme de certification est un organisme gouvernemental appelé l'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI voir [12]). Cette agence a différentes missions dont assurer la mission d'autorité nationale en matière de sécurité des systèmes d'information et assurer un service de veille, de détection, d'alerte et de réaction aux attaques informatiques, notamment sur les réseaux de l'État.

Centre d'Évaluation de la Sécurité des Systèmes d'Information. Les CESTI sont constitués d'une équipe d'experts participant à la réalisation des évaluations : ils agissent en tant que tierce partie, indépendante des développeurs de produits et des commanditaires. Ils doivent être agréés par l'organisme de certification et à ce titre sont tenus de respecter toutes les règles du schéma. Un CESTI doit être impartial et indépendant de toute pression extérieure. Il ne doit en aucun cas être impliqué dans le développement (y compris à titre de conseil) et l'évaluation d'un même produit.

À l'heure actuelle, il existe cinq CESTI agréés en France. Dans le domaine technique des logiciels et équipements réseaux nous retrouvons SILICOMP - AQL et OPPIDA. Dans le domaine technique des composants électroniques, microélectroniques et des logiciels embarqués, nous avons le CEA - LETI, le CEACI (THALES SECURITY SYSTEMS - CNES) et SERMA TECHNOLOGIES.

1.5 Processus d'évaluation

Dans ce qui suit, nous allons brièvement aborder la norme qui régit les évaluations ce qui nous permettra de mieux appréhender et comprendre le métier des évaluateurs de SERMA TECHNOLOGIES durant l'analyse sécuritaire d'un produit.

1.5.1 Les Critères Communs

Nous avons parlé des différents acteurs qui entrent en jeu lors d'un processus d'évaluation sécuritaire sans toutefois expliquer comment sont organisées ces évaluations. En effet, la délivrance d'un certificat assurant la sécurité d'un produit

passer par l'utilisation de critères d'évaluation bien précis qui sont décrits par des normes dont les Critères Communs (voir [13]). Ces critères d'évaluation résultent de la volonté d'unifier la multitude de systèmes d'évaluation utilisés par différents pays :

- les *Trusted Computer System Evaluation Criteria* pour les États-Unis,
- les *Canadian Trusted Computer Product Evaluation Criteria* au Canada,
- les *Information Technology Security Evaluation Criteria* pour les Pays-Bas, l'Allemagne, le Royaume-Uni et la France.

Cette volonté d'uniformisation donna lieu aux Critères Communs qui furent institués comme norme ISO en 1999 (voir [14, 15, 16]). Les Critères Communs sont génériques et servent de base pour l'évaluation des propriétés de sécurité des produits et systèmes des Technologies de l'Information tels que des logiciels pare-feu, des boîtiers de chiffrement ou encore des cartes à puce.

Périmètre de sécurité

Les différents acteurs entrant en jeu dans une évaluation doivent disposer d'un langage commun leur permettant de définir ce qui doit être évalué. Ainsi, les Critères Communs exigent l'établissement de trois documents décrivant les concepts sécuritaires du produit à évaluer, à savoir :

- une cible d'évaluation (*Target Of Evaluation*), c'est-à-dire le périmètre sur lequel la sécurité sera testée,
- une cible de sécurité (*Security Target*) qui détermine l'environnement dans lequel le produit évolue (e.g. fabrication, développement, etc.),
- un profil de protection (*Protection Profile*) qui décrit l'ensemble des exigences de sécurité obligatoires pour un type de produit donné (e.g. passeports électroniques, plates-formes Java CardTM).

Au travers de ces différents documents, les Critères Communs explicitent quels sont les critères que les évaluateurs doivent utiliser pour juger de la conformité des cibles d'évaluation face à leurs exigences de sécurité. Les Critères Communs décrivent les actions d'ordre général à mener sur des fonctions de sécurité sans toutefois décrire la façon dont les actions doivent être menées (i.e. la méthodologie et le savoir-faire qui est propre à chaque laboratoire).

Niveau d'assurance

L'évaluation est réalisée suivant un niveau d'assurance que souhaite atteindre un commanditaire. Autrement dit, suivant le type de produits évalués, il n'est pas nécessaire d'atteindre le même niveau de sécurité. Par exemple, un boîtier de chiffrement qui sera destiné à fonctionner dans une salle sécurisée n'est pas exposé aux mêmes menaces qu'une carte à puce et donc le niveau de sécurité exigé ne sera pas le même. Pour cette raison, les Critères Communs définissent sept niveaux d'assurance (*EAL : Evaluation Assurance Level*) allant de « testé fonctionnellement » à « conception vérifiée à l'aide de méthodes formelles et testée » (voir figure 9).

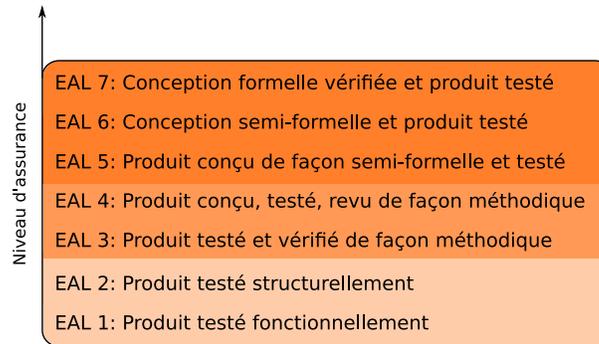


FIG. 9: Échelle d'assurance des Critères Communs.

L'EAL1 est le niveau le plus bas de l'échelle. Dans ce mode d'évaluation, les seuls documents que doit fournir le développeur aux évaluateurs concernent :

- la spécification fonctionnelle,
- la gestion de configuration,
- une campagne de tests,
- un guide d'installation, d'administration et d'utilisation.

Autrement dit, le travail de l'évaluateur consiste à vérifier si le produit fonctionne correctement et en aucun cas il ne lui est demandé de vérifier s'il est résistant face à certaines attaques. En revanche, à partir du niveau EAL5, il est nécessaire de mettre en œuvre des techniques formelles. Ainsi, le travail de l'évaluateur ne consiste plus à effectuer des tests fonctionnels mais bien à vérifier des modèles, des preuves formelles et surtout réaliser des attaques physiques et logicielles. Le niveau habituellement utilisé est *EAL4*, voire *EAL5* pour les applications embarquées sur les cartes à puce. À ce propos nous pouvons citer le projet FORMAVIE (voir [17]) qui a pour but d'effectuer une modélisation et une vérification formelle de l'architecture de Java Card™2.1.1 en vue d'obtenir une certification EAL7.

1.5.2 L'évaluation

Toute évaluation commence par l'examen des documents que sont : la cible d'évaluation, la cible de sécurité et le profil de protection. Une fois que ces documents ont été lus et que leur cohérence a été vérifiée, l'évaluateur doit s'assurer que :

- la cible de sécurité est complètement et correctement implémentée,
- le processus de développement est maîtrisé,
- la cible d'évaluation fait ce pour quoi elle a été conçue,
- le client déploie correctement la cible d'évaluation.

Pour obtenir et apporter les preuves de cette assurance, les évaluateurs doivent vérifier un certain nombre d'exigences décrites dans les Critères Communs en fonction de l'échelle d'assurance demandée par le commanditaire (voir section 1.5.1). Ceci passe par trois étapes qui sont : une analyse documentaire, des tests fonctionnels et une mise en place d'attaques.

Analyse documentaire

Pour mieux comprendre le produit à évaluer, la première étape consiste à analyser la documentation fournie par le développeur dont le contenu minimum est imposé par les Critères Communs. L'évaluateur s'assure alors de la cohérence, de la précision mais aussi de la complétude de ce contenu.

Tests fonctionnels

La campagne de tests intervient après que l'évaluateur se soit assuré de la cohérence de la documentation fournie. Les tests fonctionnels permettent de s'assurer que le produit fonctionne correctement mais surtout que le comportement sécuritaire est conforme à celui décrit dans la documentation. Ces tests sont divisés en deux parties qui sont le rejeu des tests fournis par les développeurs et les tests indépendants.

Le rejeu des tests implique de reproduire les scénarii présentés dans la documentation fournie par le développeur. L'évaluateur doit s'assurer que les résultats qu'il obtient sont les mêmes que ceux annoncés par le développeur.

La campagne de tests indépendants, pousse l'évaluateur à concevoir des tests qui ne seraient pas couverts par la campagne du développeur. L'évaluateur cherche à tester les cas limites, inventer des scénarii et tout cela afin de vérifier que le comportement du produit est correct et fait ce pour quoi il a été conçu.

Mise en place d'attaques

Après avoir mené sa campagne de tests, un évaluateur doit alors déterminer si le produit en cours d'évaluation ne présente pas de faiblesses intrinsèques et s'il est résistant à toute une gamme d'attaques.

Prenons le cas où une carte à puce est évaluée. La recherche de vulnérabilités débute par l'analyse du code source des procédures de sécurité (e.g. parties critiques du système d'exploitation, vérification de code PIN, algorithmes cryptographiques, etc.). Durant sa phase d'analyse du code source, l'évaluateur tente d'identifier des faiblesses qui, couplées à des attaques physiques (voir chapitre 3), permettraient de contourner ou d'altérer des mécanismes sécuritaires. Une fois ces vulnérabilités découvertes, l'évaluateur met en place les attaques qui permettent de valider ou d'invalider les faiblesses découvertes durant l'analyse de code source.

1.6 Conclusion

Dans ce chapitre, nous avons présenté l'essor des cartes à puce depuis leur invention jusqu'à ce qu'elles sont devenues de nos jours. Au travers de cette évolution nous souhaitons montrer comment les cartes à puce s'adaptent aux exigences du marché mais surtout à quels enjeux sécuritaires elles doivent faire face. Ainsi, nous avons vu que rendre les cartes plus sûres implique trois niveaux de sécurité constitués du corps de la carte, des composants matériels et des logiciels embarqués.

Cependant, il faut bien s'assurer que les cartes sont réellement plus sûres et qu'elles sont capables de résister à tout un panel de différentes attaques. Comme nous l'avons vu, cette tâche est déléguée à des laboratoires indépendants nommés CESTI dont le but est de mettre au jour des failles qui seraient présentes dans les composants matériels et logiciels.

C'est justement ce dernier point concernant l'évaluation de la sécurité des logiciels embarqués qui fait l'objet de nos travaux. En effet, cette thèse consiste à fournir des solutions permettant d'automatiser autant que possible le travail des évaluateurs en charge de la partie logicielle embarquée dans les cartes. Comme nous le verrons dans les chapitres suivants, nous avons mis au point des outils qui aident à la découverte de vulnérabilités présentes dans les codes sources de programmes écrits aussi bien en langage d'assemblage que dans des langages de haut niveau.

Chapitre 2

Les microcontrôleurs

Les microcontrôleurs font partie intégrante de notre vie quotidienne. Ils sont présents dans la plupart des appareils électriques qui nous entourent allant du lave-linge au dernier baladeur numérique en passant par des matériels de communication, des appareils médicaux et bien sûr les cartes à puce. Les microcontrôleurs permettent de piloter l'appareil dans lequel ils sont intégrés et ils prennent souvent en charge l'interface utilisateur. Ainsi, une voiture peut contenir jusqu'à cinquante microcontrôleurs gérant des sous-systèmes tels que la radio, le système GPS, l'injection de carburant ou encore l'ABS d'un système de freinage. Un avion peut en utiliser deux cents ou plus. Il n'est pas improbable que d'ici quelques années la quasi-totalité des appareils électriques et électroniques intègre un microcontrôleur.

Malgré leur taille réduite, les microcontrôleurs sont de véritables ordinateurs miniaturisés car ils sont constitués de circuits intégrés rassemblant un microprocesseur ainsi que d'autres composants nécessaires à leur bon fonctionnement tels que de la mémoire et des capacités d'entrée/sortie. Cependant, le fonctionnement d'un microcontrôleur diffère quelque peu de celui des ordinateurs classiques. En effet, dans la plupart des cas, un microcontrôleur reçoit des commandes au travers d'interfaces telles que des terminaux ou des interrupteurs afin de contrôler les paramètres d'un dispositif particulier comme le son ou l'éclairage. Mais en réalité, les microcontrôleurs sont bien plus que des interrupteurs améliorés, ce sont de véritables unités programmables classées en deux catégories dites généraliste ou spécifique. La première catégorie correspond aux microcontrôleurs qui sont de véritables ordinateurs miniaturisés tandis que la seconde correspond à ceux dont l'architecture est dédiée à certains traitements spécifiques comme par exemple le multimédia. Fréquemment, le logiciel est directement intégré au matériel de la puce sous la forme d'une mémoire en lecture seule créée au moment de sa fabrication.

Les cartes à puce, qui sont l'objet de cette thèse, ne sont rien d'autre que des cartes en plastiques intégrant un microcontrôleur. Ainsi, examiner la sécurité d'une carte à puce passe entre autre par l'étude du microcontrôleur qui la compose. L'objectif de ce chapitre est de donner au lecteur tous les éléments nécessaires concernant

l'architecture commune à deux familles de microcontrôleurs généralistes afin de comprendre les travaux qui vont être présentés par la suite.

2.1 Architecture d'un microcontrôleur

Les microcontrôleurs que nous étudions se nomment respectivement AT90S8515 [18] et ATmega128 [19]. Ce sont des microcontrôleurs généralistes de type RISC¹, commercialisés par le fondeur Atmel[®] et dont la taille des registres de travail est de huit bits. Dans ces microcontrôleurs, nous retrouvons les principaux composants d'un ordinateur classique à savoir de la mémoire, des bus et une unité arithmétique et logique. Cependant, la fréquence d'horloge et la capacité mémoire des microcontrôleurs sont bien inférieures à celle d'un ordinateur. En effet, là où la fréquence d'horloge d'un ordinateur classique s'exprime en gigahertz et sa capacité mémoire en gigaoctets, celles des microcontrôleurs qui nous intéressent s'expriment en mégahertz et en kilooctets. Les deux familles ont une architecture quasiment identique et la différence majeure réside dans le fait que la famille ATmega128 possède plus de mémoire et une fréquence d'horloge plus élevée que la famille AT90S8515. Le tableau 2.1 présente un comparatif entre ces deux composants en terme de mémoire, de nombre d'instructions et de fréquence d'horloge.

Caractéristiques	Architectures	
	AT90S8515	ATmega128
# Instructions	118	133
Flash	8 KOctets	128 KOctets
EEPROM	512 Octets	4 KOctets
SRAM	512 Octets	4 KOctets
Fréquence d'horloge	8 MHz	16 MHz

TAB. 2: Comparaison des architectures étudiées.

Les deux microcontrôleurs que nous présentons utilisent une architecture de type Harvard (voir figure 10) qui sépare physiquement les mémoires servant à stocker les instructions du programme et les données. Ce type d'architecture autorise seulement le microprocesseur à charger des instructions présentes dans l'espace de stockage du programme et à écrire dans l'espace de données. Ces deux opérations peuvent être réalisées en même temps créant ainsi des accès plus rapides et limitent les goulots d'étranglements contrairement à une architecture de type Von Neumann [20] qui est l'architecture des ordinateurs classiques (i.e. de type x86, SPARC). De plus, le compteur ordinal (*program counter* en anglais) ne peut adresser que l'espace de stockage du programme. Il en découle que l'espace mémoire réservé aux données ne peut être exécuté contrairement à une architecture de type Von Neumann.

¹Reduced Instructions Set Computer

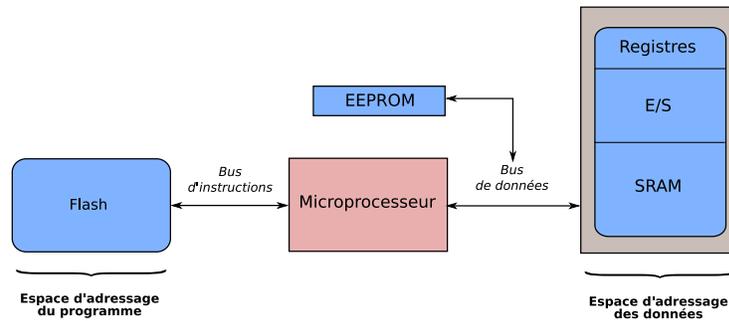


FIG. 10: Architecture de type Harvard.

Une véritable architecture Harvard interdit à des processus de venir modifier, en partie ou totalement, le programme stocké en espace non volatile. Cette approche permet de réduire un certain nombre d'attaques logicielles comme par exemple les attaques par débordement de tableau (e.g. les fameux *buffer overflows*²). Ainsi, avec une véritable architecture Harvard, une modification de l'espace de stockage du programme nécessiterait un accès physique à la mémoire. Comme ceci est impossible en pratique, les microcontrôleurs basés sur de véritables architectures Harvard sont rarement utilisés. Pour pallier à ce problème, la plupart des microcontrôleurs utilisent une architecture Harvard modifiée qui autorise la modification du programme dans certaines circonstances. L'utilisation de ces microcontrôleurs devient alors plus souple mais au détriment de la sécurité car certaines attaques par injection de code deviennent alors possibles (voir [21]).

Les deux familles de microcontrôleurs que nous étudions sont basées sur une architecture Harvard modifiée. En effet, elles possèdent des instructions³ qui permettent de charger des valeurs d'initialisation depuis l'espace de stockage du programme vers l'espace de stockage des données mais aussi de copier de grands tableaux de données dans l'espace de stockage du programme évitant ainsi de gaspiller la précieuse mémoire RAM. La figure 11 détaille de façon très précise l'architecture de ces deux familles de composants.

Dans ce qui suit nous allons détailler précisément les principales composantes des architectures que nous étudions.

2.1.1 Les registres

Un registre est un espace mémoire à l'intérieur du microprocesseur, dont la fonction consiste entre autres à contrôler l'exécution des programmes mais aussi à maintenir des informations temporaires. Les registres se divisent en deux grandes catégories qui sont les registres généraux et spécialisés. Les registres généraux servent

²Type d'attaque consistant à injecter du code sur la pile et à l'exécuter.

³Elles sont au nombre de deux et se nomment SPM (Store Program Memory) et LPM (Load Program Memory).

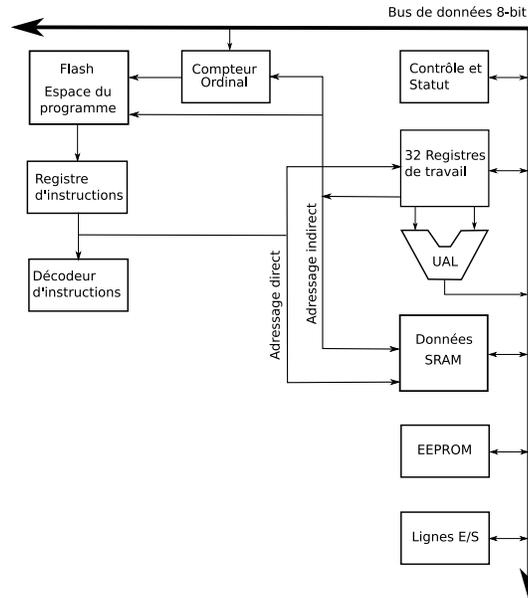


FIG. 11: Architecture détaillée des familles AT90S8515 et ATmega128.

à stocker les variables locales ainsi que les résultats de calculs intermédiaires. Cela permet d'éviter des accès à la mémoire en fournissant un accès rapide aux données fréquemment utilisées. Les architectures que nous considérons ici disposent de trente-deux registres généraux de huit bits. La seconde catégorie regroupe des registres particuliers ne servant pas à stocker le résultat de calculs mais qui sont néanmoins nécessaires au bon déroulement d'un programme. Parmi ces registres on retrouve :

- le compteur ordinal (*Program Counter*) qui contient l'adresse de la prochaine instruction qui doit être récupérée puis exécutée,
- le registre d'instruction (*Instruction Register*) qui est chargé de stocker l'instruction en cours d'exécution,
- le pointeur de pile (*Stack Pointer*) qui référence le sommet de pile sur laquelle se trouve les variables locales et les adresses de retour des fonctions,
- le registre d'état (*Status Register*) qui est un registre de contrôle dont chaque bit indique le statut du résultat de l'opération la plus récente.

Le registre d'état est un registre qui indique par exemple s'il y a eu un débordement lors d'une addition, si un nombre considéré est négatif, etc. Ces statuts d'opérations sont stockés sur chacun de ces bits dont le détail est donné ci-dessous :

- N (*Negative*) indique un résultat négatif,
- Z (*Zero*) indique un résultat nul,
- V (*oVerflow*) indique que le résultat a engendré un dépassement de capacité,
- C (*Carry*) indique que le résultat a conduit à une retenue sur le bit le plus à gauche,
- H (*Half carry*) indique que le résultat a conduit à une retenue intermédiaire

sur le bit 3,

- S (*Signed tests*) indique si la valeur considérée est signée ou non,
- T (*Transfer bit*) sert à sauvegarder la valeur d'un bit d'un registre,
- I (*global Interrupt enable*) doit être levé pour activer le mécanisme d'interruptions.

Les codes de condition sont principalement utilisés par les instructions de branchement conditionnel et de comparaison.

2.1.2 La mémoire

Les mémoires sont des composants très importants du microcontrôleur car elles servent à stocker les instructions du programme à exécuter ainsi que les données que ce dernier manipule. Comme l'illustre le schéma 10, un microcontrôleur Atmel[®] possède trois mémoires internes et une mémoire externe qui vont être détaillées ci-après.

La mémoire Flash

La mémoire Flash interne correspond à la portion de mémoire où sont stockées les instructions du programme. Le microprocesseur ne peut exécuter que le code se trouvant dans cette mémoire (i.e. le compteur ordinal ne peut adresser que cette partie de la mémoire). La mémoire Flash interne a une endurance de dix mille cycles d'écritures/effacements. De plus, pour des raisons de sécurité elle est divisée en deux sous-parties nommées respectivement application et *bootloader* (voir figure 12). La section application contient le programme à exécuter tandis que le *bootloader*

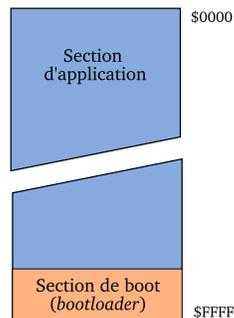


FIG. 12: Cartographie de la mémoire Flash.

correspond à une portion de code permettant de faire des opérations très particulières. Le *bootloader* permet par exemple de charger l'image d'un programme à l'intérieur du microcontrôleur. En effet, dans certains cas quand le code embarqué dans un microcontrôleur a besoin d'être changé ou mis à jour, c'est le code présent dans le *bootloader* qui va copier les nouvelles données en mémoire Flash externe dans la mémoire du programme. Ainsi, la mémoire Flash peut être programmée

soit par une connexion physique au microcontrôleur ou par auto-reprogrammation (*self-reprogramming*) via le *bootloader*.

L'espace de données

L'espace de données est adressable avec les instructions classiques et il est utilisé dans différents buts. Comme le montre le schéma 13, cet espace mémoire contient les registres généraux, les registres d'entrée/sortie, et la mémoire vive de type SRAM. La zone de mémoire nommée `.data` sert à stocker les données globales initialisées, c'est-à-dire les données dont les valeurs sont connues à la compilation. La zone `.BSS`, quant à elle sert à stocker les données globales non initialisées.

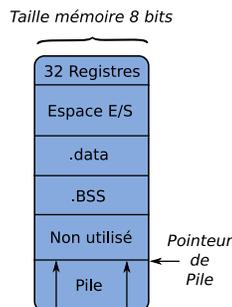


FIG. 13: Espace d'adressage des données.

Comme un microcontrôleur n'utilise pas d'unité de gestion de la mémoire (*Memory Management Unit*), aucune vérification d'adresse n'est effectuée avant l'accès à une case mémoire. Il en découle que tout l'espace de stockage des données est directement adressable.

La mémoire EEPROM

Cette mémoire possède son propre espace d'adressage et ne peut être accédée qu'en passant par des registres d'entrée/sortie dédiés. Comme cette mémoire est non volatile (i.e. qui ne s'efface pas lorsque le microcontrôleur est hors tension), elle est principalement utilisée pour stocker des données de configuration. De plus, son endurance est de cent mille cycles d'écritures/effacements.

Nous venons de voir comment les informations étaient stockées regardons à présent comment elles transitent entre les différentes parties du microcontrôleur.

2.1.3 Les bus

Un microprocesseur possède un ensemble de broches sur lesquelles transitent les communications avec l'environnement extérieur. Il existe trois types de broches qui sont les broches d'adresses, de données et de commandes. Elles sont reliées à des broches similaires de la mémoire et des circuits d'entrée/sortie grâce à un ensemble

de fils parallèles appelé bus. Un bus désigne l'ensemble des pistes de communication véhiculant les données numériques entre le microprocesseur, la mémoire vive et les divers périphériques. Toutes les informations transitant dans un microcontrôleur empruntent donc des bus qui se décomposent, comme les broches d'un microprocesseur, en trois grandes familles.

En premier lieu, il y a les bus d'adresses qui permettent au microcontrôleur de sélectionner la case mémoire ou le périphérique auquel il veut accéder pour lire ou écrire une information. Ensuite, viennent les bus de données qui permettent le transfert des informations entre les différents composants du microcontrôleur. Les informations présentes sur ce bus sont soit des instructions soit des données provenant de blocs mémoires ou de périphériques. Enfin, citons les bus de contrôle qui indiquent si l'opération en cours est une lecture, une écriture ou si un périphérique demande une interruption.

2.1.4 Fonctionnement

Nous venons d'avoir un aperçu des différents éléments qui composent un microcontrôleur. Regardons à présent comment ils interagissent entre eux pour exécuter un programme. Le fonctionnement d'un microcontrôleur peut se résumer en sept étapes :

1. Le microprocesseur récupère dans la mémoire l'instruction qui se trouve à l'adresse pointée par le compteur ordinal (initialisé à zéro lors de la mise sous tension du microcontrôleur) puis charge cette instruction dans le registre d'instruction.
2. La valeur du compteur ordinal est modifiée afin de le faire pointer vers l'adresse de la prochaine instruction à exécuter.
3. Le type de l'instruction venant d'être chargée est déterminé.
4. Si l'instruction utilise un mot de la mémoire, il faut localiser son emplacement.
5. S'il y a lieu, il faut charger le mot dans un registre du microprocesseur.
6. Exécution de l'instruction.
7. Retour à l'étape 1 pour effectuer l'opération suivante.

Cette séquence d'événements est souvent appelée cycle de chargement-décodage-exécution et nous la retrouvons dans le fonctionnement de tous les ordinateurs.

2.1.5 Jeu d'instructions

Les opérations que doit effectuer un microprocesseur sont déterminées par les instructions qu'il exécute. Dans cette partie nous allons d'abord présenter quelles sont les principales catégories d'instructions présentes sur les microcontrôleurs et dans un second temps nous aborderons le format d'une instruction.

Les types d'instructions

Le langage d'assemblage AVR ne comporte que des instructions ayant aux maximum deux opérandes dont la syntaxe est :

$$OP \ Rd, \ Rr$$

où OP désigne l'opération à effectuer et Rd (resp. Rr) le registre de destination (resp. source). Par exemple, pour additionner le contenu de deux registres on écrira : `ADD R1, R2` dont la sémantique est $R1 \leftarrow R1 + R2$. Le nombre d'instructions varie d'un microprocesseur à l'autre, cependant nous retrouvons toujours les cinq grandes catégories suivantes :

Les transferts de données. Ces instructions spécifient l'emplacement des opérandes sources et destinations, la taille des données à transférer, ainsi que la manière d'accéder à ces données. Les trois instructions principales concernent :

- le passage de la mémoire au registre (e.g. LD, LDD, LDS),
- le passage du registre à la mémoire (e.g. ST, STD),
- la copie d'un registre dans un autre (e.g. MOV).

Les opérations arithmétiques. Ces instructions servent à effectuer des opérations de base telles que :

- des additions (e.g. ADD, ADC, ADIW),
- des soustractions (e.g. SUB, SUBI),
- des multiplications (e.g. MUL, MULS, MULSU, FMUL),
- des divisions (e.g. LSR, ASR).

Les opérations logiques. Ce sont toutes les opérations ayant trait aux manipulations logiques de bits tel que :

- \vee (e.g. OR, ORI),
- \wedge (e.g. AND, ANDI),
- \oplus (e.g. EOR),
- \neg (e.g. NOT).

Les entrées/sorties. Ces instructions sont utilisées pour lire ou écrire sur des ports spécifiques permettant l'interfaçage avec des périphériques extérieurs (e.g. IN, OUT).

Les branchements conditionnels et les comparaisons. Les programmes doivent être capable de modifier leur flot de contrôle en fonction du résultat de certains tests (e.g. CP, CPI, CPC, etc.), ceci est possible grâce aux instructions de branchements (e.g. BREQ, BRLT, BRGE, etc.). Ce sont des instructions qui provoquent des sauts vers telle ou telle adresse mémoire selon que certains bits du

registre de statut sont positionnés ou non. C'est grâce à ce type d'instructions que l'on peut modéliser des boucles.

Les instructions d'appel de procédure. Une procédure (ou sous-programme) est une série d'instructions qui peut être invoquée à n'importe quel moment par un programme (e.g. `ICALL`, `CALL`). Une fois la fin du sous-programme atteinte, l'exécution du programme appelant reprend à l'instruction qui suit l'appel de la procédure.

Langage d'assemblage et instructions

Une instruction est une séquence de bits comportant plusieurs champs, où chacun d'eux joue un rôle bien spécifique. Le premier champ se nomme le code opération (*opcode* en anglais) et il permet d'identifier une instruction en indiquant s'il s'agit d'une addition, d'un branchement ou d'une autre opération. Les champs supplémentaires concernent les opérandes sources et l'emplacement de destination des résultats.

Pour faciliter la programmation, chaque instruction admet une représentation symbolique où le code opération est représenté par une abréviation appelée mnémotique. Considérons par exemple l'instruction qui permet d'additionner le contenu de deux registres et ayant pour code opération `ADD`. Cette instruction prend en paramètres deux registres nommés `Rd` pour le registre de destination et `Rr` pour le registre source. Le format de cette instruction est décrit sur le schéma 14 et la sémantique de l'opération effectuée est la suivante $Rd \leftarrow Rd + Rr$ où $0 \leq d \leq 31$ et $0 \leq r \leq 31$.

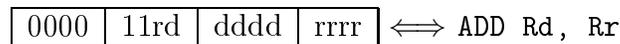


FIG. 14: Format d'une instruction 16 bits et sa mnémotique correspondante.

Les familles de microcontrôleurs que nous considérons n'ont pas toutes leurs instructions de la même longueur. Ainsi, la majorité des instructions sont codées sur 16 bits exceptées celles manipulant la mémoire : les adresses pour accéder à la mémoire étant codées sur 22 bits et l'opération à effectuer étant codée sur 10 bits il en résulte des instructions codées sur 32 bits.

2.2 Conclusion

Au travers de ce chapitre, nous avons pu présenter en détail le fonctionnement d'un microcontrôleur. D'une part, nous avons abordé la description des différents composants physiques présents dans un microcontrôleur, quels mécanismes les régissent et comment ils interagissent entre eux. Cette description permet de mieux aborder le chapitre 4 dans lequel nous décrivons un environnement logiciel qui reproduit le comportement d'un microcontrôleur Atmel[®]. Cette simulation du fonctionnement d'un microcontrôleur a pour but de tester la robustesse d'implémentations d'algorithmes cryptographiques face à des attaques basées sur l'analyse de la

consommation de courant.

D'autre part, nous avons présenté les spécificités et les particularités du langage d'assemblage qui permet de manipuler le microcontrôleur. Cette présentation pose les bases permettant d'aborder le chapitre 5 dans lequel nous décrivons un outil permettant d'analyser les programmes écrits en langage d'assemblage AVR. Cet outil a pour vocation d'aider à rechercher des attaques basées sur un déséquilibre des temps d'exécution des programmes embarqués dans des microcontrôleurs.

Le rapide descriptif que nous venons de donner sur le contenu des chapitres à venir évoquent le fait qu'il existe des attaques permettant de mettre à mal la sécurité des microcontrôleurs et des programmes qu'ils exécutent. En réalité, de part leur taille et leur utilisation dans des environnements non sécurisés, les microcontrôleurs constituent des cibles privilégiées pour des personnes mal intentionnées. Le chapitre suivant a pour vocation de donner une vue d'ensemble des différentes attaques qui visent à affaiblir la sécurité des microcontrôleurs.

Chapitre 3

Les attaques

Précédemment, nous avons présenté en détail l'architecture des microcontrôleurs et nous avons vu qu'ils étaient utilisés par un grand nombre d'appareils électroniques qui nous entourent. La carte à puce ne fait pas exception à la règle et intègre elle aussi un microcontrôleur. En revanche, contrairement aux autres appareils électroniques, elle est dotée d'un ensemble de mécanismes qui protègent physiquement le microcontrôleur contre des agressions extérieures. Pourtant, ce niveau de résistance n'est pas absolu et un attaquant ayant accès à un équipement de test pour semi-conducteurs peut arriver à extraire l'information secrète enfouie dans le microcontrôleur comme par exemple le code PIN ou des clefs de chiffrements, en manipulant ou en observant les composants électroniques de la puce. Dans son guide destiné aux architectes des systèmes de sécurité (voir [22]), IBM définit trois classes d'attaquants répartis suivants les moyens dont ils disposent et l'intelligence qu'ils peuvent déployer :

- Classe I (Les non-initiés rusés). Ces attaquants sont souvent très intelligents mais ont une connaissance très limitée du système. De plus, ils peuvent avoir accès à un équipement moyennement sophistiqué. Le plus souvent ces attaquants tirent parti de failles existantes dans le système plutôt que d'essayer d'en créer de nouvelles.
- Classe II (Les initiés bien informés). Cette classe d'attaquants possède un niveau technique d'enseignement et d'expérience substantiels. Ils ont différents degrés de compréhension des parties du système mais un accès potentiel à la plupart d'entre eux. Enfin, ils ont souvent des outils et des instruments très sophistiqués pour réaliser des analyses.
- Classe III (Les organisations possédant des fonds). Ces attaquants sont capables de rassembler des équipes de spécialistes avec des compétences complémentaires grâce à leurs grands moyens financiers. Ils sont capables d'analyser en profondeur le système, de créer des attaques sophistiquées et d'utiliser des outils d'analyse à la pointe de la technologie. Ils peuvent utiliser des personnes de Classe II comme membre de l'équipe d'attaquants.

Ainsi, le niveau de sécurité offert par une carte à puce (ou un système en général) peut être mesuré par le temps, les moyens techniques et la somme d'argent qu'un

attaquant devra mettre en œuvre pour réussir à l'attaquer. Dans ce chapitre, nous présentons un grand nombre de techniques qui rendent les périphériques cryptographiques ou les cartes à puce vulnérables à des attaquants de classe I à III.

3.1 Les attaques par modification permanente

La classe d'attaques que nous présentons ici est probablement une des plus puissantes de toutes celles qui existent. Ces attaques agissent physiquement sur les composants électroniques et partent du principe qu'il n'y a quasiment pas de limites à ce qui peut être fait pour révéler le secret enfoui dans le périphérique. Ces attaques débutent systématiquement par une ouverture physique du périphérique ciblé afin d'avoir un accès direct à ses composants électroniques (voir [23]). Par exemple, un attaquant voulant accéder au microcontrôleur d'une carte à puce fera appel à des procédés physico-chimiques (e.g. tel que l'utilisation d'acides) pour enlever le plastique et la colle sans abîmer les circuits électroniques. Les accès sont possibles soit par la face avant (i.e. côté dos de la carte) ou par la face arrière (i.e. côté puce de la carte). Notons tout de même que ces attaques peuvent avoir un effet destructeur car dans certains cas elles modifient de façon permanente la manière dont le périphérique fonctionne.

3.1.1 Le micro-sondage

Cette attaque, connue sous le nom de *microprobing* en anglais, consiste à venir poser des micro-sondes à des endroits très précis des circuits électroniques (e.g. les bus d'un microcontrôleur) dans le but de venir espionner (voire de modifier) les informations qui transitent dessus.

3.1.2 La modification de circuits

Le but est d'interférer physiquement avec le schéma de circuits électroniques afin d'en modifier le comportement et de lui faire réaliser des opérations spécifiques. Ces attaques nécessitent des équipements relativement coûteux tels que des lasers de découpe ou des canons à ions.

3.2 Les attaques par injection de fautes

Cette classe d'attaques (voir par exemple [24, 25]) consiste à perturber le fonctionnement d'un périphérique. Le but de la perturbation est de provoquer des fautes qui auront pour effet de changer la valeur d'une cellule mémoire ou de perturber le déroulement du programme en cours d'exécution. Ces attaques, particulièrement efficaces, requièrent néanmoins une grande expertise et beaucoup de temps. En effet, le processus de localisation de la bonne position pour une attaque sur les puces

modernes constitue une tâche ardue. Cette section a pour but de présenter les principales attaques connues en matière d'injection de fautes.

3.2.1 Injection par les paramètres fonctionnels d'une puce

Une première façon de provoquer des erreurs est de faire fonctionner le périphérique en dehors de la plage d'utilisation prévue par le constructeur en faisant varier différents paramètres :

1. Le voltage fournit au périphérique. Faire varier le voltage durant l'exécution d'un programme a pour effet de faire mal interpréter voire sauter des instructions au microprocesseur.
2. La fréquence de l'horloge externe. Ceci peut causer une mauvaise lecture des données ou le saut d'une instruction. (e.g. le périphérique commence à exécuter l'instruction $n + 1$ avant que le microprocesseur ait fini d'exécuter l'instruction n).
3. La température de fonctionnement. Cette attaque fait fonctionner le périphérique à des températures se trouvant en dehors des bornes définies par le constructeur provoquant ainsi des effets sur les cellules de la mémoire RAM. Par exemple, en modifiant la température du périphérique à une certaine valeur, les écritures peuvent fonctionner mais pas les lectures.

3.2.2 Injection par rayonnement

Un autre type d'attaques par injection consiste à provoquer des fautes en exposant la puce à un rayonnement afin de modifier le comportement des applications embarquées. Pour fonctionner, ces attaques nécessitent d'avoir un accès physique aux composants électroniques afin d'en exposer une partie précise à un rayonnement. Il existe trois principaux types de rayonnement permettant d'injecter des fautes dans des composants électroniques :

1. Les rayons X et les ions. Bien que ces attaques requièrent un matériel coûteux (e.g. canon à ions) elles présentent néanmoins l'avantage de ne pas avoir besoin de systématiquement pratiquer une ouverture dans la carte pour accéder la puce.
2. La lumière blanche. Le courant induit par les photons peut être utilisé pour provoquer des fautes si le circuit est exposé à une intense lumière pendant une brève période de temps (voir [26]).
3. Des émissions laser. L'effet produit est similaire à celui obtenu avec de la lumière blanche ; cependant une émission laser présente l'avantage d'être directionnelle ce qui permet des attaques plus fines. En effet, le laser autorise des attaques se concentrant sur certaines parties de la puce comme par exemple les différents types de mémoires (RAM, ROM) ou les registres.

3.2.3 Analyse différentielle de fautes

Comme nous venons de le voir, un périphérique (e.g. un microcontrôleur de carte à puce) peut être mis à mal via une température de fonctionnement élevée, un voltage non supporté, une élévation de la fréquence d'horloge ou encore des rayonnements. Face à ces agressions externes, le microcontrôleur peut produire des résultats incorrects dus à la modification des données traitées. Il existe dans le domaine de la cryptanalyse une attaque appelée analyse différentielle de fautes (*Differential Fault Analysis* voir [27, 28]) qui tire partie des résultats incorrects obtenus grâce à l'injection de fautes pour en déduire la clef de chiffrement utilisée.

3.3 Les attaques par canaux auxiliaires

Les attaques par canaux auxiliaires (*side channel attacks* en anglais) exploitent les interfaces accessibles du périphérique visé. Il s'ensuit que le périphérique (microcontrôleur, carte à puce, boîtier de chiffrement, etc.) n'est pas altéré de façon permanente et qu'aucune trace de l'attaque n'est visible. La plupart de ces attaques (voir [29, 30]) peuvent être menées avec du matériel à bas coût et constituent une menace sérieuse pour la sécurité des périphériques (cryptographiques ou non). L'idée sous-jacente de ces attaques est de déterminer le secret contenu dans un périphérique en mesurant son temps d'exécution, sa consommation de courant ou son champ électromagnétique.

3.3.1 Les attaques basées sur le temps d'exécution

Les implémentations des algorithmes de comparaisons ou de chiffrement effectuent souvent des opérations en temps non constant et ceci en raison de critères de performances. Si de telles opérations manipulent un bien secret (e.g. un code PIN ou une clef secrète), alors l'analyse des variations de leurs temps d'exécution (voir figure 15) couplée à une analyse statistique peut mener à retrouver le secret dans sa totalité. Cette idée porte le nom de *timing attack* et fut introduite de façon théorique en 1996 par KOCHER dans [31]. Très rapidement, cette attaque fut mise

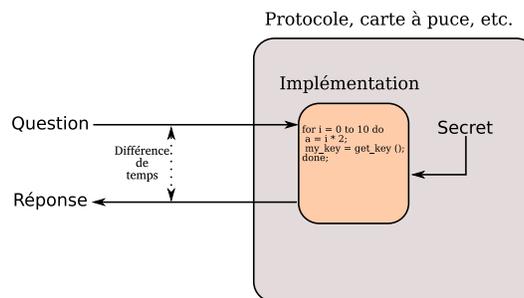


FIG. 15: Principe d'une *timing attack*.

en pratique sur des implémentations d'algorithmes cryptographiques. D'une part, ce type d'attaques a permis de déterminer le poids de Hamming de la clef secrète utilisée dans une implémentation de l'algorithme DES [32]. D'autre part, elle a permis de retrouver la clef de chiffrement utilisée dans une implémentation de l'algorithme RSA (voir [33]).

Plus récemment, un rapport technique (voir [34]) publié par BERNSTEIN démontre que l'implémentation de l'algorithme de chiffrement AES [6] utilisé dans la bibliothèque OPENSSL est vulnérable aux *timing attacks*.

3.3.2 Les attaques par analyse de la consommation de courant

Les attaques par analyse de la consommation de courant (*Power Analysis Attacks* en anglais) exploitent le fait que la consommation de courant instantanée d'un périphérique dépend des données manipulées et des opérations effectuées (voir [35]). Ainsi, en regardant la quantité de courant consommée par un périphérique, un attaquant peut déterminer ce qui se passe à l'intérieur et obtenir un certain nombre d'informations qui combinées avec d'autres techniques peuvent mener à retrouver le secret manipulé. En effet, la consommation de courant et le rayonnement électromagnétique induits par le déplacement de charges électriques dans les composants électroniques sont facilement mesurables.

Par exemple, pour mesurer la consommation de courant d'un circuit, une petite résistance est insérée en série sur la masse ou l'alimentation. Le voltage transitant dans cette résistance est mesuré puis enregistré grâce à un oscilloscope numérique. Nous noterons que ce voltage enregistré est directement proportionnel à la consommation de courant du périphérique. Ainsi, nous nous référons au voltage enregistré comme étant la consommation de courant et sa trace correspondante (i.e. la courbe affichée sur l'oscilloscope) comme la trace de consommation. Il existe trois principaux types d'attaques basés sur la consommation de courant que nous détaillons ci-après.

Analyse simple de la consommation

Une analyse simple de la consommation (abrégée en SPA pour *Simple Power Analysis* en anglais) se base sur l'observation de la représentation visuelle de la consommation de courant d'un périphérique afin d'en déduire de l'information sur le fonctionnement de l'algorithme en cours d'exécution ou sur les données secrètes que ce dernier manipule (voir [36]). La consommation totale de courant d'un périphérique dépend des instructions qu'il exécute mais aussi des données qu'il manipule durant l'exécution des différentes parties d'un algorithme. Cette propriété vient du fait que des instructions différentes, comme par exemple des opérations arithmétiques et des manipulations de la mémoire, ne consomment pas la même quantité de courant car elles n'utilisent pas exactement les mêmes circuits du périphérique. En

inspectant visuellement les traces de consommation, il devient alors possible de déterminer les caractéristiques majeures du périphérique cryptographique utilisé mais aussi d'en déduire de l'information sur l'implémentation de l'algorithme exécuté. Par exemple, les premières implémentations de l'algorithme de chiffrement RSA (voir [37]) embarquées dans des cartes à puce ont été cassées grâce à la SPA. En effet, des différences très nettes entre des opérations de multiplications et d'élevations au carré ont permis de retrouver la clef privée.

Une attaque par SPA peut s'avérer très puissante mais nécessite de bien connaître l'algorithme visé afin de pouvoir exploiter les courbes de consommation.

Analyse différentielle de la consommation

Alors qu'une SPA repose sur une inspection visuelle des courbes de consommation afin d'identifier des fluctuations significatives de courant, une attaque par analyse différentielle de la consommation (abrégée en DPA pour *Differential Power Analysis* en anglais) exploite des caractéristiques de comportement du périphérique comme par exemple le courant consommé par des transistors ou des portes logiques.

La DPA fut introduite par PAUL KOCHER (voir [38, 39]) et formalisée par THOMAS MESSERGES et al. (voir [40]). Elle utilise un modèle d'attaque couplé à une analyse statistique pour extraire de l'information enfouie dans un grand nombre de traces de consommation (voir [41]). Comme la DPA fait l'objet de résultats présentés dans la section 4.9, nous détaillons très précisément le principe de cette attaque.

Pour réaliser une DPA, un attaquant doit en premier lieu collecter un grand nombre N de traces de consommation différentes. Ces N traces correspondent au calcul d'un chiffré obtenu avec une même clef de chiffrement mais avec des messages clairs différents. Chaque trace de consommation est une collection d'échantillons de consommation $PS(n, t)$ qui représentent la consommation de courant dans la trace n au temps t (i.e. la somme du courant dissipé au temps t par tous les circuits présents sur le périphérique). L'étape suivante consiste à partitionner les échantillons de consommation $PS(n, t)$ en deux ensembles S_0 et S_1 suivant le résultat d'une fonction de discrimination D (aussi appelée fonction de sélection dans la littérature). Une fois le partitionnement effectué, l'attaquant doit calculer la courbe de consommation moyenne pour chacun des ensembles S_0 et S_1 au temps t . En soustrayant les courbes moyennes obtenues précédemment, nous obtenons un biais $B(t)$ entre les deux signaux. La fonction D cible un bit particulier du secret à attaquer et elle est choisie de telle sorte qu'à un moment donné durant l'exécution d'un algorithme cryptographique, le périphérique aura besoin de calculer la valeur de ce bit. À chaque fois que cela se produit, il y aura une légère différence dans la quantité de courant consommé suivant que la valeur du bit est positionné à zéro ou à un. Notons ϵ cette différence et supposons que l'instruction manipulant le bit ciblé par D apparaisse au temps t' , alors la valeur ϵ est égale à la différence suivante :

$$\epsilon = E[S|(D = 0)] - E[S|(D = 1)] \text{ pour } t = t'$$

Lorsque la fonction D adéquate a été choisie, la fonction de biais $B(t)$ fera apparaître des pics de consommation de hauteur ϵ au temps t' chaque fois que le bit ciblé par la fonction D est manipulé et sinon elle apparaîtra plate le reste du temps. L'apparition de pics de consommation permet donc à un attaquant de vérifier ses hypothèses concernant la valeur d'un bit de la clef secrète (i.e. intuitivement la présence de grands pics signifie que la valeur du bit attaqué a bien été retrouvée par la DPA). En répétant un nombre suffisant de fois ce procédé, un attaquant peut retrouver tous les bits de la clef secrète manipulée.

Une attaque de type DPA implique de posséder des centaines voire des milliers de courbes de consommation. Cependant, après le traitement des courbes et l'analyse statistique, seulement quelques minutes sont nécessaires au processus de DPA pour reconstruire entièrement la clef privée. Le procédé est facile à implémenter et ne nécessite qu'un équipement standard de mesure. Les attaques de type DPA n'agissent pas directement sur la puce et de ce fait ne perturbent pas le fonctionnement du périphérique attaqué. Cette particularité de l'attaque la rend particulièrement difficile à détecter. Nous noterons que le cœur d'une attaque par DPA repose entièrement sur la fonction D et elle doit être adaptée à l'algorithme visé. Finalement, la DPA ne nécessite que de très peu voire d'aucune information sur le périphérique visé et a permis d'attaquer avec succès un grand nombre de cartes à puce (voir [40]).

High-Order Differential Power Analysis

L'attaque High-Order DPA (HO-DPA voir [42, 43]) implique d'inspecter les consommations de courant entre les diverses sous-opérations d'une opération de chiffrement. Ainsi, alors que la DPA tente de trouver une corrélation entre un événement et plusieurs échantillons, la High-Order DPA peut être utilisée pour corréler de l'information entre plusieurs sous-opérations cryptographiques. Cette attaque devient alors beaucoup plus puissante que la DPA classique mais reste très coûteuse en terme de temps et d'espace mémoire car l'attaquant doit essayer toutes les corrélations possibles entre les différentes sous-opérations cryptographiques.

3.3.3 Les attaques par analyse des émissions électromagnétiques

Le dernier type de canal auxiliaire que nous présentons concernent les émissions électromagnétiques émises par le périphérique attaqué (voir par exemple [44, 45]). Le déplacement de charges électriques dans chaque composant électronique produit un champ électromagnétique qui peut être mesuré avec des sondes particulières. Suivant la taille et la position de la sonde, il est possible de mesurer le champ magnétique d'une zone particulière du périphérique visé, fournissant ainsi de l'information sur sa consommation de courant. L'attaquant devient donc plus précis et peut positionner sa sonde de façon stratégique comme par exemple juste au-dessus du coprocesseur cryptographique.

L'analyse d'émissions électromagnétiques mène à des attaques qui sont similaires à celles menées sur la consommation de courant. Ainsi, nous retrouvons l'équivalent de la SPA qui se nomme SEMA (*Simple EM Analysis*) et de la DPA qui devient la DEMA (*Differential EM Analysis*). Cette nouvelle classe d'attaques se basant sur les émissions électromagnétiques présente en plus l'avantage d'être insensible à un certain nombre de contre-mesures telles que le lissage de la consommation globale de courant ou l'ajout de bruit en sortie.

3.4 Les attaques par mauvaise utilisation

Il existe une classe d'attaques qui consiste à chercher des cas d'utilisations particuliers du périphérique non prévus par le programmeur ou qui n'ont pas été enlevés lors du lancement en production du produit (e.g. mode de débogage). Ces cas d'utilisations peuvent mettre le périphérique dans un état particulier ou donner l'accès à des fonctionnalités normalement non disponibles ou inaccessibles. Prenons par exemple le cas d'une carte à puce qui embarque une application bancaire. L'application bancaire a été implémentée conformément à une norme où chaque fonctionnalité est accessible via un *APDU* (voir section 1.2.4). Cependant, les programmeurs développant des applications bancaires doivent tester leurs applications et pour ce faire, il peuvent très bien développer des fonctions de tests réservées à leur propre usage. Imaginons que ces fonctionnalités de tests ne soient pas enlevées lors de l'embarquement du programme dans la puce, alors cela fait autant de nouveaux chemins d'attaques possibles. Un attaquant va donc tester tous les *APDU* possibles et observer le comportement de la carte à la recherche de fonctions non documentées permettant éventuellement une élévation de privilèges.

3.5 L'arrachage

Une carte à puce, contrairement aux autres systèmes informatiques, n'est alimentée en courant électrique qu'une fois insérée dans un lecteur. Cependant, il est tout à fait possible de retirer la carte du lecteur à n'importe quel moment provoquant sa mise hors tension et ayant pour effet d'arrêter brutalement toutes les opérations qu'elle était en train d'effectuer. Cette attaque se nomme un arrachage ou *tearing* en anglais.

Une attaque par arrachage vise à perturber le contenu de la mémoire car en cas d'interruption les données présentes en mémoire volatile sont perdues. Si certaines données présentes en mémoire RAM devaient être écrites en mémoire non volatile, il se peut que lors du redémarrage de la carte, le programme embarqué se trouve dans un état non prévu. En effet, certaines opérations doivent être effectuées de façon atomique, c'est-à-dire que soit toutes les instructions de l'opération sont complètement exécutées, soit aucune ne l'est. Quand seulement une partie des instructions est exécutée, le système se trouve alors dans un état incohérent. Prenons par exemple le

cas d'un porte-monnaie électronique dont le solde est stocké dans la mémoire de la carte. Imaginons que lors d'une opération de débit un attaquant retire subitement la carte du lecteur avant que le programme embarqué dans la carte ait eu le temps d'écrire le nouveau solde dans la mémoire. Si le programme est mal conçu, il se peut que le paiement soit accepté sans que le solde du porte-monnaie ne soit décrémenté.

Les programmes embarqués sur une carte à puce doivent donc tenir compte de la possibilité d'être interrompus à n'importe quel moment de leur exécution et mettre en place des mécanismes afin de rendre possible la récupération des données et de maintenir leur cohérence après une interruption.

3.6 Contre-mesures

Précédemment dans ce chapitre, nous avons présenté les principales attaques mises en œuvre pour révéler les données sensibles contenues dans un périphérique. Bien évidemment, face à ces attaques, des contre-mesures ont vu le jour essayant d'améliorer la sécurité tant au niveau matériel qu'au niveau logiciel. Dans ce qui suit, nous donnons un aperçu des contre-mesures qui sont généralement mises en place face aux attaques par micro-sondage, injection de fautes et canaux auxiliaires.

3.6.1 Micro-sondage

Nous avons déjà mentionné le fait que ces attaques étaient probablement les plus puissantes qui puissent être menées sur un périphérique. Face à de telles attaques envahissantes, les circuits électroniques doivent à tout prix être protégés de différentes façons. Les protections usuelles consistent à :

- cacher les bus en utilisant des glus logiques (i.e mélanger les lignes des bus) ou en les mettant sur les couches les plus basses de la puce. Un attaquant est alors obligé de faire du *reverse engineering* optique pour retrouver la bonne correspondance,
- chiffrer les bus,
- poser des capteurs qui détectent tout accès à la surface de la puce (e.g. détection de court-circuits).

Enfin, la technologie de gravure utilisée (0.5μ ou moins) tend à réduire la taille des circuits, ce qui rend la pose de micro-sondes encore plus difficile.

3.6.2 Injection de fautes

Les attaques par injection de fautes sont particulièrement puissantes dès qu'une méthode d'injection provoquant des effets connus à l'intérieur de la carte a été découverte. Afin d'assurer l'intégrité des données qu'elles contiennent, les cartes embarquent diverses protections qui sont présentes aussi bien au niveau matériel qu'au niveau logiciel.

Au niveau matériel

Un moyen de défense très efficace face aux attaques par injection de fautes consiste à intégrer des capteurs de haute qualité qui détectent et préviennent les effets non désirés produits dans la puce. Il existe différents types de capteurs à savoir :

- des capteurs de tension interne qui détectent des variations du voltage (i.e. sous-tension ou surtension),
- des capteurs mesurant les variations de la fréquence d'horloge qui empêchent de baisser ou d'augmenter la fréquence,
- des capteurs de températures qui préviennent tout fonctionnement de la puce en dehors des plages définies par le constructeur,
- des capteurs de lumière qui permettent d'éviter les attaques par émission laser.

Bien que ces capteurs détectent beaucoup de perturbations, il est souvent considéré comme prudent d'inclure des contre-mesures logicielles pour prévenir de nouvelles méthodes d'injection de fautes qui seraient passées au travers de la couche matérielle.

Au niveau logiciel

Au niveau logiciel, les contre-mesures consistent à protéger l'intégrité des données et détecter tout comportement anormal dans le déroulement du programme. Nous présentons trois contre-mesures classiques qui peuvent être utilisées conjointement.

Somme de contrôle. Les données sensibles sont associées avec une somme de contrôle (*checksum* en anglais) suffisamment complexe pour que la probabilité qu'une faute non détectée se produise soit négligeable. La somme de contrôle peut être vérifiée à chaque fois qu'une donnée est accédée en mémoire (i.e. lue ou enregistrée).

Redondance de données. Un certain degré de redondance est requis dans le code de telle sorte que n'importe quelle faute qui se produit sera détectée ou n'aura aucun effet sur le programme. Ceci implique par exemple d'avoir deux variables pour représenter les valeurs importantes en mémoire et d'exécuter des algorithmes permettant de tester leur cohérence.

Compteur de séquence. Les deux contre-mesures précédentes ont pour vocation de protéger les données ; mais il existe un moyen de protéger le flot d'exécution du programme (voir la thèse de M.-L. Akkar [46]).

L'idée générale consiste à positionner une variable (aussi appelée compteur de séquence) le long du flot de contrôle d'un programme. Durant l'exécution d'un programme, à chaque fois que le flot de contrôle rencontre cette variable, il incrémente sa valeur. Ponctuellement, des procédures vérifient la valeur courante de cette variable avec sa valeur attendue afin de détecter que certaines portions du programme n'ont pas été contournées. Par exemple, si sur un chemin d'exécution le compteur de

séquence est incrémenté quatre fois alors qu'il aurait dû l'être cinq fois, cela signifie qu'une perturbation physique a occasionné un saut d'instructions.

3.6.3 *Timing attacks*

Une contre-mesure manifeste pour éviter les analyses des temps d'exécution est de s'assurer que les opérations se déroulent en temps fixe. Cependant, suivant la nature des algorithmes, il n'est pas toujours possible de créer des implémentations où le temps d'exécution est constant indépendamment des données manipulées. Une alternative consiste à créer une implémentation dans laquelle le temps est soit indépendant de l'information secrète manipulée (e.g. clefs de chiffrement) soit seulement dépendant d'une quantité limitée d'information sur le secret et la divulgation de cette information limitée est sûre. Ces deux approches sont détaillées ci-après.

Temps constant

Pour s'assurer qu'une implémentation s'exécute en temps constant quelles que soient les données manipulées, un développeur doit calculer précisément le nombre de cycles d'horloge associé à une exécution où les données à comparer sont identiques puis différentes.

Si les nombres de cycles obtenus sont différents, il doit alors rééquilibrer les temps d'exécution soit en supprimant les branchements conditionnels (mais ce n'est pas toujours possible) soit en rajoutant des opérations fictives qui n'influent pas sur le contrôle de flot du programme (e.g. instruction `NOP`).

« Aveuglement »

La défense la plus communément acceptée pour protéger l'algorithme RSA contre des *timing attacks* est de réaliser une opération dite d'aveuglement (aussi appelée *blinding* voir [47]). L'idée générale est d'éviter qu'un attaquant connaisse l'entrée de la procédure d'exponentiation modulaire en mélangeant cette entrée avec un nombre aléatoire. Cette opération calcule $x = r^e g \bmod N$ juste avant le déchiffrement, où r est un nombre aléatoire, e est l'exposant public et g correspond aux données à déchiffrer. x est déchiffré normalement suivi de la division par r (i.e. $x^e / r \bmod N$). Comme r est aléatoire, x devient aléatoire et son temps de déchiffrement ne devrait pas révéler d'information sur la clef. Pour que cette contre-mesure fonctionne parfaitement, il faut générer un nombre r différent à chaque déchiffrement ; ce qui pénalise les performances.

3.6.4 Analyse du courant et des émissions électromagnétiques

Les canaux auxiliaires ayant trait au courant consommé ou aux émissions électromagnétiques comportent tellement d'informations que de multiples contre-mesures non triviales sont nécessaires pour les bloquer. Si les fuites spécifiques à chaque canal

(i.e. courant et électromagnétisme) peuvent être différentes, les attaques exploitant ces canaux partagent beaucoup de caractéristiques communes d'une part en ce qui concerne les techniques d'analyses employées et d'autre part en terme d'exploitation des fuites d'information. En effet, ces deux canaux de fuites présentent la propriété qu'à un instant donné ils fournissent de l'information sur l'activité interne du périphérique attaqué. Ainsi, les contre-mesures visant à bloquer les attaques par analyse de courant ou d'émissions électromagnétiques sont souvent similaires ou poursuivent des buts similaires. Dans ce qui suit, nous présentons des contre-mesures aussi bien matérielles que logicielles qui tendent à bloquer ce type d'attaques.

Contre-mesure matérielle

Les contre-mesures matérielles tentent de dégrader l'information disponible sur le canal auxiliaire soit en essayant de réduire les fuites soit en tentant d'augmenter le bruit. De plus, ces contre-mesures essaient aussi d'introduire un élément non prédictible (i.e. de l'aléa) dans les opérations qui sont effectuées.

Les étapes pour réduire les fuites impliquent de repenser les circuits pour éviter que les instructions ne laissent transpirer trop d'information sur les opérandes qu'elles manipulent. Cela passe par l'utilisation de boucliers (*shields*) qui isolent les effets de couplage entre les composants et atténuent les signaux compromettants. Les étapes qui visent à augmenter le bruit impliquent d'implémenter les algorithmes cryptographiques dans la couche matérielle de telle sorte qu'un grand nombre de processus puissent être effectués en parallèle : l'ajout de bruit actif par un générateur et la capacité d'occuper ou de désactiver d'autres composants pendant qu'une opération sensible est exécutée. Plusieurs contre-mesures matérielles impliquent aussi de la randomisation, ce qui réduit l'efficacité des attaques par SPA ou DPA. Elles introduisent des variations aléatoires du signal d'horloge (*random jitter*) ou bien encore randomisent le nombre de cycles pris par une instruction dans le but de rendre l'alignement de signaux difficile.

Par exemple, une des méthodes très répandue pour contrer une attaque par DPA consiste à introduire des processus d'interruptions aléatoires (*random process interrupts* ou RPIs). Au lieu d'exécuter toutes les opérations d'un programme séquentiellement, le microprocesseur entrelace l'exécution du programme avec des instructions fictives de telle sorte que les cycles des opérations ne correspondent pas à cause des décalages de temps. Ces décalages de temps provoquent des effets de désynchronisation (ils peuvent être considérés comme de l'ajout de bruit) et ont pour effet de ne pas faire apparaître des pics de consommation sur les traces.

Enfin, une autre contre-mesure prometteuse (mais coûteuse) provient de la technologie appelée *precharged dual rail logic* où chaque bit est représenté par deux circuits (voir [48]). À un moment donné, un 0 logique est physiquement représenté par un 01, et un 1 logique par 10. La transition vers la prochaine unité de temps passe soit par l'état physique 00 ou 11 de telle sorte que le même nombre de bits

change d'état indépendamment de l'état ultérieur. Ainsi, si les deux rails sont parfaitement équilibrés, la consommation du microprocesseur ne dépend plus des données qui sont manipulées.

Bien que ces contre-mesures matérielles soient très utiles pour réduire le rapport signal-bruit, elles ne sont pas toujours suffisantes pour prévenir des attaques et en pratique des contre-mesures logicielles sont ajoutées.

Contre-mesures logicielles

Les contre-mesures logicielles renferment une grande variété de techniques dont les principales impliquent des contre-mesures temporelles et le masquage de données.

Ajouter des contre-mesures temporelles signifie utiliser des astuces de programmation empiriques dans le but d'adapter le temps d'exécution d'un processus. Une instruction critique peut avoir son exécution randomisée de façon logicielle : si elle n'apparaît jamais au même moment, les analyses statistiques deviennent plus difficiles. À l'inverse, d'autres situations nécessitent que le programme s'exécute en temps constant dans le but de le protéger des attaques par *timing attacks* ou SPA. Par exemple, un branchement conditionnel peut être compensé avec une portion de code fictive dont la durée est similaire et sa courbe de consommation électrique quasiment identique.

Une contre-mesure efficace consiste à masquer les données (voir par exemple [49, 50]). Le but est double : éviter que les données soient manipulées en clair mais aussi rendre inactif toute prédiction concernant leur comportement observé via un canal auxiliaire. Par exemple, la technique d'aveuglement décrite plus haut (voir 3.6.3) peut aussi être appliquée pour bloquer des attaques qui analysent la consommation électrique. Dans ce qui suit nous présentons une contre-mesure introduite par Louis Goubin et Jacques Patarin (voir [51]) qui consiste à dupliquer une donnée.

Duplication. La duplication (d'ordre n) d'une variable V à l'aide d'une fonction f , consiste à remplacer V par un ensemble de n variables V_1, V_2, \dots, V_n en respectant les conditions suivantes :

1. Il est possible de retrouver V à partir des n variables V_1, V_2, \dots, V_n .
2. La connaissance d'un sous-ensemble strict des V_1, V_2, \dots, V_n ne donne aucune connaissance de V .
3. La fonction f est telle qu'il est possible de calculer les variables intermédiaires de l'algorithme correspondant à V_1, V_2, \dots, V_n au cours du calcul sans revenir à la valeur V .

Nos travaux nous ont amenés à tester des implémentations de l'algorithme de chiffrement DES afin d'étudier leur robustesse face à une attaque par DPA (voir chapitre 4). Pour cette raison, nous détaillons ici comment la technique de duplication peut être appliquée pour protéger cet algorithme. Le DES peut être protégé

par une 2-duplication où la fonction f sera le ou-exclusif. Une variable X sera dupliquée de la manière suivante : $X_1 \oplus X_2 = X$. Cette décomposition respecte bien la condition 1 et il est aisé de voir que la connaissance de X_1 ou de X_2 ne dévoile aucune information sur la valeur de X , ce qui satisfait la condition 2. La condition 3 est plus délicate à satisfaire.

Le DES est composé des sous-fonctions suivantes : permutations de bits, ou-exclusifs et boîtes-S. La manipulation des variables dupliquées pour les opérations de permutations et de ou-exclusifs ne pose aucun problème. Le point délicat concerne les boîtes-S qui ne sont pas linéaires, l'astuce consiste donc à recourir à deux séries de huit nouvelles boîtes qui auront douze bits en entrée et quatre bits en sortie. Elles seront définies de la manière suivante à l'aide d'une transformation secrète A de douze bits vers quatre bits :

$$(X'_1, X'_2) = S'(X_1, X_2) = (A(X_1, X_2), S(X_1 \oplus X_2) \oplus A(X_1, X_2))$$

Si la fonction A est choisie aléatoirement et demeure secrète, la condition 3 est alors vérifiée et il devient alors possible de construire l'algorithme DES en entier en utilisant une 2-duplication par ou-exclusif.

3.7 Conclusion

Les cartes à puce sont des périphériques destinés à protéger les secrets qu'elles contiennent (e.g. code PIN, clés de chiffrement, etc.) et ceci même lorsqu'elles se trouvent dans des environnements qui ne sont pas de confiance. Cette particularité fait que les cartes à puce sont la cible privilégiée d'un grand nombre d'attaques dont le but reste invariablement le même, à savoir récupérer le secret qu'elles contiennent.

Ces attaques se décomposent en deux grandes familles : les attaques actives et les attaques passives. La première famille concerne les attaques qui d'une manière ou d'une autre vont perturber le fonctionnement du microcontrôleur. De part leur nature agressive, ces attaques peuvent généralement être sabotées par des composants physiques qui sont directement intégrés dans le microcontrôleur. Quant à la seconde famille, elle regroupe les attaques qui n'agissent pas directement sur la puce, ce qui les empêche d'être détectées. Au cours de ce chapitre, nous avons tout particulièrement détaillé une attaque passive à savoir l'attaque par analyse différentielle de consommation (*DPA*) car elle fait l'objet de travaux dans le chapitre suivant.

Chapitre 4

Simulateur d'un microcontrôleur de carte à puce

Depuis la nuit des temps, la cryptographie s'est occupée d'assurer la confidentialité des communications. En effet, dès l'antiquité elle est utilisée par des politiciens, des diplomates et autres généraux afin de transmettre en toute sécurité des messages secrets à leurs interlocuteurs. Plusieurs siècles sont nécessaires pour que l'usage de la science des messages secrets se banalise et il faut attendre l'essor des moyens de communication modernes pour que l'apogée de la cryptographie soit atteinte. En effet, des mécanismes cryptographiques sophistiqués sont présents dans les objets électroniques de la vie quotidienne tels que le téléphone cellulaire, les technologies de la télévision à péage, les paiements électroniques développés par l'Internet et les cartes bancaires. La sécurité des algorithmes cryptographiques des premières heures reposait entièrement sur le secret du mode de chiffrement. Néanmoins, au fil des ans, la théorie de la cryptographie s'est étoffée et tend à montrer que les algorithmes utilisés devraient être publics. D'ailleurs, dans les années 1970, les gouvernements russe et américain ont publiés leurs propres standards de chiffrement appelés respectivement GOST (voir [52]) et DES (*Data Encryption Standard* voir [53]). La cryptologie secrète cède peu à peu la place à la cryptologie publique où les algorithmes sont connus, reposant entièrement sur une sécurité calculatoire¹. Autrement dit, même avec une puissance calculatoire importante, en règle générale, une recherche exhaustive de clef de chiffrement prendrait plusieurs dizaines d'années voire plusieurs siècles². Les algorithmes de chiffrement étaient devenus très robustes théoriquement et paraissaient donc incassables en pratique. C'était sans compter sur l'ingéniosité de certaines personnes qui ne s'attaquèrent pas à l'algorithme lui-même mais à la façon dont il était utilisé ou implémenté. C'est en 1996 qu'a commencé à

¹Cependant, certaines sociétés comme Airbus ou la RATP continuent à utiliser des algorithmes de chiffrement propriétaires.

²Néanmoins pour l'algorithme DES, il existe un super ordinateur à base de FPGAs qui est capable de retrouver la clef en moins de neuf jours (voir [54]).

apparaître un nouveau type d'attaques physiques appelés attaques par canal auxiliaire (*side channel attacks*). Ces attaques non intrusives consistent à observer les effets physiques liés à des calculs tels que le temps d'exécution, le rayonnement électromagnétique ou encore la consommation de courant pour retrouver les données secrètes manipulées par un algorithme lors de son exécution (voir section 3.3). En raison de leur caractère portable, les cartes à puce sont particulièrement exposées à ce nouveau type d'attaques.

4.1 Contexte

À l'heure actuelle, il y a un manque d'outils qui permettraient de démontrer rapidement qu'une implémentation résiste à une attaque par canal auxiliaire. Mettre en place une DPA est relativement simple et ne nécessite que peu de matériel : principalement un oscilloscope numérique et un ordinateur. Cependant, au-delà de la mise en place, c'est une expertise réelle qui nécessite un large spectre de compétences allant de l'électronique aux statistiques en passant par la cryptographie et l'informatique ; ce qui fait que la DPA reste très difficile à intégrer dans un processus de développement.

Dans l'état actuel des choses même si un développeur possède toutes les compétences requises pour être capable d'exploiter une DPA, voilà quelle est sa démarche. Tout d'abord, il doit posséder le matériel nécessaire pour charger le code qu'il a développé sur son ordinateur dans un microcontrôleur, ce qui en pratique est loin d'être le cas. Ensuite, après avoir collecté un grand nombre de traces de consommation, il doit déterminer la fonction de sélection adéquate puis tenter d'exploiter au mieux les courbes pour réaliser une DPA. Dans le cas où l'attaque réussit, il faut que le développeur repère dans son programme l'endroit qui fuit, ce qui implique d'inspecter toutes les instructions qui manipulent des données sensibles puis d'ajouter la contre-mesure adéquate. Enfin, à chaque contre-mesure ajoutée dans le code, il faut recommencer le processus pour vérifier si elle est efficace ou non. Par conséquent, la mise en place d'une attaque de type DPA par un développeur même chevronné reste difficile, laborieuse et consommatrice de temps.

4.2 Approche retenue

Notre approche consiste à fournir un environnement logiciel dédié à l'analyse de la résistance d'algorithmes cryptographiques contre les attaques par canaux auxiliaires et plus particulièrement contre la DPA. Nous avons décidé de nous focaliser sur l'analyse du binaire prêt à être chargé puis exécuté par un microcontrôleur de carte à puce et ceci plusieurs raisons.

En premier lieu, s'il est vrai que l'analyse du code source (e.g. langage d'assemblage, langage C) peut permettre d'identifier certaines contre-mesures qui sont connues pour être robustes face à des attaques par canaux auxiliaires (nous pensons

par exemple au masquage des clefs de chiffrement), rien ne garantit qu'elles aient été correctement implémentées. Il n'est pas impossible que les contre-mesures présentent des bogues subtiles qui les rendraient inefficaces et qui ne pourraient être détectés simplement en analysant le code source.

Ensuite, certaines opérations de bas niveau comme les calculs d'index de tableaux ou le temps précis de l'exécution d'une procédure sont abstraits dans le code source. Par conséquent, s'il y a des fuites d'information, elles ne pourront pas être détectées à ce niveau.

Enfin, le développeur n'a que très peu de contrôle sur le code produit par le compilateur. En particulier, un compilateur amené à optimiser du code, même de façon très simple, peut être amené à supprimer des contre-mesures.

Notre méthode consiste donc à analyser les traces d'exécution du binaire instruction par instruction. Pour cela, nous avons développé un simulateur de microcontrôleur qui nous permet d'analyser précisément les différents états du microprocesseur. En appliquant un modèle de consommation sur les états successifs du microprocesseur, nous pouvons alors définir une trace abstraite de consommation de courant. En collectant différentes traces de consommation abstraites, nous sommes alors capables de faire une attaque par DPA.

4.3 Relation entre résistance théorique et résistance réelle

Un modèle prétend montrer un fonctionnement particulier de structures formelles ; c'est une abstraction de la réalité. Notre modèle de consommation théorique est une abstraction de la consommation réelle d'un microcontrôleur et comme tout modèle il possède ses propres limites. Par exemple, notre modèle considère que toutes les pistes des bus présentes sur le microcontrôleur sont de la même longueur et que chaque piste consomme la même quantité de courant. De plus, il considère que tout se déroule dans un environnement non bruité alors que dans la réalité, les mesures sont affectées par différentes sources de bruit.

Au vu de notre modélisation, nous devons nous demander si d'une part une implémentation qui ne fuit pas lors de la simulation pourra fuir une fois embarquée sur un composant réel et d'autre part si une implémentation qui fuit au cours de la simulation continuera à fuir lorsqu'elle sera exécutée sur un composant réel.

En premier lieu, si la simulation montre qu'une implémentation est résistante à une attaque par *DPA* (i.e. aucune source de fuite n'a été identifiée) alors nous ne pouvons pas assurer qu'il en sera de même une fois le programme implanté dans un composant réel. En effet, notre modèle théorique reste une abstraction et omet la modélisation de phénomènes physiques qui pourraient être des sources de fuites dans la réalité (e.g. longueur des pistes d'un bus). Néanmoins, nous souhaitons un peu tempérer l'affirmation précédente. Nous travaillons dans un environnement non

bruité où il n'y a aucune interférence lors de l'acquisition des courbes de consommation. Ceci nous porte à croire que dans la réalité les fuites subtiles qui ne seraient pas prises en compte par la modélisation pourraient être masquées par du bruit.

En second lieu, nous dirons que si une fuite apparaît au niveau de la simulation alors elle continuera à apparaître une fois l'algorithme implanté sur un composant réel. La simulation permet d'exhiber les sources de fuites d'une implémentation et en ce sens elle peut être vue comme un environnement qui permet de s'apprêter aux attaques physiques : c'est une *pré-attaque* qui prépare le terrain à une attaque réelle qui concentrera tous ses efforts sur les sources de fuites identifiées.

Cependant, toute la question consiste à savoir si oui ou non cette fuite pourra être détectée sur le composant réel. Autrement dit, une fois implanté sur un composant réel, il est possible que certaines fuites très isolées (e.g. un registre particulier) soient masquées par la consommation générée par les autres composants du microcontrôleur et qu'elles ne soient pas détectables. Néanmoins, tout est une question de granularité et s'il est certain que la consommation d'un registre particulier sera difficile (voire impossible) à analyser dans le monde réel, il reste possible d'analyser la consommation de composants spécifiques (e.g. mémoires). En effet, gardons à l'esprit que des attaques par analyse du rayonnement électromagnétique existent (voir section 3.3.3) et qu'elles permettent de cibler la consommation d'un composant particulier tout en faisant abstraction du bruit environnant des autres composants.

Quoiqu'il en soit, l'environnement de simulation permet d'affiner la recherche de fuites et améliore, aussi bien en rapidité qu'en précision, le travail réalisé sur les bancs de tests DPA.

4.4 Étude de l'existant

Comme nous venons de le voir, notre but est d'étudier la résistance de l'implémentation d'un algorithme cryptographique aux attaques par canaux auxiliaires d'un point de vue aussi bien théorique que pratique. Notre travail se focalise sur les microcontrôleurs AVR Atmel[®] car ils sont très répandus dans l'industrie. Au cours de ces dix dernières années, plusieurs projets traitant des aspects de simulation des microcontrôleurs ont vu le jour. Ces différents projets peuvent être classés en deux catégories à savoir les simulateurs classiques et les simulateurs d'attaques par canal auxiliaire.

4.4.1 Les simulateurs classiques

Par simulateur classiques, nous entendons les simulateurs qui se contentent de reproduire le fonctionnement d'un microcontrôleur mais qui ne traitent aucun aspect sécuritaire (e.g. pas de simulation d'attaques). Dans cette partie, nous présentons brièvement trois principaux simulateurs implémentés pour des microcontrôleurs AVR Atmel[®] et qui sont largement utilisés par la communauté des développeurs.

Le premier simulateur que nous présentons est proposé par la société IAR SYSTEMS (voir [55]) qui en plus de fournir des solutions de développement pour l'AVR (e.g. compilateurs, débogueurs), offre la possibilité de simuler l'exécution d'un programme.

Ensuite, vient AVRORA (voir [56]) qui est un simulateur implémenté en JAVA et reproduisant le comportement des microcontrôleurs présents dans des réseaux de capteurs. Il travaille au niveau des instructions machines et sa précision se situe au niveau des cycles d'horloge. D'après leurs auteurs ce simulateur surpasse des travaux précédents réalisés pour simuler des réseaux de capteurs tels que TOSSIM [57] ou ATEMU [58].

Enfin, il y a SIMULAVR (voir [59]) qui est un simulateur supportant la plupart des microcontrôleurs de la famille AVR. Il peut être utilisé conjointement avec GDB [60] afin de faire du débogage au niveau du code source écrit en langage C. Notons que ce simulateur a été implémenté en langage C et qu'il reste à ce jour le plus rapide de tous les simulateurs libres pour la famille de microcontrôleurs AVR.

4.4.2 Les simulateurs d'attaques

Nos recherches sur des simulateurs existants ont abouti à la découverte d'un seul simulateur logiciel d'attaques par canal auxiliaire qui se nomme PINPAS (voir [61, 62]). Ce simulateur a été développé en JAVA et il travaille au niveau des instructions afin de réaliser des attaques physiques. Le simulateur développé par nos soins est proche de PINPAS car il emploie la même approche pour calculer les traces de consommation permettant de faire une attaque par DPA.

Cependant, contrairement à PINPAS, notre simulateur emploie différents modèles abstraits de consommation permettant d'isoler et de localiser très précisément des fuites d'information. De plus, nous sommes capables de cibler des parties spécifiques du microcontrôleur pour réaliser une DPA. Par exemple, nous sommes capables de concentrer notre analyse sur des adresses manipulées par des pointeurs. En outre, nous sommes intéressés par les aspects théoriques de la résistance des programmes embarqués aux attaques par canaux auxiliaires. Nous prévoyons donc de faire évoluer notre simulateur en une version écrite en fonctionnel pur de telle sorte qu'il puisse être utilisé dans un assistant de preuve (voir [63, 64]). De ce fait, nous serions capables d'écrire des preuves formelles permettant de démontrer qu'une implémentation est résistante ou non à une attaque par DPA. Ainsi, nous avons besoin d'un simulateur qui soit suffisamment rapide pour faire des attaques pratiques mais aussi qui puisse être transformé en une sémantique opérationnelle utilisable dans un assistant de preuve. L'assistant de preuve Coq (voir [63]), permet de faire de la vérification de programme purement fonctionnel écrits en ML (voir [65]). En écrivant notre simulateur en OBJECTIVE CAML (qui est un dérivé du langage ML, voir [66, 67, 68]) et en utilisant ses traits fonctionnels (même si la terminaison des fonctions reste à démontrer) nous souhaitons pouvoir le *plonger* dans l'assistant de preuve Coq.

4.5 Présentation du simulateur

Ce simulateur (voir [69]) a été implémenté dans le but de reproduire le comportement des microcontrôleurs AVR 8-bit d'Atmel[®]. Pour nos expérimentations, nous nous sommes focalisés sur la simulation des microcontrôleurs AT90S8515 et ATmega128 (voir chapitre 2). Cependant, nous devons préciser que le simulateur que nous avons développé ne se restreint pas à la simulation des deux architectures citées précédemment. En effet, il est générique et prend en charge toute la gamme des familles AT90XX et ATmegaXX (voir par exemple [70]).

Le simulateur prend en entrée un programme binaire compilé pour une architecture Atmel[®]. Cette approche a été retenue car il apparaît pertinent de considérer le dernier maillon de la chaîne de compilation : le code prêt à être chargé puis exécuté par un microprocesseur de carte à puce. La figure 16 donne une représentation simple de l'architecture du simulateur. L'architecture intègre deux modules qui peuvent être utilisés durant la phase de simulation :

- le module d'attaques en fautes qui utilise un fichier de configuration renseigné par l'utilisateur permettant d'injecter des fautes durant la simulation,
- le module d'analyse de consommation qui permet d'observer la consommation théorique du microcontrôleur (par rapport à un modèle de consommation).

Ces deux modules peuvent être combinés afin de monter des attaques complexes visant à tester la robustesse d'un programme.

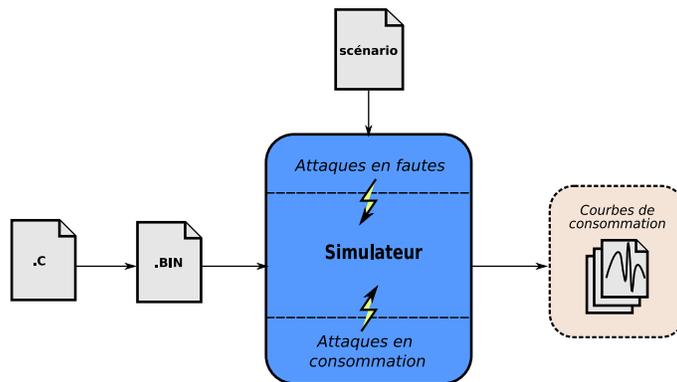


FIG. 16: Architecture de l'environnement de simulation

4.5.1 Fonctionnement du simulateur

Le cœur du simulateur est composé de quatre parties principales décrites ci-après :

Chargement. L'étape de chargement consiste à lire un fichier binaire compilé pour une architecture AVR d'Atmel[®]. Pendant cette étape, les informations contenues dans le fichier binaire sont chargées dans une structure de données permettant

de faciliter la simulation. La structure de données retenue est une table de hachage dont les clefs (resp. le contenu) sont les adresses des codes opérations (resp. les codes opérations³). Cette structure de données présente l'avantage de donner un accès en temps constant aux instructions. En effet, les instructions ne sont accessibles que par leurs adresses via le compteur ordinal. Ainsi, en définissant les clefs de la table de hachage comme étant les adresses des instructions, les accès se font en $O(1)$.

Décodage. Les instructions AVR sont constituées d'une opération et au maximum de deux opérands⁴. L'étape de décodage permet de passer de la représentation binaire d'une instruction à sa représentation mnémonique suivie éventuellement des opérands manipulées. À la fin de l'étape de décodage, la mnémonique et ses opérands éventuelles sont stockées dans une structure de données fortement typée. Cette structure contient toutes les informations nécessaires pour une exécution.

Exécution. Le processus d'exécution aiguille une instruction décodée vers sa fonction d'interprétation correspondante. Cette fonction a pour but de simuler l'exécution de cette instruction en mettant à jour l'état courant du microprocesseur (e.g. les valeurs de registres, la mémoire, les indicateurs (*flags*), etc.).

Consommation de courant. La dernière étape du processus de simulation consiste à établir un profil de consommation pour l'instruction précédemment exécutée. Ce profil est calculé par rapport à un modèle de consommation défini par l'utilisateur puis passé en paramètre au simulateur. Les modèles de consommation utilisés sont détaillés dans la section 4.6.

Évolution possible

Nous venons de détailler les grandes étapes de fonctionnement de notre simulateur. Nous souhaitons toutefois évoquer une possible évolution concernant la façon d'établir des profils de consommation. Comme nous venons de le voir, nous établissons des profils de consommation théoriques, toutefois nous pouvons essayer de définir des profils de consommation par l'expérience. Autrement dit, nous pensons à des profils empiriques qui soient plus détaillés et qui reflètent plus ce qui se passe dans la réalité (e.g. prise en compte du bruit voir [71]).

Pour ce faire, nous pourrions partir d'un véritable microcontrôleur à qui nous ferions exécuter des instructions spécifiques. Ensuite, à l'aide d'un oscilloscope numérique, nous pourrions déterminer de façon empirique pour chaque instruction et ses opérands un profil de consommation (via une analyse statistique) : nous obtiendrions une caractérisation de la consommation du périphérique visé. Ainsi, dans

³voir section 2.1.5

⁴Certaines opérations peuvent ne pas posséder d'opérands. C'est notamment le cas de l'instruction RET qui indique la fin de sous-programmes.

notre simulateur, nous pourrions mettre en correspondance l'instruction et ses opérandes en cours d'exécution avec leur profil de consommation déterminé de façon empirique, ce qui nous permettrait d'obtenir des courbes de consommation plus précises.

Limitation. Cette approche, bien que prometteuse, reste difficile à mettre en place et n'est réaliste que sur des architectures huit bits.

En effet, établir un profil de consommation de la sorte requiert pour chaque instruction de faire varier la valeur de chacune de ses opérandes. Par exemple, en AVR l'instruction ADD manipule simultanément deux registres de huit bits, nous aurions alors 256^2 profils de consommation ce qui est conséquent et nous ne parlons même pas du temps d'acquisition. Sur une architecture seize bits, nous aurions 65536^2 soit 4294967296 profils de consommation différents ce qui nous fait goûter les limites de cette approche.

4.5.2 Scénario d'attaque physique

L'environnement de simulation fournit une interface permettant de réaliser des attaques par injection de fautes. Il est ainsi possible de venir perturber de façon très précise le compteur ordinal ainsi que les registres de travail du microprocesseur. Le scénario d'attaque s'établit en remplissant un fichier de configuration dans lequel un utilisateur indique à quel moment (en nombre de cycles d'horloge) doit intervenir la perturbation et quel sera son effet (la valeur que prendra le compteur ordinal ou le registre). La figure 17 présente un exemple de scénario d'attaque et nous détaillons ci-après les différents champs présents dans le fichier de configuration :

- `cycles` : le cycle auquel l'attaque doit avoir lieu,
- `name` : le nom du registre qui doit être attaqué (e.g. 1 pour R1),
- `value` : la nouvelle valeur que doit prendre la partie attaquée.

```
[Registers]
  cycles = 6, name = 1, value = 25;
[EndOfRegisters]

[ProgramCounter]
  cycles = 29, value = 43;
[EndOfProgramCounter]
```

FIG. 17: Exemple de scénario visant à attaquer le registre de travail R1 et le compteur ordinal.

La simulation d'injection de fautes a été pensée afin de couvrir l'état de l'art des attaques en fautes y compris les attaques par double impulsion laser (i.e. deux impulsions laser successives sur un même endroit du microcontrôleur). Ce greffon

d'attaque physique peut aussi être utilisé pour valider des contre-mesures qui ont été ajoutées dans un programme afin de détecter d'éventuelles perturbations physiques.

4.5.3 Performances

Nous souhaitons que notre simulateur fasse partie intégrante d'un processus de développement, ce qui signifie que ses performances en terme de vitesse de simulation sont importantes. Dans cette partie, nous comparons les performances du simulateur que nous avons développé avec les simulateurs existants nommés SIMULAVR et AVRORA sur l'implémentation de l'algorithme DES de GNUPG. Nous devons préciser que PINPAS ne figure pas dans notre tableau de comparaison des performances car aucune version de l'outil n'est disponible en ligne. De plus, les articles le concernant ne comparent pas ses performances avec d'autres simulateurs.

Comme l'illustre le tableau 3, notre simulateur (appelé ici OSCAR) est 2,7 fois plus lent que SIMULAVR mais il est 2,6 fois plus rapide que AVRORA. Ces différences

	SIMULAVR	OSCAR	AVRORA
Temps (en sec.)	0,11	0,29	0,78
Langage	C	OCaml	Java

TAB. 3: Comparaison entre les différents simulateurs.

s'expliquent par les choix des langages de programmation retenus pour implémenter les simulateurs. Cependant, ce test nous permet de voir que les performances de notre simulateur sont acceptables et qu'il est utilisable en pratique.

4.6 Modèles de consommation de courant

Il existe deux principaux modèles formels pour corréler la consommation de courant d'un microprocesseur aux données qu'il manipule (voir par exemple [35, 40, 72]). Ces deux modèles sont la distance de Hamming et le poids de Hamming. Nous décrivons à présent ces deux modèles.

4.6.1 Le modèle distance de Hamming

Dans un microprocesseur manipulant des registres de m bits, les données binaires sont encodées par $D = \sum_{j=0}^{m-1} d_j 2^j$ où les bits d_j prennent les valeurs zéro ou un. Le poids de Hamming est une fonction qui compte le nombre de bits dont la valeur vaut un dans un mot binaire soit $H(D) = \sum_{j=0}^{m-1} d_j$. Il est généralement admis que les fuites d'information qui peuvent être observées, via le canal auxiliaire de consommation de courant, dépendent du nombre de bits basculant d'un état à l'autre à un instant donné. Un microprocesseur peut être modélisé comme une machine à états où les transitions entre états sont déclenchées par des événements tel que le

front montant d'un signal d'horloge. Cette approche est pertinente quand on regarde comment est implémentée une porte logique élémentaire dans la technologie CMOS (voir [73]) ; le courant consommé est relié à l'énergie requise pour faire basculer l'état des bits.

Soit R une valeur binaire manipulée précédemment par le microcontrôleur, alors le nombre de bits basculant de R à D est donné par $H(R \oplus D)$, aussi appelé distance de Hamming entre R et D . Nous supposons qu'il existe une relation affine entre la consommation de courant et $H(R \oplus D)$. Ce modèle ne représente pas la consommation globale d'un microcontrôleur mais seulement la consommation des parties dépendant des données. Ce n'est pas irréaliste car les pistes des bus sont traditionnellement considérées comme les éléments qui consomment le plus à l'intérieur d'un microcontrôleur. Ainsi, le modèle de dépendance de données peut s'exprimer par l'équation suivante :

$$Y = aH(R \oplus D) + b$$

où Y est la consommation de courant, b une modélisation du bruit lors de l'acquisition de la consommation et a une pondération entre la distance de Hamming et le courant consommé Y . Avec ce modèle, un attaquant peut établir une correspondance entre les valeurs successives transférées sur des bus et la consommation de courant.

4.6.2 Le modèle poids de Hamming

Le modèle que nous présentons ici implique une relation affine entre le courant consommé et le poids de Hamming d'une valeur manipulée à un instant donné. Ce modèle est décrit par l'équation suivante (la notation est la même que celle utilisée précédemment) :

$$Y = aH(D) + b$$

Ce modèle est valide dans le cas de microcontrôleurs remettant leurs bus à zéro après qu'une valeur ait transité dessus ; on parle alors de bus pré-chargés. Dans ce cas précis, la consommation de courant dépend du poids de Hamming des données présentes sur le bus car $H(0 \dots 0 \oplus D) = H(D)$. C'est le modèle original qu'a utilisé Kocher pour réaliser sa DPA (voir [38]).

4.7 Modèle de consommation abstrait

D'une manière générale, les microcontrôleurs ont une structure commune sur laquelle transite les données. Cette structure est composée de registres de travail, de bus de données, de mémoire et d'une unité arithmétique et logique ainsi que d'une interface de communication. Un attaquant focalisera ses efforts sur ces parties car elles peuvent fuir lorsqu'elles manipulent des données. Dans un premier temps nous allons appliquer le modèle de distance de Hamming entre deux états consécutifs du microprocesseur. Un état est composé des parties suivantes :

- le registre d'état (SREG),
- les registres de travail (GPR),
- le pointeur de pile (SP),
- la mémoire (MEM).

Exécuter un programme sur notre simulateur revient à exécuter un programme dans un environnement non bruité. Autrement dit, nous avons utilisé des modèles de distance de Hamming et de poids de Hamming ne modélisant pas de bruit. De plus, pour l'instant, notre modèle ne prend pas en compte les interfaces de communications car elle ne sont pas nécessaires pour les expérimentations menées (voir section 4.9).

Modèle formel

Nous estimons à présent la consommation de courant du microprocesseur avec le modèle de la distance de Hamming. Cela revient à appliquer la fonction de la distance de Hamming sur chacune des composantes du microcontrôleur détaillées ci-dessus. De façon formelle, nous définissons un état du microprocesseur (CS) à un instant t de la simulation par :

$$CS_t = \langle SREG_t, GPR_t, SP_t, MEM_t \rangle$$

Nous pouvons estimer la consommation (EC_t) en appliquant la formule de la distance de Hamming aux différentes composantes de l'état d'un microprocesseur :

$$\begin{aligned} EC_t &= \alpha.d(SREG_{t-1}, SREG_t) \\ &+ \beta.d(GPR_{t-1}, GPR_t) \\ &+ \gamma.d(SP_{t-1}, SP_t) \\ &+ \lambda.d(MEM_{t-1}, MEM_t) \end{aligned}$$

où α, β, γ et λ correspondent à des coefficients de pondération qui permettent d'ajuster au mieux le modèle de consommation de certaines parties du microcontrôleur.

Ainsi, en stockant les consommations successives estimées dans une liste, nous obtenons une trace de consommation abstraite (ACT) du microprocesseur :

$$ACT = [EC_0; EC_1; \dots; EC_{t-1}; EC_t; \dots; EC_n]$$

4.8 Attaque différentielle sur le DES

Dans la section 3.3.2 nous avons présenté les principes généraux de la DPA sans nous intéresser à son application sur un algorithme cryptographique particulier. Dans cette partie, nous commencerons par rappeler comment fonctionne l'algorithme de chiffrement DES avant de présenter en détail les différentes étapes qui permettent de réaliser une DPA sur cet algorithme.

4.8.1 Le système de chiffrement DES

Le système de chiffrement DES (*Data Encryption Standard* voir [53]) est un système de chiffrement symétrique⁵ développé par la société IBM. Il fut adopté comme standard de chiffrement pour des applications non classifiées par le NIST (*National Institute of Standards and Technology*) en 1977. L'algorithme chiffre un bloc de texte de 64 bits en utilisant une clef K de 56 bits. Il permet d'obtenir des blocs de texte chiffrés de 64 bits. L'algorithme se déroule en trois étapes :

1. Soit x un bloc de texte clair de 64 bits sur lequel est appliquée une permutation $IP(x)$ afin d'obtenir une chaîne x_0 . Nous obtenons donc : $x_0 = IP(x) = L_0R_0$ où L_0 contient les 32 premiers bits de la chaîne x_0 et R_0 contient les 32 bits restants.
2. 16 itérations d'une certaine fonction f dépendant de la clef K sont effectuées. On calcule $L_iR_i, 1 \leq i \leq 16$ en appliquant la règle suivante :

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \end{cases}$$

où \oplus représente le ou-exclusif bit à bit de deux chaînes. f est une fonction à deux variables, l'une de 32 bits (correspondant à R_{i-1} à la $i^{\text{ème}}$ itération) et l'autre composée des 48 bits de la clef K donnés dans un ordre particulier. La figure 18 représente un tour de chiffrement du DES.

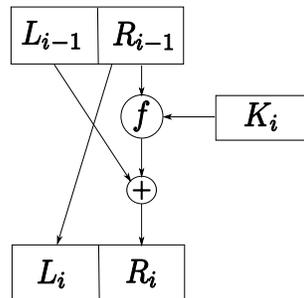


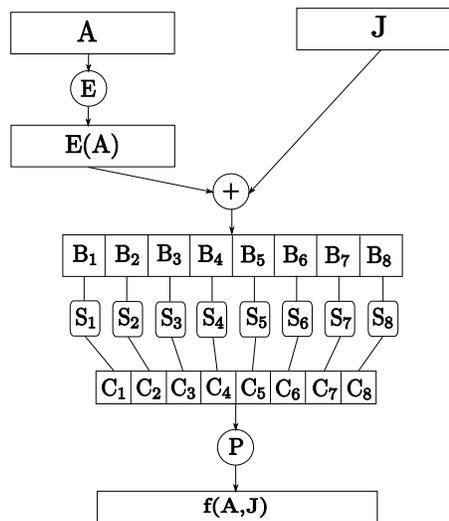
FIG. 18: Un tour de DES.

3. La permutation inverse IP^{-1} est appliquée à $R_{16}L_{16}$ pour obtenir un bloc de texte chiffré $y = IP^{-1}(R_{16}L_{16})$. Dans ce dernier tour, R_{16} et L_{16} ont été inversés.

La sécurité du DES repose sur la fonction f . Cette fonction prend en entrée deux arguments nommés A et J où A est une chaîne de 32 bits correspondant à la partie droite du bloc à chiffrer et J est une chaîne de 48 bits correspondant à une clef diversifiée. Le calcul de $f(A, J)$ se décompose en plusieurs étapes :

⁵Cela signifie que la même clef secrète est utilisée pour chiffrer et pour déchiffrer.

1. Le premier argument A subit une permutation expansive E , c'est-à-dire que les 32 bits sont permutés et certains sont répétés. Ceci amène à une nouvelle chaîne de 48 bits.
2. $B = E(A) \oplus J$ est calculé et découpé en 8 sous-chaînes consécutives de 6 bits chacune : $B = B_1 B_2 B_3 \dots B_8$. Chacune des sous-chaînes B_i est passée en entrée à une boîte de substitution S_i (les boîtes-S ou *SBox* en anglais) afin de donner en sortie un bloc C_i de 4 bits.
3. La chaîne de 32 bits $C = C_1 C_2 C_3 \dots C_8$ est réordonnée en suivant une permutation fixée P . Le résultat $P(C)$ définit $f(A, J)$.

FIG. 19: La fonction f du DES.

4.8.2 Differential Power Analysis sur le DES

Cette attaque a été proposée en 1999 par KOCHER et al. dans [38] mais depuis d'autres ont vu le jour (voir par exemple [51]). L'attaque repose sur une fonction de sélection $D(C, b, K_s)$ qui consiste à calculer la valeur du bit $0 \leq b < 32$ de la sous-chaîne L au seizième tour du DES pour un chiffré C ; les six bits entrant dans la boîte-S correspondants au bit b sont représentés par $0 \leq K_s < 2^6$. La DPA utilise l'analyse de la consommation de courant pour déterminer si une sous-clef K_s est corrélée avec un bit b de la sous-chaîne L . L'implémentation de la DPA se déroule en trois étapes principales décrites ci-dessous :

1. La première étape consiste à acquérir m courbes de consommation du DES ciblé pour l'attaque. Pour cela, nous générons m fichiers binaires qui contiennent la même implémentation du DES et qui utilisent la même clef secrète K mais des messages clairs $P_{1..m}$ différents. Au préalable, tous les chiffrés $C_{1..m}$ correspondant aux messages clairs $P_{1..m}$ ont été calculés puis sauvegardés. Chaque

fichier binaire est ensuite passé comme paramètre au simulateur afin d'obtenir sa consommation théorique CT_i . Nous stockons dans une structure de données tous les couples (CT_i, C_i) afin d'avoir une correspondance entre les consommations théoriques et leurs chiffrés associés.

2. La deuxième étape consiste à calculer la valeur de L à l'entrée du seizième tour. Afin de réaliser cette opération, nous créons une fonction nommée $DES^{-1}(C, K_s)$ qui consiste à inverser le dernier tour du DES. Cette fonction prend en argument deux paramètres : C le chiffré et K_s qui est une sous-clef.
3. La dernière étape consiste à appliquer la fonction de sélection $D(C_i, b, K_s)$ qui permet de répartir les courbes de consommation théorique CT_i en deux ensembles. Si le bit b calculé est égal à un alors nous mettons la courbe de consommation dans un ensemble A sinon nous la mettons dans un ensemble B . Une fois ces deux ensembles obtenus, il ne reste plus qu'à appliquer le principe de la DPA décrit dans la section 3.3.2 et à déterminer si K_s et b sont corrélés.

4.9 DPA théorique sur le DES

L'environnement de simulation a été utilisé pour attaquer une l'implémentation non sécurisée de l'algorithme de chiffrement DES (voir section 4.8.1). Pour nos expérimentations, nous avons utilisé l'algorithme DES provenant de la suite logicielle GNUPG (voir [74]). Cette implémentation du DES a été compilée avec `avr-gcc 4.3.3` (voir [75]) pour une cible `ATmega128` (voir [19]). Le résultat de la compilation est un fichier objet et nous utilisons `avr-objcopy` avec l'option `-O binary` pour le transformer en un fichier binaire valide pour la cible spécifiée.

Dans un premier temps, nous chercherons à valider l'attaque par rapport aux modèles que nous avons définis afin de vérifier que tout fonctionne correctement. Puis, nous expliquerons comment il est possible grâce au simulateur de concentrer une attaque sur des parties très précises du programme ou du microcontrôleur au travers du concept de DPA chirurgicale. Enfin, nous terminerons en commentant les différents résultats obtenus.

4.9.1 DPA théorique

Nous cherchons tout d'abord à valider l'attaque par DPA par rapport au modèle de la distance de Hamming (défini dans la section 4.6.1). Nous considérons trois critères pour comparer les résultats obtenus :

- le modèle de consommation,
- le nombre de traces de consommation,
- le temps nécessaire pour retrouver la dernière sous-clef.

Notre modèle formel (défini dans la section 4.7) prend en compte les principales parties du microcontrôleur qui consomment du courant ; ce qui signifie que la

consommation théorique que nous obtenons (voir figure 20) est corrélée à la consommation réelle du périphérique.

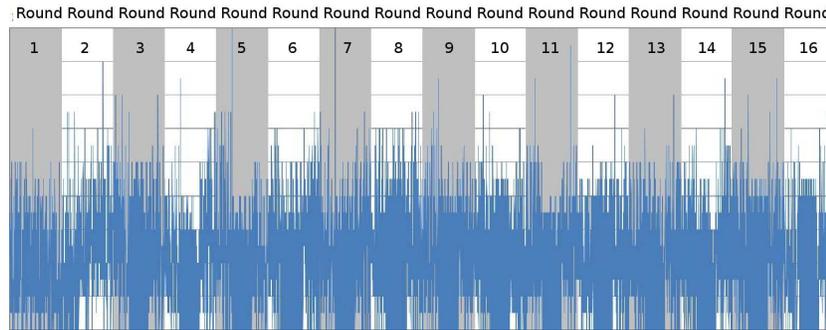


FIG. 20: Exemple d'une trace de consommation théorique pour l'algorithme DES.

Pour les premiers essais de l'attaque par DPA, 1000 traces de consommation théorique ont été collectées. Il s'est avéré que ce nombre de traces était suffisant pour révéler les 48 bits de la dernière sous-clef en 6 minutes environ sur un Intel Core 2 Duo cadencé à 2.1 GHz avec 4 Go de RAM. Nous avons voulu déterminer pour ce modèle le nombre de traces minimum nécessaires pour retrouver la dernière sous-clef du DES. En procédant par dichotomie, nous avons trouvé que 540 traces de consommation étaient suffisantes pour retrouver la clef en 3 minutes et 5 secondes. Les résultats obtenus sont rassemblés dans le tableau 4.

# Traces	# Sous-clefs	Temps
250	3	1'30''
500	6	3'
540	8	3'05''
1000	8	6'

TAB. 4: Résultats de la DPA avec le modèle de distance de Hamming.

Les résultats présentés ici permettent de valider notre approche et de vérifier que par la simulation il est possible de retrouver la clef secrète utilisée par le DES. De plus, le temps nécessaire pour retrouver toute la clef est tout à fait acceptable (i.e. un peu plus de trois minutes) ce qui nous conforte dans le fait que notre façon de vérifier si un programme est vulnérable à une DPA peut être intégrée dans un processus de développement.

4.9.2 Introduction du concept de « DPA chirurgicale »

L'environnement de simulation offre plusieurs possibilités pour analyser très précisément le comportement du microcontrôleur. D'une part, il est possible d'étudier un composant particulier du microcontrôleur pour vérifier s'il fuit ou non (i.e. laisse

fuir des informations censées être secrètes). D'autre part, l'étude d'une partie du programme en cours d'exécution devient réalisable.

Plusieurs expérimentations ont été menées en se focalisant sur les parties suivantes du microcontrôleur :

- l'adresse des boîtes-S manipulées par le microcontrôleur,
- les données transférées sur les bus pendant les opérations de lecture et d'écriture,
- la permutation de la fonction f .

Dans ce qui suit nous présentons en détail ces expérimentations.

Adresses des boîtes-S

L'idée est de déterminer quelles sont les instructions AVR qui manipulent les adresses qui permettent d'accéder aux boîtes-S car ces instructions manipulent une donnée sensible. En désassemblant le binaire AVR du DES, il s'avère que seulement l'instruction LDD est utilisée pour accéder à la mémoire. Cette instruction charge un octet contenu à une certaine adresse de l'espace de données dans un registre de travail. Sur les architectures Atmel[®], l'espace de données est accessible grâce à deux registres spéciaux de 16 bits appelés Y et Z (voir [76]). Le registre Y (resp. Z) est obtenu par concaténation des registres de travail R29:R28 (resp. R31:R30). En se focalisant sur ces deux registres, nous obtenons un modèle de consommation très précis que l'on nomme (Y, Z) .

Données des boîtes-S transférées sur des bus

Comme mentionné précédemment, nous considérons une implémentation non sécurisée du DES. Ceci signifie que les données transférées sur les bus ne sont pas masquées et peuvent laisser fuir de l'information éventuellement corrélée à la clef. Tous les transferts de données sont effectués pendant les opérations d'écriture et de lecture. Pour analyser ces accès, nous ne devons pas considérer les adresses manipulées mais leur contenu. Nous définissons alors trois modèles de consommation ne prenant en compte :

- que les lectures (le modèle *Lectures*),
- que les écritures (le modèle *Ecritures*),
- les lectures et les écritures (le modèle *LE*).

Ainsi, nous éliminons les composantes SREG, SP et GPR de notre modèle précédent (voir section 4.7) afin de ne considérer que les entrées/sorties de MEM.

Pour établir une trace de consommation par rapport à ce modèle, il faut étudier les valeurs manipulées par les instructions AVR qui permettent de lire (LD, LDD, LDS) et/ou d'écrire (ST, STD, STS) dans la mémoire.

Permutation de la fonction f

Nous souhaitons étudier de près la permutation de la fonction f afin de déterminer si elle peut être source de fuites d'informations. Établir une trace de consommation avec ce modèle consiste à venir repérer dans le programme l'appel à la procédure qui effectue la permutation ainsi que l'endroit où elle termine puis d'enregistrer la consommation théorique comprise entre ces deux points.

Les résultats basés sur ces nouveaux modèles de fuite sont présentés dans les parties suivantes.

4.9.3 « DPA chirurgicale » appliquée aux boîtes-S et aux bus

Dans cette partie nous étudions l'impact des modèles de consommation sur la DPA avec les modèles des adresses des boîtes-S et des données des boîtes-S transférées sur les bus (voir section 4.9.2). Nous nous intéresserons tout particulièrement au nombre de traces nécessaires pour retrouver toutes les parties de la dernière sous-clef utilisée par le DES.

Les tableaux 5 et 6 comparent les performances d'une attaque DPA en prenant en compte différents modèles de fuite. Le tableau 5 met en lumière le fait que les

# Traces	Modèles de fuite				Temps
	(Y, Z)	<i>Lectures</i>	<i>Écritures</i>	<i>LE</i>	
250	0	0	0	1	1'30"
500	2	5	5	5	3'
1000	2	8	7	8	6'
2000	6	8	8	8	12'30"
3000	8	8	8	8	18'30"

TAB. 5: DPA chirurgicale avec le modèle distance de Hamming.

modèles de fuite *Lectures*, *Écritures* et *LE* combinés au modèle de distance de Hamming ne semble pas très efficace. En effet, 1000 courbes de consommation sont nécessaires pour retrouver la dernière sous-clef contre 540 pour le modèle de fuite prenant en compte l'état complet du microprocesseur (voir tableau 4).

En revanche, se focaliser seulement sur les instructions qui manipulent les adresses des valeurs stockées dans les boîtes-S (i.e. modèle de fuite (Y, Z)), associé au modèle du poids de Hamming semble particulièrement efficace. En effet, seulement trois minutes et 500 traces de consommation sont nécessaires pour retrouver l'intégralité de la dernière sous-clef (voir tableau 6).

4.9.4 « DPA chirurgicale » et permutation

Les résultats obtenus précédemment ont permis d'une part de valider que la DPA fonctionnait correctement et d'autre part de faire émerger le concept de DPA

# Traces	Modèles de fuite				Temps
	(Y, Z)	Lectures	Écritures	LE	
250	7	1	2	3	1'30
500	8	3	2	3	3'
1000	8	7	7	8	6'
2000	8	8	8	8	12'30"
3000	8	8	8	8	18'30"

TAB. 6: DPA chirurgicale avec le modèle poids de Hamming.

chirurgicale, ce qui constitue un premier pas non négligeable. Néanmoins, nous avons attaqué des parties comme les boîtes-S et les données transférées sur les bus qui sont connues pour laisser fuir de l'information. Aussi, nous nous sommes intéressés à vouloir vérifier si d'autres parties de l'implémentation de l'algorithme DES pouvaient elles aussi laisser échapper de l'information permettant de retrouver la dernière sous-clef. C'est pourquoi nous nous sommes penchés sur la permutation de la fonction f .

Le DES provenant de la suite logicielle GNUPG sur lequel nous avons effectué tous nos tests, est un DES optimisé (mais non sécurisé) où les opérations d'expansion, de substitution et de permutation sont réalisées par le seul passage dans une boîte-S. Autrement dit, avec cette implémentation, il est impossible de venir cibler précisément la consommation de la permutation de la fonction f . Afin de pouvoir mener à bien nos expérimentations, nous avons du travailler sur un DES non optimisé dont l'implémentation est conforme au standard public décrit dans [53]. Comme modèle de consommation, nous avons utilisé la distance de Hamming.

Premiers résultats

Avec le DES optimisé de GNUPG, nous avons déterminé qu'il fallait 540 traces de consommation pour révéler les 48 bits de la dernière sous-clef. Nos expérimentations nous ont montrées qu'avec une implémentation non optimisée, seulement 375 traces de consommation étaient nécessaires ce qui représente un gain de plus de trente pourcent sur le nombre de traces nécessaires.

Ce résultat s'explique de la manière suivante. Dans l'implémentation du DES non optimisé, toutes les opérations se déroulant dans la fonction f sont décomposées en différentes parties distinctes à savoir l'expansion, le ou-exclusif avec la sous-clef de chiffrement, le passage dans les boîtes-S et enfin la permutation. Dans cette implémentation, la fonction f manipule plus de données intermédiaires pendant une longue période de temps (i.e. environ 5 000 cycles d'horloge), ce qui a pour effet de générer des courbes de consommation détaillées. Ainsi, comme les différences de consommation lors des manipulations de données sont plus marquées (i.e. plus de pics apparaissent), effectuer la corrélation avec la clef manipulée nécessite beaucoup moins de traces.

Permutation

Précédemment, nous avons mentionné que les opérations de la fonction f étaient facilement repérables, ce qui nous permet de nous focaliser seulement sur la permutation. En examinant l'exécution de la permutation, nous avons déterminé qu'elle nécessitait 1 000 cycles d'horloge, ce qui constitue un temps assez long pour obtenir des courbes de consommation exploitables par la DPA.

Nous avons effectué un premier test avec 500 traces qui s'est soldé par un échec : la DPA n'a pas retrouvée une seule partie de la sous-clef. Nous avons donc augmenté le nombre de traces pour voir si cela avait un impact sur le succès de la DPA. Nous sommes alors passés à 1 000 traces de consommation et nous avons relancé la DPA : une fois encore elle a échoué et n'a permis de découvrir aucune partie de la sous-clef. Finalement, nous avons relancé la DPA avec 2 000 traces et comme précédemment, nous n'avons retrouvé aucune partie de la sous-clef. Forts de ces expérimentations, nous avons décidé de modifier la fonction de sélection afin d'augmenter la précision de l'attaque.

Fonction de sélection « adaptée ». Nous allons présenter la nouvelle fonction de sélection que nous avons utilisée. Au lieu de faire des hypothèses sur un bit du bloc L_{15} , nous faisons des hypothèses sur un bit sortant directement d'une boîte-S juste avant son entrée dans la fonction de permutation (i.e. un bit appartenant à un bloc C_i sur la figure 19). Considérer un bit en sortie des boîtes-S est équivalent à considérer un bit après la permutation, il s'agit des mêmes bits à une permutation près. Nous avons choisi cette méthode de sélection pour des raisons pratiques : il nous est plus facile de suivre le cheminement d'un bit particulier dans l'implémentation de la fonction de permutation.

Pour nos premières expérimentations avec cette nouvelle fonction de sélection, nous avons changé la sélection des bits et lancé une DPA avec 500 traces de consommation, ce qui nous a permis de retrouver deux parties de la sous-clef (i.e. 16 bits d'information). En augmentant progressivement le nombre de traces jusqu'à 2 000 nous obtenons invariablement le même résultat : toujours les deux mêmes parties de la sous-clef apparaissent. Face à ces résultats troublants, nous avons décidé de regarder de plus près comment la fonction de permutation était implémentée.

Raffinement dans la sélection des bits. Cette implémentation de la permutation ne considère pas un bloc de trente-deux bits, mais quatre blocs de huit bits. La fonction traite les blocs un par un et à l'intérieur de chaque bloc effectue une permutation bit à bit en utilisant les données contenues dans la table de permutation (modulo quelques astuces de programmation). Ainsi, cette fonction retourne en sortie quatre blocs de huit bits dont la concaténation correspond bien à la permutation du mot de trente-deux bits de départ. Le code source écrit en langage C est présenté sur la figure 21 où `P_permutation` correspond à la table de permutation, `*in` aux données à permuter et `*out` au résultat de la permutation.

```

void perm32( iu8 *out, iu8 *in) {
    iu8 i,j, off, oct;

    /* Itérations sur les blocs de 8 bits */
    for (i=0; i<4; i++) {
        out[i] = 0;

        /* Traitement des bits à l'intérieur d'un bloc */
        for (j=0; j<8; j++) {
            out[i] = out[i] << 1;
            oct = in[P_permutation[i*8+j] / 4];
            off = 3 - (P_permutation[i*8+j] % 4);
            out[i] = out[i] | ((oct >> off) & 0x01)
        }
    }
}

```

FIG. 21: Code C de la fonction de permutation.

Notre intuition est que pour une DPA, quand une donnée est manipulée longtemps, sa consommation influe longtemps sur le signal et l'attaque s'en trouve plus efficace. En regardant les détails d'implémentation, nous nous rendons compte que le premier bit permuté est stocké dans la partie la plus à droite du bloc et qu'à chaque nouvelle permutation, il est décalé d'une position sur la gauche par l'instruction `out[i] = out[i] << 1`; (le bit représenté en rouge sur la figure 22).

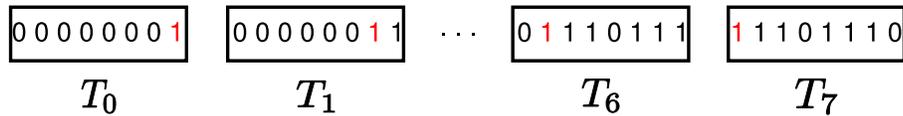


FIG. 22: Ordre de mise à jour des bits permutés.

Pour effectuer notre attaque par DPA, sélectionner à la sortie des boîtes-S le premier bit sur lequel portera la permutation (i.e. celui qui sera manipulé le plus longtemps) nous a permis d'obtenir toutes les parties de la dernière sous-clé avec 500 traces; ce qui confirme notre intuition première. Ces résultats encourageants nous ont poussés à vouloir déterminer quel était, pour cette implémentation de la permutation, le nombre minimum de traces nécessaires. Il s'est avéré que seulement 30 traces ont suffi à retrouver toutes les parties de la dernière sous-clé. Le tableau 7 présente un récapitulatif du nombre des parties de la sous-clé découvertes avec la fonction de sélection *classique* puis *adaptée*.

Fonction de sélection	# de traces				
	30	250	500	1000	2000
<i>Classique</i>	0	0	0	0	0
<i>Adaptée</i>	8	8	8	8	8

TAB. 7: Fonction de sélection et DPA sur la permutation.

Discussion sur la permutation et la DPA. À notre connaissance, aucune expérimentation visant à montrer qu'uniquement la permutation de la fonction f laisse fuir ou non de l'information n'avait été réalisée.

S'il est certain que nos expérimentations se basent sur des modèles de consommation théorique, elles tendent néanmoins à montrer que pour l'implémentation que nous avons considérée, la permutation est une partie critique qui laisse transpirer de l'information concernant la sous-clef. Nous pouvons même dire qu'elle laisse transpirer beaucoup d'informations puisque seulement 30 traces ont été nécessaires pour retrouver toutes les parties de la dernière sous-clef.

C'est d'ailleurs cet aspect qui est assez troublant. Les premières expérimentations que nous avons menées ne tenaient pas compte de la façon dont les bits seraient permutés et même avec un nombre de traces très important (i.e. 2 000 traces) aucune corrélation n'a pu être établie. En revanche, dès l'instant où nous avons compris comment se déroulait la permutation et que nous avons choisi les bits visés par la DPA de façon à ce qu'ils soient manipulés le plus longtemps possible, les résultats ont dépassés nos espérances. Néanmoins, ces expérimentations n'en sont qu'à leurs débuts et le fait que la DPA appliquée sur la permutation ne requiert que 30 traces de consommation nécessite d'être confirmé.

Le concept de DPA chirurgicale a été poussé à son maximum car après avoir ciblé un ensemble d'opérations très précises comme la permutation de la fonction f , nous avons fait le lien avec l'implémentation et adapté de façon très précise la fonction sélection, chose qui ne s'était jamais produite jusqu'à présent dans nos expérimentations. Cette nouvelle façon de procéder nous ouvre de nouvelles perspectives d'améliorations du concept de DPA chirurgicale. Enfin, nous aimerions mener de nouvelles investigations en réalisant à nouveau une DPA sur la fonction f complète mais cette fois-ci en sélectionnant les bits qui marquent le plus pour la permutation. Au travers de cette nouvelle expérimentation nous souhaiterions vérifier si cette approche permet de faire chuter grandement le nombre de courbes nécessaires pour retrouver la dernière sous-clef.

4.9.5 Commentaires sur la DPA théorique

Le concept de DPA théorique, qui consiste à analyser la résistance intrinsèque d'une implémentation face à des attaques qui analysent la consommation, n'est pas nouveau en soi. En effet, certains travaux universitaires ont montré qu'il était possible d'effectuer une DPA théorique sur une implémentation écrite en langage C

de l'algorithme DES. Si ces résultats sont intéressants, ils restent à un niveau qui est trop éloigné de ce qui se passe dans la réalité car en fin de compte, ni le code qui sera vraiment exécuté (i.e. programme binaire) ni le comportement du microcontrôleur ne sont pris en compte. Pour pallier à ces limitations, nous avons souhaité pousser ce concept plus en avant afin de se rapprocher au plus près de ce qui se passe dans la réalité. C'est ainsi nous avons décidé de mettre au point un environnement de simulation de microcontrôleur en vue d'analyser les fuites d'information inhérentes à une implémentation. Ces analyses ont permis de pousser à leur maximum les attaques par analyse différentielle de consommation et de faire émerger la notion de « DPA chirurgicale ».

Le fait d'approximer la consommation réelle d'un microcontrôleur dans un environnement non bruité, permet de profiler précisément la consommation de ses différents composants. Les résultats qui viennent d'être présentés (voir section 4.9) permettent de vérifier qu'il n'est pas nécessaire de modéliser la consommation de tout le microcontrôleur pour mener une attaque par analyse différentielle de la consommation. En effet, nous venons de voir qu'en se focalisant sur certaines parties du microprocesseur (i.e. en faisant une DPA chirurgicale), il était possible de retrouver la dernière sous-clef manipulée par l'algorithme de chiffrement DES. La DPA chirurgicale permet donc de concentrer les attaques sur des parties très précises du microcontrôleur telles que certains registres ou des accès mémoires. Elle permet de vérifier efficacement des hypothèses concernant les sources de fuites du microcontrôleur ou du programme et par la même occasion de mettre au point des contre-mesures ciblées.

L'autre point intéressant de ces résultats concerne l'importance du choix d'un modèle de consommation (indépendamment du modèle de fuite). Il existe plusieurs modèles théoriques de consommation, cependant pour nos expérimentations nous avons choisi les deux modèles qui sont couramment utilisés pour mener des attaques de type DPA (voir [35]). Ce que nous avons montré au travers de nos résultats, c'est que pour la simulation, le changement d'un modèle de consommation (ici le modèle de poids de Hamming et de distance de Hamming) n'influe pas sur le fait de pouvoir retrouver la clef secrète de chiffrement mais plus sur la vitesse à laquelle cette clef sera découverte.

Nous avons identifié des fuites d'information sur une implémentation non sécurisée de l'algorithme DES, ce qui permet d'identifier ses faiblesses et éventuellement de les corriger. Néanmoins la simulation a permis de mettre au jour certaines fuites très subtiles (cf. modèle (Y, Z)) et toute la question est de savoir si dans un environnement réel cette fuite pourra toujours être détectée. Malheureusement, pour des raisons de temps, nous n'avons pu pousser plus loin nos investigations.

4.10 Conclusion

Dans ce chapitre nous avons présenté un environnement logiciel dédié à l'analyse de la résistance de l'implémentation d'algorithmes cryptographiques face à des attaques de type DPA. En particulier, notre environnement permet de venir greffer différents modèles de fuite d'un microcontrôleur mais aussi différents modèles de consommation théorique et d'étudier leur impact sur une attaque de type DPA. Pour nos expérimentations, nous nous sommes focalisés sur deux modèles de consommation théorique qui sont le poids de Hamming et la distance de Hamming et nous avons montré que ces deux modèles permettent de mener à bien une attaque de type DPA.

Notre environnement permet de définir facilement des modèles de fuite associés à des modèles de consommation et de vérifier si les hypothèses de fuites sont justes. Ainsi, il permet à des développeurs de définir des contre-mesures ciblées et peu coûteuses en terme de taille de code et de performances. Sans forcément se concentrer sur une fuite en particulier, il permet à des évaluateurs de déterminer en quelques minutes si une implémentation est vulnérable à une attaque de type DPA préparant ainsi le terrain à une attaque réelle.

Nous aimerions pousser plus loin nos investigations de DPA en développant un greffon qui permettrait d'approximer les émissions électromagnétiques d'un microcontrôleur en train d'exécuter un algorithme cryptographique. Ce greffon nous permettrait alors de vérifier si une attaque théorique par analyse des émissions électromagnétiques (DEMA voir section 3.3.3) est possible et si elle est plus efficace (en terme de temps d'exécution et de nombre de courbes nécessaires) qu'une DPA théorique.

Le chapitre suivant s'inscrit dans la continuité des travaux qui viennent d'être présentés car nous introduisons un outil qui analyse des programmes écrits en langage d'assemblage AVR dans le but de déterminer s'ils sont robustes à des attaques basées sur le temps d'exécution.

Chapitre 5

Analyse de programmes écrits en langage d'assemblage

Les langages machine sont des langages de très bas niveau directement interprétés par le microprocesseur d'un ordinateur. Leur sémantique permet d'accéder à toutes les fonctionnalités du microprocesseur mais leur syntaxe reste rebutante car elle est composée d'un alphabet à deux lettres qui sont 0 et 1. Ainsi, écrire un programme en langage machine revient à aligner des suites de 0 et de 1 ce qui a pour effet de rendre les programmes incompréhensibles pour le commun des mortels. Pour pallier au problème de manipulation des langages machine, des langages de plus haut niveau ont fait leur apparition et en premier lieu arrivent les langages d'assemblage. Ces langages sont utilisés en raison de leur simplicité (toute relative) par rapport aux langages machine. En effet, un langage d'assemblage n'est en fait que la représentation textuelle d'un langage machine. Il fournit une méthode d'écriture plus lisible que celle des langages machine eux-mêmes car il utilise des noms (e.g. tels que `ADD`, `MOV` ou `DIV`) et des adresses symboliques au lieu de codes binaires et hexadécimaux. Le programme réalisant la traduction d'un langage d'assemblage en langage machine est appelé un assembleur.

Bien que très puissante, la programmation en langage d'assemblage est loin d'être aisée. Il est beaucoup plus long, mais aussi beaucoup moins sûr (en terme d'erreurs de programmation), d'écrire un programme en langage d'assemblage que dans un langage de haut niveau (e.g. C, Java, Objective Caml, etc.). Les langages de haut niveau fournissent une façon d'organiser et de structurer un programme de manière claire tout en faisant abstraction des données qui sont manipulées (e.g. on ne manipule plus explicitement des registres et des adresses mais des variables et des pointeurs). Ainsi, le temps requis par des opérations de débogage et de maintenance des programmes s'en trouve grandement réduit.

Néanmoins, deux raisons principales peuvent être données pour utiliser un langage d'assemblage : les gains de performance et les facilités d'accès aux différentes ressources de la machine. D'une part, le code d'un langage d'assemblage est beaucoup plus compact et ses performances en terme de temps d'exécution accrues car les

personnes qui l'ont développé ont apporté un soin particulier au choix et à l'ordonnement des instructions. D'autre part, le langage d'assemblage permet d'accéder directement aux ressources de la machine telles que le bit de débordement ou les interruptions. Cet aspect est crucial pour des applications embarquées voulant tirer partie de toutes les fonctionnalités d'une architecture comme c'est le cas dans les téléphones cellulaires ou les cartes à puce.

Autrement dit, contrairement aux langages de haut niveau, il est possible de déterminer quelles instructions seront exécutées, dans quel ordre et combien de temps cela prendra (voir [77]). Ce qui revient à dire qu'analyser le code source d'un programme écrit en langage de haut niveau pour rechercher des attaques basées sur son temps d'exécution (voir section 3.3.1) n'a aucun sens. Ainsi, pour s'assurer de la sécurité des programmes écrits en langages d'assemblage, un expert doit posséder une batterie d'outils permettant de faciliter son analyse.

5.1 Approche retenue

Notre but est de déterminer si un programme écrit en langage d'assemblage est vulnérable à des attaques basées sur le temps d'exécution (voir section 3.3.1). Pour déterminer qu'un programme n'est pas vulnérable à de telles attaques, il est nécessaire de montrer que son temps d'exécution est constant et qu'il ne dépend pas d'une information secrète. Pour faciliter cette analyse, nous proposons un outil semi-automatique qui permet de donner une représentation visuelle du flot d'exécution sous forme de graphes orientés et surtout qui permet de mesurer le temps d'exécution de code écrit en langage d'assemblage.

Notons qu'une telle analyse ne peut être entièrement automatique car ce problème est indécidable (voir [78, 79]) notamment en raison de la détermination du nombre d'itérations d'une boucle dans un programme. Notre outil se concentre sur le langage d'assemblage AVR d'Atmel[®]. Ce langage est fait pour être exécuté sur des architectures simples ne comportant pas de mémoire cache¹ ni de pipeline² sur plusieurs niveaux facilitant ainsi l'analyse du temps d'exécution des programmes. Notre outil propose trois fonctionnalités principales, à savoir :

- détecter les boucles et le code inatteignable,
- analyser le langage d'assemblage et reconstruire le graphe d'appel de procédures ainsi que le graphe de contrôle de flot,
- aider à analyser le temps d'exécution de chemins particuliers en fournissant un interpréteur d'expressions régulières.

Notre outil a été implémenté avec le langage OBJECTIVE CAML (voir [67]) car l'analyse sémantique de code source est facilitée grâce à la construction de types inductifs. Pour visualiser les différents graphes orientés générés, nous utilisons GRAPHVIZ

¹Petite mémoire rapide qui sert à stocker les mots mémoires les plus souvent utilisés.

²Découpage du traitement des instructions en plusieurs sections élémentaires, chacune pouvant être traitée en parallèle par un composant matériel spécifique.

(voir [80]) qui est très pratique à manipuler pour représenter ce type de structures hiérarchiques.

5.2 Étude de l'existant

Notre outil (voir [81]) analyse statiquement le temps d'exécution de programmes écrits en langages d'assemblage en additionnant les cycles d'horloge des instructions rencontrées sur des chemins d'exécution. Il fournit en plus des fonctionnalités permettant de visualiser et donc de mieux comprendre la structure des programmes. Cependant, cet aspect de visualisation n'est pas complètement nouveau. En effet, des outils de visualisation de programmes écrits en langages d'assemblage existent déjà et servent à différents desseins. Tout d'abord, certains sont utilisés à des buts éducatifs (voir par exemple [82]) notamment pour aider des étudiants à comprendre comment un compilateur génère des instructions pour un langage d'assemblage donné. Ensuite, des simulateurs de microcontrôleurs comme GSPIM [83] ou AVRORA [56] proposent une représentation visuelle du programme en cours de simulation. Enfin, les systèmes de visualisation sont utilisés dans le domaine de la sécurité afin d'aider à désobfusquer en partie du code ou rechercher des trappes (*back doors* en anglais) dans des portions de code écrites en langage d'assemblage. Le logiciel commercial IDA Pro [84] en est le parfait exemple. Cependant, bien que tous ces outils permettent de comprendre la structure d'un programme, aucun ne présente des fonctionnalités permettant de trouver des failles de sécurité en examinant le temps précis (en terme de cycles d'horloge) de certains chemins d'exécution critiques.

5.3 Représentation visuelle de langages d'assemblage

Bien que les langages d'assemblage soient très efficaces et permettent de tirer partie des fonctionnalités d'une machine donnée, leur lecture et la compréhension de leur comportement reste difficile à appréhender et ceci en raison de leur nature non structurée. Pour faciliter la compréhension de l'architecture des programmes écrits en langages d'assemblage, une solution consiste à en fournir une représentation sous forme de graphe. Dans cette partie, nous abordons la reconstruction de graphe (voir [85, 86]) pour le langage d'assemblage AVR d'Atmel[®].

5.3.1 Quelques définitions

Cette section a pour but de fixer les définitions ayant trait à la structure d'un programme. En effet, un programme est constitué d'une séquence d'instructions dont la structure obéit à certaines règles (voir par exemple [87, 88]).

Les blocs de base

Le flot de contrôle d'un programme est défini par des instructions de saut, qui sont des branchements intra-procéduraux, et des instructions d'appel de procédures, qui sont des branchements inter-procéduraux. Les branchements divisent un programme en « blocs de base » (*basic block* voir [88]), c'est-à-dire en une séquence d'instructions consécutives dans laquelle le flot de contrôle n'a aucune possibilité de s'arrêter ou d'effectuer des branchements sauf à la fin du bloc.

Les procédures

La structuration d'un programme passe par la réutilisabilité de certaines de ses parties. Les parties réutilisables d'un programme sont souvent paramétrées et portent le nom de « procédures ». Il est important de souligner que chaque bloc de base appartient à une et une seule procédure. Dans le langage d'assemblage AVR, les procédures commencent par une étiquette (*label* en anglais), c'est-à-dire un identifiant unique dans le programme et terminent systématiquement par l'instruction RET.

Les boucles

Le terme « boucle » sera utilisé pour désigner des boucles naturelles (*natural loops*) comme définies par Aho et al. [87]. Une boucle naturelle satisfait deux propriétés :

1. Une boucle naturelle doit posséder un seul point d'entrée appelé l'en-tête (*header*).
2. Une boucle naturelle doit posséder au moins un chemin de retour vers son en-tête.

Le concept de boucles naturelles est important car il interdit à un programme de venir sauter dans le corps de la boucle. Bien que ce soit une restriction, les codes compilés et les codes bien écrits à la main n'utiliseront pas d'autres types de flot de contrôle cyclique.

Code inatteignable

Il arrive que dans certains programmes, il existe des portions de code qui ne seront pas exécutées quelles que soient les données d'entrées. Ceci s'appelle du code inatteignable (*unreachable-code* dans la littérature). Il peut ne jamais avoir été exécuté pour n'importe quelles données d'entrées ou il peut avoir été obtenu à la suite d'optimisations. Nous distinguerons deux types de code inatteignable. Le premier concerne les branchements conditionnels dont l'évaluation de la condition est toujours vraie (ou fausse). Ceci implique que le flot d'exécution ne prendra jamais la seconde branche de la condition et que toute une partie de code ne sera jamais exécutée. Le second type de code inatteignable concerne des nœuds du graphe de

flot de contrôle qui n'ont aucun successeur ou prédécesseur (i.e. des blocs de base isolés).

Néanmoins déterminer si le flot d'exécution d'un programme sera toujours le même quelles que soient les données en entrée est un problème très ardu et qui se situe en dehors du cadre de cette thèse. Dans nos travaux, c'est le second type de code inatteignable qui nous intéresse et que nous détectons.

5.3.2 Graphe de flot de contrôle

Un « graphe de flot de contrôle » (*Control Flow Graph*) est un graphe dans lequel les nœuds représentent des blocs de base et les arêtes correspondent au flot de contrôle entre ces nœuds. La reconstruction d'un graphe de flot de contrôle (voir la thèse d'Henrik Theilling [86]) implique de rechercher des blocs de base qui composent un programme ainsi que leurs connexions. Pour construire les blocs de base, nous commençons par rechercher l'ensemble des instructions (ou étiquettes) qui indiquent le début des blocs et que nous nommerons des *leaders*. Pour cela nous appliquons l'algorithme défini par Aho et al. [87] :

1. La première instruction est un *leader*.
2. Toute instruction étant la cible d'un saut conditionnel ou inconditionnel est un *leader*.
3. Toute instruction qui suit immédiatement un saut conditionnel ou inconditionnel est un *leader*.

Pour chaque *leader*, son bloc de base consiste en ce *leader* suivi de toutes les instructions consécutives jusqu'au prochain *leader* (mais ne l'incluant pas). La figure 23 illustre le procédé qui vient d'être expliqué.

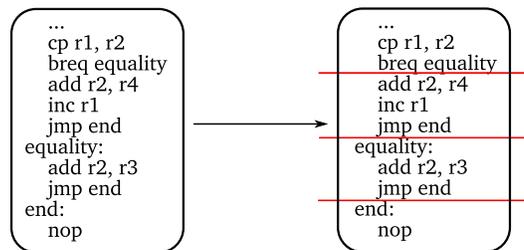


FIG. 23: Découpage en blocs de base.

L'étape suivante consiste à ajouter les arêtes entre les blocs de base dans le but d'obtenir le graphe de flot de contrôle comme illustré sur le schéma 24. Une arête est ajoutée entre les blocs de base dans les trois cas suivants :

1. Un bloc de base B_2 suit immédiatement un bloc de base B_1 dans l'ordre du programme et que B_1 ne se termine pas par un branchement inconditionnel.
2. Il y a une branche partant de la dernière instruction d'un bloc B_1 vers la première instruction du bloc B_2 (i.e. un *leader*).

3. La fin d'une procédure et la prochaine instruction du bloc de base appelant.

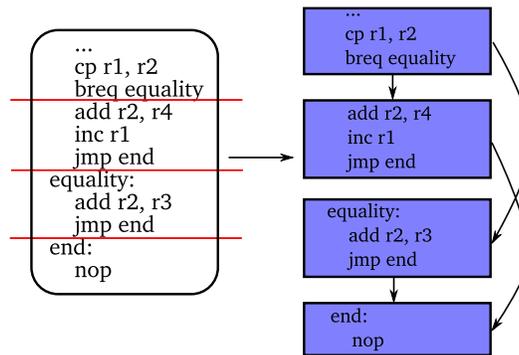


FIG. 24: Ajout des arêtes entre les blocs de base.

5.3.3 Graphe d'appel

Un graphe d'appel est un graphe orienté qui représente les relations d'appels entre les procédures d'un programme. Chaque nœud du graphe contient le nom d'une procédure et il y a une arête orientée partant du nœud appelant vers le nœud appelé. Un exemple de graphe d'appel est donné sur le schéma 25.

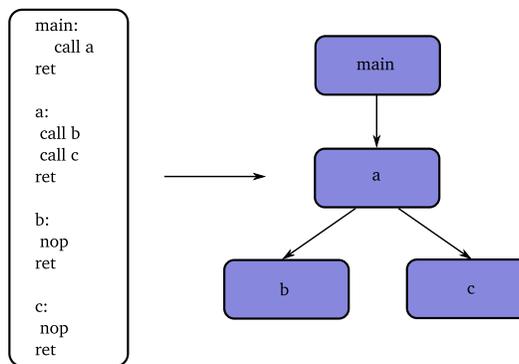


FIG. 25: Exemple de graphe d'appel.

Une procédure débute avec une étiquette et termine avec l'instruction `RET`. Cependant, comme toutes les étiquettes présentes dans le programme ne correspondent pas à une procédure, nous devons faire la différence entre une simple étiquette utilisée pour des sauts conditionnels (ou inconditionnels) et une étiquette identifiant une procédure. Nous identifions les procédures en regardant deux points :

1. Si une étiquette indique le début du programme à analyser ; cette information est généralement renseignée par l'utilisateur.

2. Si une étiquette correspond à la destination d'une instruction d'appel de procédure. Ceci passe par l'analyse de toutes les instructions d'appel présentes dans le programme.

5.3.4 Limitations à la reconstruction de graphes pour le langage AVR

Suivant les langages d'assemblage et les architectures sur lesquelles ils seront exécutés, la reconstruction du flot de contrôle d'un programme pose plusieurs problèmes liés à des phénomènes de cache, de pipeline ou tout simplement de résolution d'adresse de saut (voir [86]). Un problème que nous rencontrons avec le langage AVR concerne la résolution des adresses de branchements inconditionnels spécifiques. Le langage d'assemblage AVR utilise les instructions `jmp` et `call` pour dérouter le flot d'exécution du programme vers une adresse précise de la mémoire. Néanmoins, il existe deux variantes de ces instructions qui permettent d'adresser indirectement la mémoire via la concaténation du contenu des registres `r30` et `r31` (16 bits d'information); ces instructions se nomment `ijmp` et `icall`.

Ces instructions ont des répercussions sur la reconstruction du flot de contrôle car pour savoir à quelle adresse (i.e. étiquette ou procédure) le programme sera dérivé, il faut connaître les valeurs contenues dans les registres. Notre outil n'est pas en mesure de reconstruire les graphes d'appel et de flot de contrôle lorsqu'un programme contient une des deux instructions précédentes; il se contente de signaler à l'utilisateur qu'une telle instruction a été rencontrée. Nous tenons à souligner que la reconstruction de graphes de flot de contrôle est un problème très difficile et qu'aucun outil à l'heure actuelle n'est capable de le faire correctement pour les langages d'assemblage courants tels que le `x86` ou l'AVR.

Cependant, nous avons quelques idées pour déterminer la valeur des registres à un moment donné. L'approche retenue consisterait à mélanger simulation et analyse statique. Plus précisément, nous souhaiterions faire communiquer notre analyseur statique avec notre simulateur (voir chapitre 4). L'analyseur statique pourrait faire une première passe et noter tous les endroits qu'il ne sait pas résoudre. Ensuite, en faisant appel au simulateur, la valeur des registres pourrait être déterminée levant ainsi l'ambiguïté.

5.3.5 Exemple de résultats obtenus pour la représentation visuelle

Afin d'étayer nos propos, nous allons prendre pour exemple un petit programme écrit en langage d'assemblage AVR. Ce programme fait peu de choses, une procédure principale, appelée `main`, initialise un registre à la valeur quinze et fait appel à une sous-procédure, nommée `decr_loop` qui décrémente ce registre jusqu'à zéro. La procédure de décrémentation positionne un bit à un lorsque le registre atteint la valeur cinq. Ce programme est détaillé dans la figure 26.

```

decr_loop:
    cpi r16, $0          ; r16 == 0 ?
    breq end_loop      ;
    dec r16             ; r16 <- r16 - 1
    cpi r16, $5        ; r16 == 5 ?
    breq set_t         ;
    jmp decr_loop
set_t:
    set                 ; bit t <- 1
    jmp decr_loop
end_loop:
    nop
    ret                ; Fin de la sous-procédure
main:
    clt                 ; t <- 0
    eor r16, r16       ; r16 <- 0
    ori r16, $15      ; r16 <- 15
    rcall decr_loop
    nop

```

FIG. 26: Décrémentant d'un registre.

Bien que ce programme ne soit pas très long, comprendre sa structure ne reste pas une chose aisée car faire la différence entre des procédures et des étiquettes est une tâche ardue. En invoquant notre outil sur ce programme, nous obtenons son graphe de flot de contrôle et son graphe d'appel (cf. figure 27). Pour faciliter la lecture des graphes, nous avons défini le code couleur suivant :

- les arêtes de retour des boucles sont dessinées en rouge,
- les arêtes d'appel de procédures sont dessinées en vert,
- les arêtes de retour de procédure (i.e. l'instruction qui vient juste après le `ret`) sont affichées en orange.

De plus, la représentation d'un bloc de base est découpée en trois parties. Le coin en haut à gauche contient le nom d'une étiquette (si elle existe), le coin en haut à droite contient l'identifiant unique d'un bloc de base, la partie restante contient les instructions qui composent le bloc.

5.4 Évaluation de chemins

Le principal objectif de notre outil est de détecter des chemins non équilibrés qui pourraient apparaître au sein d'une procédure identifiée comme critique. Par chemins non équilibrés nous désignons les chemins dont les temps d'exécution, exprimés comme la somme des cycles d'horloge de chaque instruction rencontrée, sont

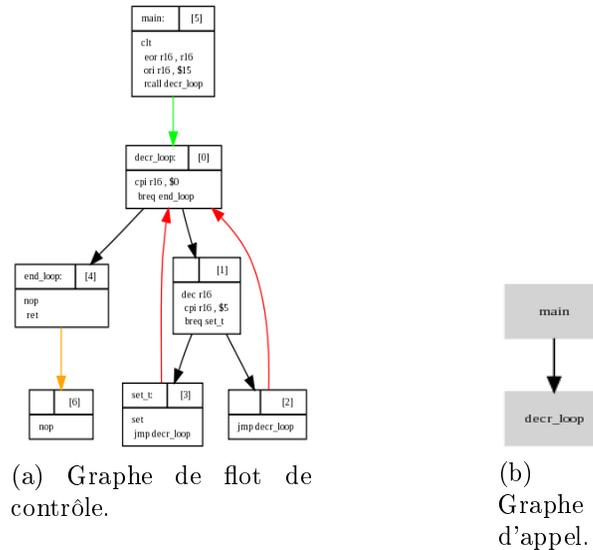


FIG. 27: Représentation des graphes orientés.

différents. Si de tels chemins peuvent être découverts dans des procédures manipulant des données sensibles, ils peuvent être exploités par une attaque analysant le temps d'exécution du programme (voir section 3.3.1). L'approche retenue pour notre outil se décompose en deux étapes. D'une part, il doit fournir un moyen d'identifier un chemin d'exécution particulier et d'autre part, il doit calculer automatiquement la somme des cycles d'horloge de chaque instruction présente sur le chemin, puis afficher le résultat.

Toutefois, il est important de noter que le calcul du nombre de cycles d'un chemin ne peut être réalisé de façon entièrement automatique. Ceci est dû, entre autre, à l'indécidabilité du nombre de fois que le programme passera dans une boucle. Pour cette raison, nous fournissons un outil interactif (i.e. semi-automatique) dans lequel l'utilisateur doit guider l'analyse afin de résoudre ce qui est indécidable.

5.4.1 Expressions régulières

Les expressions régulières nous paraissent bien adaptées pour décrire les chemins d'un automate. Or, le graphe de flot de contrôle peut être vu comme un automate où les états correspondent aux blocs de base et où les transitions entre blocs correspondent au flot d'exécution du programme. Cet automate reconnaît le langage des identifiants des blocs de base. Notre idée est d'obtenir de façon automatique l'expression régulière associée au graphe de flot de contrôle de telle sorte qu'un utilisateur n'ait plus qu'à l'instancier pour décrire un chemin d'intérêt.

Détaillons à présent cette construction sur l'exemple décrit sur la figure 27a. En supposant que le premier bloc de base porte le numéro 5 et que le dernier porte le numéro 6, tous les chemins d'exécution sont modélisés par l'expression régulière

minimale :

$$E := 5 ((0\ 1\ 3) + (0\ 1\ 2))^* 0\ 4\ 6$$

où * est le symbole Kleene étoile et + l'union de chemins.

5.4.2 Système d'équations de langage

Bien qu'une expression régulière soit parfaitement adaptée pour décrire les différents flots d'exécution d'un programme, la calculer à la main reste une tâche difficile et fastidieuse. Cependant, ce calcul peut être entièrement automatisé et c'est ce que propose notre outil. Nous souhaitons attirer l'attention du lecteur sur le fait que le calcul d'une expression régulière associée à un automate est un problème très classique (voir [89, 90, 91, 92]) et que nous n'avons pas amélioré une technique de calcul. Dans ce qui suit nous décrivons la méthode que nous avons appliquée.

Notre outil construit un système d'équations de langage prenant en compte les relations entre les blocs de base. Comme un bloc de base (i.e. un nœud du graphe de flot de contrôle) ne peut avoir au maximum que deux successeurs, toutes les équations sont de la forme :

$$L_i = aL_j + bL_k$$

où L_i, L_j, L_k sont les langages associés aux blocs de base B_i, B_j, B_k du graphe de flot de contrôle et a, b correspondent aux transitions entre le bloc B_i et les blocs B_j, B_k . Pour résoudre de tels systèmes d'équations de langage, nous utilisons le lemme d'Arden (voir [93]) qui affirme qu'une équation de la forme $X = AX + B$ avec $A \neq \epsilon$ où ϵ est le langage vide, admet une unique solution $X = A^*B$. À titre d'illustration, le système d'équations correspondant au graphe de flot de contrôle représenté sur la figure 27a est le suivant :

$$\begin{cases} L_0 = 0.L_4 + 0.L_1 \\ L_1 = 1.L_3 + 1.L_2 \\ L_2 = 2.L_0 \\ L_3 = 3.L_0 \\ L_4 = 4.L_6 \\ L_5 = 5.L_0 \\ L_6 = 6 \end{cases}$$

La figure 28 représente la solution du système d'équations ci-dessus. Cette solution est exprimée sous la forme d'une expression régulière automatiquement calculée par notre outil. Notons que notre outil ne produit pas l'expression régulière minimale

$$E = (((\ 5) . ((\ 0\ 1\ 3) + (\ 0\ 1\ 2))^*) . (\ 0\ 4\ 6))$$

FIG. 28: Expression régulière de tous les chemins du graphe de flot de contrôle du schéma 27a.

associée au programme car pour le moment, il ne calcule pas l'automate minimal

associé au graphe de flot de contrôle. Ceci signifie que pour certains programmes, l'expression régulière obtenue peut être un peu grande.

5.4.3 Instanciation d'expressions régulières

L'instanciation d'une expression régulière consiste à remplacer les symboles étoiles par la valeur correspondante au nombre d'itérations d'une boucle car dans les programmes que nous considérons, il n'y a pas de boucles infinies, le nombre de fois qu'une boucle doit itérer est connu à l'avance. Examinons le graphe de flot de contrôle représenté sur la figure 27a et supposons qu'un utilisateur souhaite examiner le chemin d'exécution où la boucle itère deux fois à travers le bloc B_2 , alors l'expression régulière associée est la suivante :

$$Exec := 5\ 0\ (0\ 1\ 2)\ (0\ 1\ 2)\ 4\ 6$$

La figure 29 illustre comment l'expression régulière ci-dessus se traduit dans notre outil. Les crochets identifient une boucle tandis que le corps de la boucle est repré-

```
[ Number of clock cycles ]
Please enter the path to examine:

> 5 0 [(0 1 2)*2] 4 6
--
Result: 28 Clock cycles
```

FIG. 29: Instanciation d'une expression régulière.

senté entre parenthèses et son nombre d'itérations se situe juste après l'étoile.

5.4.4 Méthode d'évaluation

Montrons maintenant comment ces fonctionnalités sont utilisées lors d'un processus d'évaluation. Notre but est de déterminer si le temps d'exécution d'un programme est corrélé aux données qu'il manipule et si cette corrélation peut être exploitée.

Pour nos besoins, nous considérons une procédure qui compare le contenu de deux tableaux. Elle permet de vérifier si un code PIN entré par un utilisateur correspond bien au code PIN stocké dans un périphérique. La figure 30 présente le détail de cette procédure écrite en langage C.

La procédure `verif_pin` compare deux tableaux de caractères de même taille et dès qu'un des éléments est différent elle renvoie un. Si les deux tableaux sont identiques, la procédure renvoie zéro.

Au vu du code C, nous pouvons penser que cette procédure est vulnérable à des attaques basées sur le temps d'exécution car nous nous rendons vite compte que

```

char
verif_pin( char* user_pin, char* true_pin, char taille ) {
    char idx;

    for ( idx = 0; idx < taille; idx ++ )
        if ( user_pin[ idx ] != true_pin[ idx ] )
            return '1';

    return '0';
}

```

FIG. 30: Procédure de comparaison de tableaux de caractères.

plus il y aura d'éléments identiques entre les deux tableaux et plus le programme fera d'itérations. En traduisant la procédure `verif_pin` dans le langage d'assemblage AVR grâce au compilateur `avr-gcc`, nous pourrions utiliser notre outil afin de valider (ou invalider) l'hypothèse de vulnérabilité concernant les *timing attacks*.

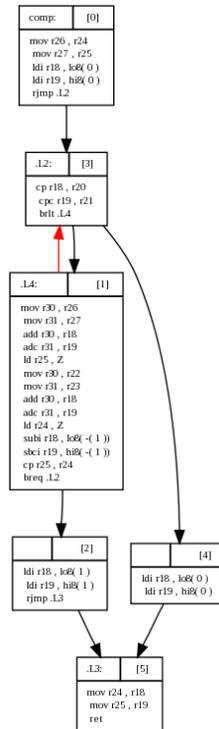
Analyse du graphe de flot de contrôle

L'approche consiste à invoquer notre outil sur la procédure écrite en langage d'assemblage et à examiner le graphe de flot de contrôle qu'il a généré (voir figure 31) afin de pouvoir examiner des chemins d'exécution distincts.

À partir du graphe de flot de contrôle de la figure 31, nous allons tenter de repérer les boucles ainsi que les codes de retour de la procédure. La détermination de ces informations nous permettra de mieux appréhender les chemins d'exécution.

Les boucles. La flèche rouge du graphe de flot de contrôle modélise l'arc de retour d'une boucle, ce qui nous permet d'en déduire que les blocs de base 1 et 3 correspondent respectivement aux conditions d'entrée dans la boucle et au corps de la procédure de comparaison.

Codes de retour. En examinant le bloc de base 2 (resp. 4), nous remarquons que ses instructions correspondent au chargement de la valeur un (resp. zéro) dans les registres `r18` et `r19`. Souvenons nous que dans le programme C décrivant la procédure de comparaison (voir figure 30), la valeur un (resp. zéro) était renvoyée quand les tableaux comparés étaient différents (resp. identiques). Cette analyse nous permet d'en déduire que le bloc 2 (resp. bloc 4) correspond à la définition du code de retour lorsque les tableaux sont différents (resp. identiques).

FIG. 31: Graphe de flot de contrôle de la procédure `verify_pin`.

Analyse des temps d'exécution

Au vu de tous ces éléments, nous sommes en mesure de définir les chemins d'exécution valides du programme en cours d'analyse et de les décrire en utilisant des expressions régulières. À partir du graphe de flot de contrôle de la figure 31 nous obtenons les trois expressions suivantes :

- $E_1 := 0\ 3\ 4\ 5$ (sortie de la boucle quand `idx >= taille`)
- $E_2 := 0\ (3\ 1) * 4\ 5$ (tableaux identiques)
- $E_3 := 0\ (3\ 1) * 2\ 5$ (tableaux différents)

Le chemin décrit par E_1 correspond à une sortie immédiate du programme sans manipulation des données contenues dans les tableaux ; ce qui fait qu'aucune corrélation ne pourra être établie entre temps d'exécution et données manipulées.

L'analyse du chemin décrit par E_2 nous donne seulement de l'information concernant le temps d'exécution de la procédure lorsque deux tableaux à comparer sont égaux. Bien que des données soient manipulées, elles sont identiques et ne permettent pas de déterminer si le temps d'exécution est corrélé ou non aux données manipulées. Nous devons donc nous intéresser aux exécutions qui comparent deux tableaux différents et c'est pour cela que nous allons concentrer nos efforts sur l'analyse de l'expression E_3 .

Commençons à présent l'analyse de l'expression E_3 . Pour une première approche, nous allons considérer que les deux premiers éléments comparés sont différents (i.e. `user_pin[0] != true_pin[0]`), ce qui nous mène à instancier l'expression régulière suivante :

$$E'_3 := 0\ 3\ 1\ 2\ 5$$

Notre outil nous informe que ce chemin requiert trente-trois cycles d'horloge pour s'exécuter.

Regardons à présent ce qui se passe lorsque nous considérons que les deux premiers éléments comparés sont identiques mais que les seconds sont différents (i.e. `user_pin[0] == true_pin[0]` et `user_pin[1] != true_pin[1]`). Pour cela nousinstancions l'expression régulière :

$$E''_3 := 0\ (3\ 1)\ (3\ 1)\ 2\ 5$$

Notre outil répond qu'il faut quarante-sept cycles d'horloge à ce chemin pour s'exécuter.

Nous pouvons d'ores et déjà constater qu'il existe une différence de quatorze cycles d'horloge entre les deux chemins d'exécution que nous avons analysé. Mais le plus important est sans doute d'avoir pu quantifier cette différence de temps. En effet, quatorze cycles d'exécution constituent une différence suffisamment importante pour être détectée, à l'aide d'un oscilloscope par exemple, et nous sommes maintenant convaincu que nous pouvons mettre en place une *timing attack* qui donnera des résultats.

Conclusion sur la méthode d'évaluation. Vérifier si un programme est vulnérable à des attaques basées sur le temps d'exécution ne peut être validé en se basant sur le code source écrit dans un langage de haut niveau et ceci pour deux raisons. D'une part, analyser le code source de haut niveau pour trouver des *timing attacks* fait seulement appel à l'intuition et non à des faits précis. D'autre part, même si un déséquilibre des temps d'exécution existe, ce n'est pas pour autant qu'il puisse être exploitable en pratique. Autrement dit, il y aurait une perte de temps à vouloir monter une *timing attack* sur un programme où la différence entre les exécutions ne pourra de toute façon pas être exploitée.

Nous venons de vérifier qu'une procédure était effectivement vulnérable à des attaques basées sur le temps d'exécution (i.e. que les données qu'elle manipule sont corrélées à son temps d'exécution). L'exemple présenté ici est un cas d'école, néanmoins il permet de bien illustrer toutes les étapes qui permettent de bien interpréter le graphe de flot de contrôle et de calculer des temps d'exécution sur des chemins d'intérêts. Le plus important réside dans le fait que nous avons pu valider qu'une attaque était possible. En effet, imaginons un instant que la différence enregistrée entre les deux chemins ne soit que d'un ou deux cycles. La procédure aurait alors présenté un déséquilibre mais qui n'aurait pas pu être exploité en pratique car la différence n'est pas assez significative. Le fait de valider en amont si une attaque est

possible ou non permet d'économiser plusieurs heures voire jours de tests inutiles et de se focaliser sur les véritables vulnérabilités du programme en cours d'analyse.

Enfin, précisons que dans les cas particuliers où des procédures ne possèdent pas de boucles, notre outil calcule et compare automatiquement le nombre de cycles d'horloge de tous les chemins d'exécution, épargnant ainsi énormément d'efforts à un humain.

5.5 Ergonomie

L'objectif de cette thèse consiste à fournir des outils ergonomiques destinés à être déployés afin de faciliter et d'automatiser les analyses de code source. Ainsi, dans le but de faciliter l'analyse de programmes écrits en langages d'assemblage, nous avons interfacé notre outil avec ECLIPSE qui est un environnement de développement intégré (voir [94]). Cet interfaçage nous permet d'exploiter les fonctionnalités d'Eclipse en terme d'affichage et de navigation dans le code. Nous souhaitons qu'Eclipse formate l'affichage du programme suivant des informations calculées par notre outil concernant le découpage en blocs de base et leur nombre de cycles d'horloge.

Pour ce faire, nous avons opté pour une architecture client-serveur où un greffon (*plug-in*) Eclipse (voir [95]) envoie des requêtes de calculs à notre outil. Ce dernier les traite et consigne les résultats dans un fichier au format XML. Le greffon Eclipse vient lire le fichier généré, analyse son contenu et modifie l'affichage du programme en conséquence. La figure 32 résume le mode de fonctionnement de notre architecture.

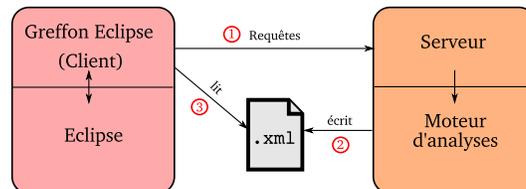


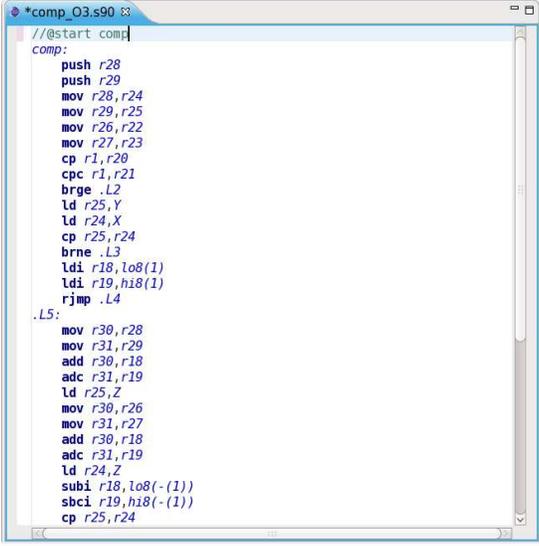
FIG. 32: Architecture client-serveur.

5.5.1 Blocs de base et nombre de cycles associés

Pour plus de lisibilité du code écrit en langage d'assemblage, nous avons défini une coloration syntaxique qui est appliquée automatiquement dès l'ouverture d'un fichier portant l'extension `.s90`. Les instructions et les données manipulées deviennent ainsi plus facile à lire (voir figure 33) ³.

Afin d'obtenir le découpage du programme en blocs de base, il est nécessaire d'indiquer à l'outil par quelle procédure l'analyse doit débiter. Cette information

³Pour des raisons de clarté, le code source de notre exemple a été épuré de toutes les pseudos-instructions générées par le compilateur.



```

*comp_O3.s90
//@start comp
comp:
  push r28
  push r29
  mov r28,r24
  mov r29,r25
  mov r26,r22
  mov r27,r23
  cp r1,r20
  cpc r1,r21
  brge .L2
  ld r25,Y
  ld r24,X
  cp r25,r24
  brne .L3
  ldi r18,lo8(1)
  ldi r19,hi8(1)
  rjmp .L4
.L2:
  mov r30,r28
  mov r31,r29
  add r30,r18
  adc r31,r19
  ld r25,Z
  mov r30,r26
  mov r31,r27
  add r30,r18
  adc r31,r19
  ld r24,Z
  subi r18,lo8(-(1))
  sbci r19,hi8(-(1))
  cp r25,r24

```

FIG. 33: Coloration syntaxique de code AVR dans Eclipse.

supplémentaire est insérée sous forme de commentaire dans le code source affiché par Eclipse. Plus exactement, il s'agit d'une balise spéciale nommée `//@start` suivie du nom de la procédure. Dans notre exemple, cette balise apparaît sur la première ligne du fichier.

Grâce à une combinaison de touches (e.g. `ctrl+r+s`), une requête est envoyée au serveur demandant le découpage en blocs de base et le calcul des cycles d'horloge de chaque instruction. Le greffon Eclipse modifie alors l'affichage du programme en fonction des données lues dans le fichier XML généré par notre outil. Le résultat obtenu est illustré par la figure 34.

En dessous de chaque bloc de base se trouve une balise `//@cycles` suivi du détail de la somme des cycles d'horloge de chaque instruction rencontrée dans ce même bloc. Il est important de noter qu'à la fin de certains blocs, deux résultats de somme peuvent apparaître. Ceci est dû aux instructions de branchement conditionnels qui ne prennent pas le même nombre de cycles suivant que la condition évaluée soit vraie ou fausse.

En cliquant sur le symbole \ominus qui se trouve à côté de chaque bloc, il est possible de masquer son affichage. Ceci permet à l'évaluateur de cibler son analyse sur une portion de code très précise sans être perturbé par la présence d'autres lignes de code.

```

*comp_O3.s90
//start comp
comp:
push r28
push r29
mov r28,r24
mov r29,r25
mov r26,r22
mov r27,r23
cp r1,r20
cpc r1,r21
brge .L2
//@cycle: 2 + 2 + 1 + 1 + 1 + 1 + 1 + 1 + 2 = 12
//@cycle: 2 + 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 11

ld r25,Y
ld r24,X
cp r25,r24
brne .L3
//@cycle: 2 + 2 + 1 + 2 = 7
//@cycle: 2 + 2 + 1 + 1 = 6

ldi r18,lo8(1)
ldi r19,hi8(1)
rjmp .L4
//@cycle: 1 + 1 + 2 = 4

.L5:
mov r30,r28
mov r31,r29
add r30,r18
adc r31,r19
ld r25,Z

```

FIG. 34: Affichage des blocs de base et de leurs nombres de cycles d'horloge.

5.5.2 Affichage du code inatteignable

Le code inatteignable (cf. 5.3.1), n'est pas gênant en soi car il ne joue pas de rôle dans le déroulement de l'exécution d'un programme. Cependant, il peut venir complexifier l'analyse de code car il faut alors démêler les portions de code qui correspondent à une exécution de celles qui n'y correspondent pas. Afin de répondre à ce problème, notre outil détecte le code inatteignable et indique dans le fichier XML l'endroit où il se trouve.

Dans notre exemple, nous avons commenté le branchement conditionnel qui se trouve à la fin du premier bloc de base (i.e. `brge .L2`) et nous l'avons remplacé par un saut vers une nouvelle étiquette (i.e. `jmp new_label`) que nous avons positionné quelques lignes plus bas (voir figure 35). La portion de code comprise entre l'instruction de saut et la nouvelle étiquette ne sera jamais exécutée. Elle apparaît comme étant barrée, facilitant l'interprétation du programme.

5.5.3 Visualisation d'impacts d'attaques par injection de fautes

Nous venons de voir que notre greffon Eclipse permet d'afficher le nombre de cycles associé à chaque bloc de base d'un programme mais aussi le code inatteignable rencontré lors de l'analyse. Cependant, l'aspect majeur de notre greffon réside dans le fait qu'il permet de modifier (i.e. rajouter ou commenter) des instructions du programme en cours d'analyse (comme l'illustre la figure 35). Autrement dit, il offre une façon de modéliser des attaques par injection de fautes.

```

*comp_O3.s90
//@start comp
.ccomp:
    push r28
    push r29
    mov r28,r24
    mov r29,r25
    mov r26,r22
    mov r27,r23
    cp r1,r20
    cpc r1,r21
    ;brge .L2
    jmp new_label
//@cycle: 2 + 2 + 1 + 1 + 1 + 1 + 1 + 1 + 3 = 13

.L2:
    ld r25,Y
    ld r24,X
    cp r25,r24
    brne .L3
new_label:
    ldi r18,lo8(1)
    ldi r19,hi8(1)
    rjmp .L4
//@cycle: 1 + 1 + 2 = 4

.L5:
    mov r30,r28
    mov r31,r29
    add r30,r18
    adc r31,r19
    ld r25,Z

```

FIG. 35: Marquage du code inatteignable.

Comme nous l'avons évoqué dans le chapitre 3, les injections de fautes consistent à perturber le déroulement du programme en cours d'exécution dans un microcontrôleur. En revanche, dans le cadre de l'analyse de codes sources écrits en langage d'assemblage, il reste difficile de prévoir et de visualiser quel sera précisément l'impact de ces perturbations sur le flot d'exécution du programme.

Cependant, avec notre greffon, il devient possible de simuler des attaques par injection de fautes et de visualiser de façon quasi instantanée les dégâts que peuvent causer ces perturbations. Ainsi, modéliser une attaque en faute revient à commenter des instructions ou à rajouter un saut incondtionnel dans le code source. En relançant l'analyse une fois les modifications effectuées, l'évaluateur est alors à même de vérifier quels sont les changements intervenus sur le flot de contrôle. Par exemple, l'apparition de code inatteignable constitue un point de départ pour l'évaluateur car si dans cette portion de code qui ne sera jamais exécutée se trouve l'appel à une procédure de sécurité alors une faille vient d'être mise au jour.

Nous devons préciser que cette approche qui permet de valider ou d'invalider des chemins d'attaques lors de l'évaluation logicielle constitue un gain de temps non négligeable. En effet, éliminer en amont tout un ensemble de chemins d'attaques permettra aux évaluateurs en charge des attaques physiques de concentrer leurs efforts sur des perturbations qui mèneront nécessairement à des résultats.

5.6 Conclusion

L'analyse manuelle de programmes écrits en langages d'assemblage reste une tâche ardue qui nécessite des connaissances poussées aussi bien de l'architecture sur laquelle sera exécuté le programme que sur la sémantique du langage d'assemblage. Nous avons présenté dans ce chapitre un outil destiné à aider l'analyse de programmes écrits en langage d'assemblage. Plus exactement, en automatisant en grande partie les tâches que doit réaliser un évaluateur, notre outil permet de vérifier si une implémentation est vulnérable aux *timing attacks*.

L'idée principale consiste à décrire des chemins du flot de contrôle d'un programme grâce à des expressions régulières. Une fois interprétées par notre outil, ces expressions régulières serviront à exprimer le temps exact d'exécution des chemins de flot de contrôle qu'elles modélisent. Il devient alors aisé de comparer le temps d'exécution de deux chemins différents et donc de vérifier si le programme est vulnérable à une attaque basée sur le temps d'exécution.

La valeur ajoutée de notre outil provient de deux facteurs. D'une part, nous avons amélioré et automatisé une méthode d'évaluation grâce à laquelle nous avons maintenant une mesure réelle de la résistance d'un programme face aux *timing attacks*. D'autre part, nous avons souhaité que notre outil soit ergonomique et pour cette raison, nous l'avons interfacé avec l'environnement de développement intégré Eclipse. Cet interfaçage permet d'obtenir une représentation originale du détail des temps d'exécution de chaque bloc de base rencontré dans un programme et surtout, il permet de modéliser des attaques par injection de fautes puis de visualiser l'impact de ces perturbations sur le flot de contrôle d'un programme fournissant ainsi une arme supplémentaire à un évaluateur.

Jusqu'à présent nous avons concentré nos travaux sur des aspects très bas niveaux comme la modélisation de microcontrôleurs et l'analyse de langages d'assemblage. Le prochain chapitre s'inscrit dans la suite logique des travaux précédents car nous allons aborder l'aide à l'analyse de programmes écrits dans un langage de haut niveau et plus spécifiquement le langage C.

Chapitre 6

Analyse de programmes écrits en langage de haut niveau

Nous avons vu que les langages d'assemblage, bien que très puissants, présentent plusieurs limitations. Tout d'abord ces langages sont difficiles à prendre en main car leur syntaxe est trop éloignée de celle des langages naturels. Ensuite, un langage d'assemblage est dédié à une architecture particulière. Ceci implique qu'une même application destinée à être exécutée sur différentes architectures doit être réécrite autant de fois qu'il y a de langages différents pour ces d'architectures. Enfin, le temps de développement d'applications écrites en langages d'assemblage est particulièrement long.

Pour pallier à ces limitations, des langages de haut niveau ont été conçus. Ils sont plus simples à manipuler que des langages de bas niveau car leur syntaxe est plus proche du langage naturel mais leur principale caractéristique réside dans le fait qu'ils font de fortes abstractions de l'architecture sur laquelle ils seront exécutés (i.e. de tels langages masquent le détail des opérations d'un microprocesseur : gestion des accès mémoires, etc.). En effet, ils disposent de structures de contrôle complexes, abstraient les données manipulées (i.e. on manipule des variables et non plus des registres) et proposent un typage des données qui permet d'éviter certains bogues. L'apparition de ces langages a constitué une véritable révolution dans le monde de l'informatique car ils offrent plus de souplesse au processus de programmation.

Néanmoins, si les programmes étaient devenus plus faciles à écrire, plus modulaires et plus simples à maintenir, ils présentaient toujours des problèmes de correction¹ ou de sécurité. En réalité, ces langages n'ont supprimé ni les erreurs de programmation (voir [96]) ni les vulnérabilités des programmes face à des attaques spécifiques (voir par exemple [34, 97, 98]). Pour pallier au problème des erreurs de programmation, voire des attaques, plusieurs techniques et outils ont été développés (voir par exemple [99, 100, 101, 102]).

¹Le mot correction est utilisé dans le sens d'être correct et non de corriger, c'est-à-dire que nous souhaitons avoir l'assurance que le comportement du programme est celui spécifié.

Bien qu'une carte à puce puisse par bien des manières être comparée à un ordinateur, elle n'est pas soumise aux mêmes menaces. Nous avons vu que les cartes à puce ne sont pas utilisées dans des environnements de confiance, ce qui les expose à quantité d'attaques dont les attaques physiques (i.e. où un accès physique à la carte est nécessaire, voir section 3.2.2). Ainsi, lors du processus d'évaluation de la sécurité d'une carte à puce, les évaluateurs en charge de l'analyse de code des logiciels sont tenus d'examiner quels pourraient être les impacts de telles attaques sur les politiques de sécurité et sur la robustesse des applications embarquées dans la carte.

6.1 Analyse sécuritaire de code source

Une analyse sécuritaire de code source consiste à chercher des faiblesses dans l'implémentation de programmes dans le but de contourner des mécanismes de sécurité ou de révéler des secrets (e.g. code PIN, clefs de chiffrements). Suivant les applications expertisées, les vulnérabilités ne sont pas les mêmes, par exemple la liste des vulnérabilités potentielles d'une carte bancaire ne sera pas la même que celle d'un passeport électronique. Au début de l'évaluation, l'expert génère donc une liste de points à vérifier correspondant à l'application dont il doit évaluer la sécurité. Plus exactement, il s'agit d'une liste de vulnérabilités génériques qui décrit des faiblesses d'implémentation de mécanismes ou politiques de sécurité. Par exemple, pour une vérification de code PIN, un point d'analyse pourrait être de s'assurer que le nombre d'essais courants a bien été incrémenté après la saisie d'un mauvais code. Pour couvrir ces points d'analyse, l'expert parcourt de façon manuelle le code source afin de parfaitement comprendre comment sont implémentés les mécanismes de sécurité.

Lors de son analyse, l'évaluateur est donc confronté à deux problèmes. D'une part, il doit comprendre très rapidement comment fonctionne un programme et pour ce faire il a besoin d'obtenir en un instant des renseignements sur les procédures ou variables rencontrées. D'autre part, il doit valider que les politiques de sécurité présentes dans la carte résistent à toute une gamme d'attaques (voir chapitre 3). Plus exactement, il doit repérer dans le code source des variables ou procédures dont la perturbation permettrait de contourner des mécanismes de sécurité et/ou d'accéder à des biens secrets. Le résultat de cette expertise est alors transmis aux personnes en charge des attaques physiques afin qu'elles concentrent tous leurs efforts sur les endroits identifiés comme critiques. Il doit donc disposer d'outils qui facilitent la compréhension du programme qu'il d'analyse et qui aident à vérifier que certaines propriétés de sécurité sont correctement implémentées en supposant que des injections de fautes sont possibles.

6.1.1 Fonctionnalités pour un outil d'aide à l'analyse de code

Examiner comment les évaluateurs procèdent à leur analyse de code nous a permis d'inventorier une liste de fonctionnalités pour un outil qui les aiderait dans

leur travail. Après avoir décortiqué puis trié ces fonctionnalités, nous nous sommes rendus compte qu'elles pouvaient être réparties en quatre grandes classes :

- la première classe concerne la recherche de vulnérabilités spécifiques qui peuvent compromettre la sécurité du code embarqué dans un périphérique,
- la seconde classe contient toutes les requêtes qui facilitent la navigation dans le programme en cours d'analyse. Ces requêtes permettent d'obtenir rapidement des informations sur les variables et les procédures ; nous avons nommé ces requêtes des assistants,
- la troisième classe permet d'analyser finement les relations entre les différentes procédures d'un programme en tenant compte des expressions conditionnelles rencontrées sur des chemins du graphe de flot de contrôle,
- la quatrième classe a trait à la vérification de propriétés de programmes écrits en langage C en utilisant des requêtes exprimées en logique temporelle. Elle fait l'objet d'un chapitre à part entière (voir chapitre 7).

6.2 Étude de l'existant

Comme nous venons de le voir, notre but est de faciliter la recherche de vulnérabilités dans des programmes écrits en langage C et destiné à être exécutés sur des cartes à puce. Néanmoins, au cours de la dernière décennie, différents outils similaires à celui que nous présentons ont vu le jour et peuvent être classés en deux catégories :

- d'une part les outils qui sont utilisés pour développer ou pour mieux comprendre la structure d'un programme. Parmi les plus célèbres, nous pouvons citer SOURCE-NAVIGATOR [103], SOURCE INSIGHT [104] ou encore UNDERSTAND [105]. Bien que ces outils soient très utilisés par la communauté des développeurs, ils n'ont pas été conçus pour facilement pouvoir rajouter des greffons et visualiser les résultats.
- d'autre part, les outils qui permettent de faire de l'analyse statique de code dans le but de rechercher des bogues. Nous pouvons citer FRAMA-C [106], POLYSPACE [107] et KLOCKWORK [108]. Bien que ces outils soient très puissants et trouvent beaucoup de failles de sûreté de fonctionnement dans les programmes qu'ils analysent, modifier leur interface graphique reste très difficile et surtout ils n'ont pas été conçus pour prendre en compte la possibilité d'attaques physiques qui peuvent modifier le flot d'exécution du programme.

Ainsi, la plate-forme que présentons fournit une alternative aux outils que nous venons de voir car elle permet d'automatiser une partie des tâches que doit réaliser un évaluateur en proposant des outils qui supposent que les attaques par injection de fautes sont possibles.

6.3 Approche retenue

Notre but consiste à fournir un outil qui assiste autant que possible l'évaluateur en charge de la partie logicielle dans sa démarche d'analyse de programmes écrits en langage C. Pour ce faire, nous avons mis au point un outil (voir [109]) qui est dédié à la recherche de vulnérabilités dans des programmes menacés par des attaques physiques. Avant de détailler l'architecture de notre outil, nous souhaitons présenter l'environnement CIL qui nous a servi de base pour mettre au point nos analyses.

6.4 CIL

Le langage C (voir [110]) est bien connu pour sa flexibilité quand il s'agit de traiter des constructions de bas niveau (e.g. manipulation de bits) mais il est aussi tristement célèbre pour être difficile à comprendre et à analyser, aussi bien par des humains que par des outils automatisés.

George Necula et al. ont mis au point CIL (*C Intermediate Language* voir [111, 112]) qui est un langage intermédiaire représentant un sous-ensemble très structuré du langage C associé à un ensemble d'outils permettant de faciliter l'analyse et la transformation de programmes écrits en C (*source-to-source transformation*). Comparé au langage C, CIL possède bien moins de constructions. Il décompose certaines constructions compliquées du langage C en de plus simples, permettant ainsi de travailler à un niveau plus bas que celui de l'arbre de syntaxe abstraite (*abstract syntax tree* en anglais). Cependant, CIL est de plus haut niveau que les langages intermédiaires typiques (e.g. code trois adresses) créés pour la compilation. Il comporte un nombre réduit de formes syntaxiques et conceptuelles ; par exemple toutes les constructions de boucles sont réduites à une forme simple, tous les corps des procédures se voient ajouter des déclarations explicites de retour (i.e. l'instruction `return`) et les sucres syntaxiques tel que « `->` » sont éliminés. Ainsi, nous avons une représentation qui facilite l'analyse et la manipulation de programmes écrits en langage C et qui est sous une forme qui ressemble au code source original. Notons aussi que l'environnement possède un *front-end* qui traduit dans CIL les programmes ANSI C mais aussi ceux utilisant les extensions de Microsoft ou de GNU C.

6.4.1 Analyses fournies par CIL

L'environnement CIL propose un ensemble d'outils destinés à faciliter l'analyse et la transformation de programmes écrits en C. Nous pouvons par exemple citer :

- la conversion de code C en C++,
- la transformation de code C en code trois adresses,
- l'élimination de code mort (i.e. élimination de calculs qui ne sont jamais utilisés),
- la recherche d'expressions disponibles (*available expressions*) qui consiste à déterminer pour chaque point du programme les expressions qui n'ont pas

- besoin d’être recalculées,
- la possibilité d’insérer le corps d’une procédure à chaque fois où cette procédure est appelée dans le programme (i.e. *function inliner*).

Cette liste n’est pas exhaustive mais donne une idée plus précise des possibilités fournies par CIL et illustre bien qu’il s’agit d’un environnement dédié à l’analyse de programmes.

6.4.2 CIL et graphe de flot de contrôle

CIL fournit aussi bien des informations de haut niveau sur la structure d’un programme que des informations de bas niveau concernant son flot de contrôle. En effet, le flot de contrôle d’un programme est capturé par une structure récursive d’assertions (*statements* en anglais) où chaque assertion est annotée avec des informations concernant son successeur et son prédécesseur dans le flot de contrôle (voir figure 36). Ainsi, cette représentation d’un programme peut aussi bien être utilisée dans

```

stmt ::= Instr(instr list)           | Return(exp option)
      | Goto(stmt)                   | Break
      | Continue                     | If(exp, stmt list, stmt list)
      | Switch(exp, stmt list, stmt list) | Loop(stmt list)

```

FIG. 36: Syntaxe des déclarations de CIL.

des routines qui nécessitent un arbre de syntaxe abstraite (e.g. analyses de type) que dans des routines qui requièrent un graphe de flot de contrôle (e.g. analyses du flot de données).

6.4.3 Conclusion sur CIL

CIL possède une architecture minimale qui tente de transformer les constructions du langage C en de plus petites mais avec une interprétation précise. De plus, CIL reste très près des constructions de haut niveau du code source de telle sorte que le résultat des transformations d’un programme soit proche du code original.

Nous noterons que plusieurs outils utilisent CIL comme un moyen d’accéder à l’arbre de syntaxe abstraite et/ou au graphe de flot de contrôle de programmes écrits en C. Parmi ces outils, nous pouvons citer FRAMA-C (voir [106]) qui est un outil d’analyse de programmes ou bien encore COMPCERT (voir [113]) qui est un compilateur C prouvé en COQ.

6.5 Architecture

Notre architecture est similaire à celle retenue pour faire communiquer notre outil d'analyse de programmes écrits en langage d'assemblage AVR avec un environnement de développement intégré (voir section 5). En effet, là encore, nous avons opté pour une architecture client-serveur où, côté serveur, nous disposons d'un moteur d'analyse qui utilise de façon intensive l'environnement CIL (voir section 6.4). Nous noterons que le serveur et le moteur d'analyse ont été écrits en OBJECTIVE CAML (voir par exemple [67, 114, 68]) car c'est un langage qui se prête particulièrement bien à l'analyse de programme mais aussi parce que CIL est développé dans ce langage.

Côté client, nous avons besoin d'une interface graphique homogène permettant de naviguer dans le code source d'applications, dont la maintenance soit aisée et dont les fonctionnalités peuvent être facilement étendues suivant nos besoins. Il s'est avéré que l'environnement de développement Eclipse se conformait en tous points à nos spécifications, nous permettant ainsi d'obtenir un client léger et facilement ajustable. Plus précisément, le client est un greffon Eclipse (voir [95]) qui permet d'une part de faciliter la navigation dans de grands programmes séparés en plusieurs fichiers et d'autre part de présenter de façon claire les résultats produits par notre moteur d'analyse.

Ce greffon a été entièrement écrit en JAVA (voir [115]), qui est le langage dans lequel Eclipse est développé. La figure 37 présente l'organisation de notre architecture client-serveur.

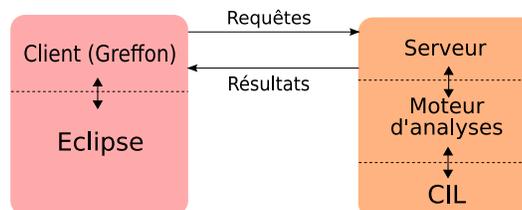


FIG. 37: Architecture client-serveur.

Ainsi, notre serveur accepte des requêtes du client et les envoie vers le moteur d'analyse. Le moteur d'analyse utilise alors CIL afin de construire des structures de données spécifiques (e.g. graphe de flot de contrôle, graphe d'appel de procédures) et les analyse dans le but d'exhiber des propriétés correspondant à la requête exprimée par le client.

6.6 Recherche de vulnérabilités

Toute évaluation sécuritaire de programmes embarqués passe par une analyse du code source (de bas niveau ou de haut niveau) dans le but de rechercher des politiques de sécurité mal implémentées ou des vulnérabilités. Une vulnérabilité dans

un programme informatique est une erreur dans son implémentation ou sa conception qui peut être utilisée pour altérer le comportement normal du programme en question. Exploitée avec des intentions malicieuses, une vulnérabilité peut avoir des conséquences désastreuses comme par exemple obtenir une élévation des privilèges de l'utilisateur ou bien encore révéler des clefs de chiffrement. Afin de renforcer la sécurité des programmes embarqués dans les cartes à puce, nous nous focalisons sur deux types de vulnérabilités. D'une part, nous cherchons à identifier des mauvaises pratiques concernant les déclarations de tableau. Cette vulnérabilité, si elle est exploitée peut permettre d'écraser des données dans l'espace mémoire qui est réservé lors de l'appel d'une procédure. D'autre part, nous recherchons des valeurs particulières de constantes utilisées dans les expressions conditionnelles. Nous verrons dans ce qui suit que l'identification de ces valeurs couplée à des attaques par injection de fautes peuvent mener à contourner des mécanismes de sécurité.

Les deux types de vulnérabilités que nous traitons ici ne sont que des exemples de réalisation d'interfaçage pour la première classe de fonctionnalités identifiées dans la partie 6.1.1. En effet, ces analyses sont plus une preuve de concept, dans le sens où nous désirons montrer que s'il existe des vulnérabilités logicielles dans les programmes embarqués, nous disposons d'une plate-forme dans laquelle il est possible de venir greffer des analyses capables de les détecter.

6.6.1 Les déclarations dangereuses de tableaux

Dans le langage C, il existe principalement deux façons d'allouer des tableaux dont la taille n'est pas connue pendant l'étape de compilation mais durant l'exécution d'un programme. La première façon de procéder à des allocations de mémoire dynamique passe par l'utilisation de procédures spécifiques (e.g. `void *malloc(size_t size)` où `size` correspond à la taille des données à allouer). Ces procédures réservent alors de l'espace mémoire sur le tas (*heap* en anglais) et renvoient un pointeur vers le premier élément alloué.

L'autre façon, moins élégante, consiste à utiliser la notion de tableau de taille variable (*variable length array*). Il s'agit de tableaux dont la taille est passée en paramètre d'une procédure; un exemple d'une telle construction est donné sur la figure 38. Le tableau de taille variable est alors considéré comme étant une variable

```
int
ma_fonction( int len ) {
    int T[ len ]; // Vulnérabilité
    // Reste de la fonction
    // ...
    return 0;
}
```

FIG. 38: Tableau de taille variable.

automatique² et par exemple le compilateur `gcc` l'alloue dans la pile de la procédure en cours d'exécution. Ce genre de déclaration peut être très dangereux si un attaquant parvient à changer le paramètre de la procédure en une valeur négative : l'allocation fonctionnera normalement mais un débordement de tableau se produira.

Un débordement de tableau (le fameux *buffer overflow*) se produit lorsqu'un programme essaye de stocker des données au-delà de l'espace mémoire réservé pour le tableau. Les données supplémentaires sont alors écrites dans les emplacements mémoires adjacents au tableau et peuvent par la même occasion causer de sérieux dommages au programme. Par exemple, l'adresse de retour de procédures peut être écrasée et des mécanismes de sécurité contournés.

Si certains compilateurs comme `gcc` sont très permissifs et autorisent ces constructions sans émettre d'avertissement, qu'en est-il de compilateurs plus spécifiques dédiés aux microcontrôleurs ? Pour tenter d'apporter des éléments de réponse, nous avons testé la compilation du programme présenté sur la figure 38 avec trois compilateurs commerciaux, à savoir : CODE WARRIOR [116], CALMSHINE [117] et KEIL [118]. Comme l'illustre le tableau 8, les compilateurs commerciaux que nous avons

Compilateurs	Message d'erreur
Code Warrior	<i>Illegal index value</i>
CalmShine	<i>Integer expression must be constant</i>
Keil	<i>Non constant case/dim expression</i>

TAB. 8: Détection des tableaux de taille variable.

testé détectent une erreur et n'autorisent pas la compilation. Néanmoins, nous n'avons pas testé l'intégralité des compilateurs disponibles sur le marché et il se peut que certains soient plus permissifs et n'émettent pas d'avertissement s'ils rencontrent ce type de construction. De plus, l'outil que nous développons est pour le moment utilisé sur des programmes destinés aux cartes à puce mais dans l'avenir il devra aussi être utilisé pour analyser des programmes destinés à des ordinateurs classiques où ce genre de constructions peut apparaître. Ainsi, il se peut que certains programmeurs peu expérimentés voient dans les tableaux de taille variable une façon plus simple de gérer la mémoire et introduisent par la même occasion une vulnérabilité et pour cette raison nous fournissons une méthode pour les détecter. La méthode que nous avons utilisée pour découvrir ce type de vulnérabilités est très simple et elle est décrite par l'algorithme 1.

6.6.2 Les valeurs faibles

Quand un évaluateur procède à une analyse de code de carte à puce, il est souvent amené à analyser le type d'expressions conditionnelles décrit sur la figure

²Variable qui est allouée automatiquement quand le flot d'exécution du programme atteint la portée de cette variable.

Algorithme 1 Recherche de déclarations dangereuses de tableaux.

ENTRÉES: Un programme P valide et préprocessé

Pour chaque fonction f du programme P **faire**
 $fp = \{ \text{paramètres formels de } f \}$
Pour chaque déclaration D de tableau **faire**
si $\exists e \in fp$ utilisé par D **alors**
 Vulnérabilité découverte
fin si
fin pour
fin pour

39. `CST_MACRO` est une constante spécifique définie avec la commande `#define` du

```
if ( ma_variable != CST_MACRO )
    verifier_pin();
else
    pas_de_verification();
```

FIG. 39: Expression conditionnelle faible.

préprocesseur C et `ma_variable` est une variable contenant une valeur numérique entière (i.e. de type `char` ou `int`).

Dans un contexte normal d'utilisation, ce genre de construction ne présente pas de problème de sécurité. Néanmoins, une faille de sécurité peut apparaître si nous considérons que le programme peut être la cible de perturbations physiques. Les attaques visant à perturber le programme et à modifier son flot de contrôle sont souvent réalisées via des rayonnements (voir section 3.2.2). Ici, nous considérons que le programme embarqué dans la carte sera perturbé par une émission laser. Plus précisément, une attaque par émission laser permet d'attaquer les registres du microprocesseur et d'altérer les valeurs qu'ils contiennent. En effet, l'énergie émise par le laser suffit à modifier l'état des bits d'un registre. Nous devons préciser que lorsque nous disons modifier l'état des bits, cela ne signifie pas positionner une valeur particulière dans le registre ; le laser a pour effet de faire passer tous les bits du registre à l'état zéro ou à l'état un.

Cette propriété mène tout droit à la notion de *valeurs faibles*. Nous considérons qu'une valeur est faible lorsque dans sa représentation binaire tous les bits sont positionnés à zéro ou à un.

De l'importance de l'architecture

Bien que nous analysions du code source de haut niveau, l'architecture sur laquelle il sera utilisé joue un rôle capital dans la recherche des valeurs faibles. En effet, suivant le type d'architecture et la taille des registres considérés, il se peut qu'une

valeur décimale soit faible ou non. Par exemple, considérons la valeur 0xFF00. Sur une architecture où la taille des registres est de seize bits, la définition de valeur faible ne peut pas s'appliquer car il faut un seul registre pour stocker cette valeur et les bits du registre ne sont pas tous positionnés à zéro ou à un. En revanche, sur une architecture comportant des registres de huit bits, cette valeur décimale ne pourra être stockée sur un seul registre, ce qui implique de la stocker sur deux registres de huit bits. Ainsi, la comparaison se fera en deux temps, premièrement en comparant les octets de poids faible (ici 0x00) puis en comparant les octets de poids fort (ici 0xFF) de la constante avec ceux de la variable. Dans ce cas précis nous obtenons deux valeurs faibles, ce qui induit une vulnérabilité.

Basée sur les définitions précédentes, nous proposons une méthode de recherche de valeurs faibles décrite par l'algorithme 2.

Algorithme 2 Recherche de valeurs faibles.

ENTRÉES: Un programme P valide et préprocessé
 La taille T des registres de travail du microprocesseur
Pour chaque fonction f du programme P **faire**
 Pour chaque expression conditionnelle $cd \in f$ **faire**
 Pour chaque constante $c \in cd$ **faire**
 $\{REGS\}$ = séparer c en mots binaires de tailles T
 Pour chaque mot $w \in REGS$ **faire**
 si w est faible **alors**
 Émettre un warning
 fin si
 fin pour
 fin pour
fin pour

Vulnérabilités et valeurs faibles. Une attaque par valeur faible résulte de la conjonction de divers éléments. Tout d'abord, le code source doit contenir des expressions conditionnelles dans lesquelles la valeur d'une variable sera comparée à celle d'une constante. Ensuite, il faut que la définition de valeur faible puisse s'appliquer à la valeur de cette constante. Enfin, l'attaque par émission laser doit venir modifier le contenu de la variable pour la rendre égale (ou différente) de la valeur de la constante et ceci avant la comparaison.

Lorsque tous ces éléments sont réunis, il est alors possible de modifier le flot de contrôle du programme et de contourner des mécanismes de sécurité. Par exemple, en supposant que sur la figure 39 la constante `CST_MACRO` est définie comme équivalente à 0xFF et qu'un attaquant modifie le registre contenant la valeur de la variable `ma_variable`, alors le flot de contrôle du programme ne passera pas par le

chemin contenant l'appel à la fonction de vérification du PIN, exploitant ainsi une vulnérabilité.

6.7 Les assistants de navigation

Un assistant est un objet graphique destiné à faciliter la navigation et l'analyse du code source d'un programme. Nous avons développés plusieurs assistants qui peuvent être classés en deux catégories que nous décrivons ci-après.

6.7.1 Variables et procédures

La première catégorie d'assistants regroupe des fonctionnalités qui permettent d'obtenir des informations sur les procédures ou les variables et de les localiser dans le code source. Ces fonctionnalités concernent :

- les variables locales. Cet assistant permet de lister les variables qui sont déclarées localement dans une procédure,
- l'affectation de variables. Cet assistant liste toutes les variables (i.e. locales et globales) qui sont modifiées au sein d'une procédure donnée,
- un graphe d'appel statique *inverse*. Cet assistant liste pour une procédure donnée toutes les procédures qui l'utilisent. Dans ce qui suit nous détaillons ce concept.

Graphe d'appel *inverse*. Un graphe d'appel statique est un graphe orienté qui représente toutes les relations d'appel entre les procédures d'un programme. Il s'agit d'une sur-approximation de l'ensemble des exécutions d'un programme où chaque nœud représente le nom d'une procédure et chaque arête modélise une relation d'appel. Par exemple, sur la figure 40, le graphe d'appel nous indique que la procédure *A* appelle les procédures *B* et *C* et que la procédure *D* appelle elle aussi la procédure *C*.

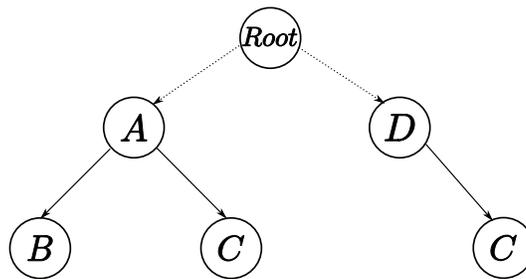


FIG. 40: Graphe d'appel.

Les graphes d'appel font partie des outils de base pour comprendre la structure d'un programme (e.g. CIL est capable d'exhiber un graphe d'appel). Une des applications des graphes d'appels consiste par exemple à trouver des procédures qui ne

sont jamais appelées.

Néanmoins, dans le processus d'analyse de code, un évaluateur éprouve souvent le besoin d'obtenir la relation inverse : c'est-à-dire connaître pour une procédure donnée, quelles sont toutes les procédures qui l'utilisent. Calculer ces relations entre procédures passe par la construction d'un graphe d'appel statique *inverse*. Il s'agit d'un graphe orienté construit à partir des informations calculées par un graphe d'appel statique ; ce qui signifie que ce graphe aussi sur-approxime les relations d'appel entre les procédures.

Par exemple, le graphe de la figure 40 nous permet de déduire que la procédure *C* est appelée par les procédures *A* et *D*, ce qui nous donne un graphe d'appel statique *inverse* représenté sur la figure 41.

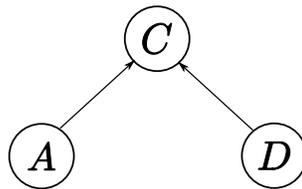


FIG. 41: Graphe d'appel inverse.

Cette notion de graphe d'appel inverse est importante pour un évaluateur. Imaginons que la procédure *C* présente une vulnérabilité, alors grâce au graphe d'appel inverse, nous en déduisons que cela impact directement les procédures *A* et *D*, ce qui nous fournit dans l'absolu deux chemins d'attaques possibles, qui resteront à valider ou invalider.

6.7.2 Les rapports

Cet ensemble contient des fonctionnalités qui permettent de mémoriser ce qui a été analysé par l'évaluateur et d'exporter automatiquement le résultat de l'analyse en cours dans des rapports. Quand une liste de vulnérabilités apparaît, comme par exemple juste après avoir lancé l'analyse concernant les valeurs faibles, l'évaluateur doit inspecter manuellement chaque élément de la liste. Pour simplifier la tâche d'évaluation, chaque fois qu'un élément de la liste a été vérifié (voir figure 42), l'évaluateur peut spécifier s'il s'agit :

- ✓ d'une fausse alerte (i.e il n'y a pas de vulnérabilité),
- ● d'une vulnérabilité et une attaque peut être menée,
- ▲ de quelque chose de suspicieux qui requiert une expertise ultérieure (et éventuellement l'aide d'un autre évaluateur)

Une fois l'évaluation terminée, il est possible de générer un rapport au format *HTML* afin de garder une trace des vulnérabilités trouvées et/ou de ce qui reste à analyser.

Function	Hexadecim	Decimal	File	Line
✓ third	0x2	2	test.c	23
▲ second	0x80	128	test.c	48
● weak_values	0x0	0	test.c	68
● weak_values	0x1	1	test.c	72
✓ main	0x20	32	test.c	155

FIG. 42: Vérification des éléments générés par une analyse.

6.8 Graphes d'appel et précision

Le calcul d'un graphe d'appel qui soit exact (i.e. qui décrit seulement les exécutions réelles d'un programme) est un problème récurrent dans l'analyse de programmes. Malheureusement, il s'agit d'un problème indécidable et les solutions existantes consistent à fournir des approximations qui sont générées au moyen d'analyses dynamiques ou statiques.

Un graphe d'appel dynamique correspond à *l'enregistrement* d'une exécution d'un programme réalisé par exemple à l'aide d'un outil de profilage (e.g. `gprof` voir [119]). Un tel graphe est certes exact (i.e. pas de sur-approximation de flot de contrôle) mais seulement pour une exécution particulière d'un programme. À l'inverse, les graphes d'appel calculés de façon statique considèrent toutes les exécutions d'un programme et sur-approximent donc les relations entre les procédures. Ceci implique que certains appels de procédures ne pourront jamais être atteints car ils ne correspondent pas à des exécutions réelles d'un programme.

Dans ce qui suit, nous présentons une méthode semi-automatique de raffinement de graphe d'appel qui se situe à la frontière de la génération dynamique et statique.

6.8.1 Navigation dans le flot de contrôle

Une des sources d'indécidabilité qui empêche le calcul d'un graphe d'appel exact à trait à l'évaluation d'expressions conditionnelles. Le fait de ne pas pouvoir résoudre, pour chaque expression conditionnelle rencontrée, si le programme empruntera la branche vraie ou la branche fausse du test, oblige à considérer les deux et donc à obtenir des sur-approximations. En revanche, dans bien des cas, un humain est capable de déterminer par quelle branche du test un programme passera.

Aussi, nous proposons un outil graphique qui offre la possibilité à un utilisateur de positionner le résultat de chaque expression conditionnelle rencontrée dans une procédure à trois valeurs possibles : **True**, **False** ou **Unknown** quand l'évaluateur ne sait pas déterminer quelle branche doit emprunter le programme.

Quand le résultat d'une expression conditionnelle est positionné à **True** ou à **False**, seulement un chemin (i.e. correspondant à la branche vraie ou fausse) est généré dans le graphe d'appel. En outre, quand **Unknown** est sélectionné, nous considérons l'ensemble des chemins qui partent de cette expression conditionnelle, ce qui constitue une sur-approximation (voir figure 43).

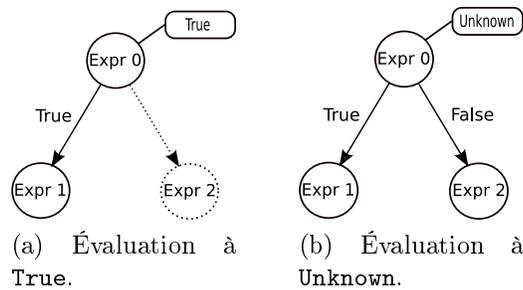


FIG. 43: Graphes d'appel obtenus par navigation dans le flot de contrôle.

Ainsi, nous offrons à l'évaluateur un moyen de naviguer dans le flot de contrôle, en créant des graphes d'appels plus précis.

6.8.2 Application à la sécurité

Un évaluateur de la sécurité des logiciels embarqués dans une carte à puce se place toujours du point de vue d'un attaquant qui est capable de perturber le programme à n'importe quel moment (i.e. grâce aux attaques physiques). Autrement dit, l'évaluateur se pose constamment la question de savoir quel impact la perturbation d'une variable aura sur le flot de contrôle d'un programme et sur l'enchaînement d'appel de procédures (e.g. si la variable a voit sa valeur modifiée avant le test t est-ce que la procédure p sera toujours appelée?). Grâce au système de navigation que nous proposons, il devient plus facile d'évaluer ces impacts et de déterminer si des procédures de sécurité (e.g. vérification de code PIN, somme de contrôle, etc.) seront toujours appelées ou non. De plus, notre système permet aussi de traquer des failles en vérifiant si des politiques de sécurité sont correctement implémentées ou si un programme a un comportement attendu.

Dans ce qui suit nous donnons un exemple de procédure comportant une erreur de programmation que la navigation dans le flot de contrôle permet de découvrir.

Exemple

Dans cet exemple, nous considérons la procédure nommée `switch_case` dont le détail est présenté sur la fenêtre numérotée ① de la figure 44.

Cette procédure illustre un mécanisme de modification de code PIN qui a été mal implémenté. Elle commence par récupérer les droits de l'utilisateur via l'appel à `get_conditions_access` et ensuite vérifie si ses conditions autorisent la modification du code PIN en invoquant `modify_pin`. Si les conditions ne sont pas suffisantes, une erreur est renvoyée au travers la procédure `set_status_error`. La fenêtre numérotée ② correspond au graphe d'appel et regroupe l'ensemble des procédures appelées (en respectant leur ordre d'appel).

Imaginons à présent qu'un évaluateur souhaite vérifier si cette procédure est correctement implémentée d'un point de vue sécuritaire. Sa démarche consiste à

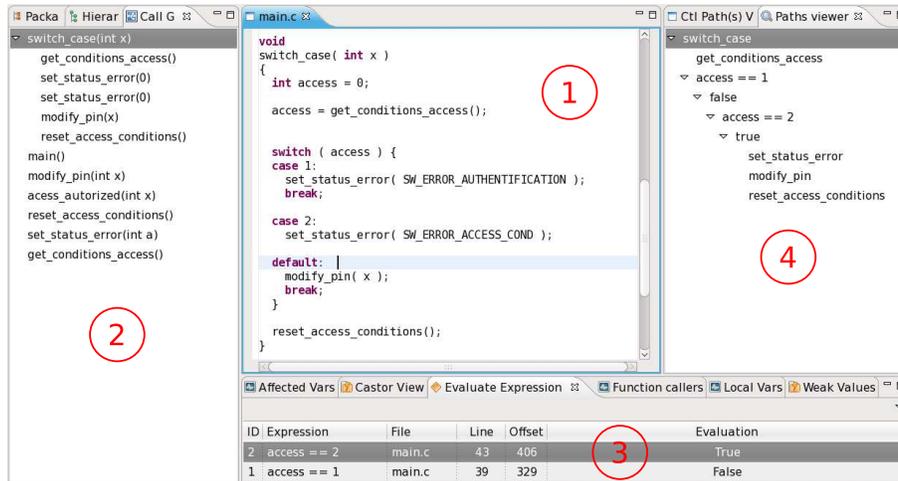


FIG. 44: Exemple d'une analyse intra-procédurale de graphe d'appel.

regarder ce qui se passe lorsque les conditions d'accès ne sont pas remplies (i.e. `case 1` : et `case 2` :). Dans le cas présent, nous considérons que l'évaluateur fixe les valeurs des expressions conditionnelles en considérant que `x == 1` est faux mais que `x == 2` est vrai comme présenté sur la fenêtre numérotée ③.

Une fois ces valeurs fixées, l'évaluateur demande la génération du graphe d'appel qui s'affiche dans la fenêtre numérotée ④. En étudiant de près les procédures qui sont appelées (et leur ordre d'appel), nous remarquons qu'entre les procédures `set_status_error` et `reset_access_conditions` s'est glissé un appel à la procédure `modify_pin` qui normalement ne doit pas apparaître en cas d'erreur.

En examinant de plus près le code source, nous nous rendons compte que le programmeur a oublié de mettre l'instruction `break` juste avant l'instruction `default`. Cet oubli a pour effet de pas dérouter le flot de contrôle en dehors de la structure de contrôle `switch` et de continuer l'exécution avec les instructions qui suivent.

Ainsi, grâce à la navigation dans le flot de contrôle, l'évaluateur vient de mettre au jour une faille de sécurité : il est possible de modifier le code PIN même sans remplir toutes les conditions d'accès.

6.9 Conclusion

Dans ce chapitre nous avons présenté une plate-forme qui au travers d'une interface graphique homogène permet d'automatiser une partie des tâches que doit réaliser un évaluateur durant son analyse de code.

Tout d'abord, grâce à ses fonctionnalités graphiques, notre outil facilite la navigation dans le code en offrant des informations sur les variables et procédures en cours d'analyse. Ensuite, il illustre le fait qu'il peut aussi servir de plate-forme dans laquelle il est possible de venir greffer des analyses recherchant automatiquement

des vulnérabilités dans un programme. Enfin, il propose une solution qui permet à la fois de mesurer les impacts d'une perturbation physique mais aussi de vérifier des politiques de sécurité concernant l'enchaînement d'appels de procédures. Le fait de pouvoir positionner la valeur de certaines expressions conditionnelles permet d'obtenir un graphe d'appel plus précis qui évite des sur-approximations trop importantes.

Nous précisons que le fait de ne pas avoir développé un programme monolithique intégrant à la fois le moteur d'analyse et l'affichage nous permet d'obtenir un outil plus flexible. En effet, ceci nous a permis de déporter l'interface graphique dans le logiciel Eclipse ce qui constitue un point important car les évaluateurs du pôle logiciel utilise quotidiennement cet environnement. Notre outil s'intègre donc au mieux dans leur processus d'évaluation.

Le chapitre suivant présente un aspect de l'outil qui n'a pas été abordé ici à savoir une manière de vérifier des propriétés sur un programme en utilisant des requêtes exprimées en logique temporelle.

Chapitre 7

Model checking sur des programmes écrits en C

Lors d'une analyse sécuritaire de code embarqué dans une carte à puce, il est courant que l'évaluateur ait besoin de vérifier si certaines politiques de sécurité sont correctement implémentées. Vérifier une politique de sécurité nécessite de s'assurer qu'un enchaînement d'événements soit correct, ce qui implique d'être en mesure d'exprimer des propriétés sur des variables, des procédures voire des successions d'appels de procédures. Par exemple, un évaluateur peut souhaiter s'assurer qu'un programme qui manipule des biens sensibles (e.g. une clef de chiffrement) efface les zones mémoires dans lesquelles les biens ont été manipulés.

Pour vérifier ce type de propriétés, il faut un formalisme qui d'une part permette d'exprimer des propriétés sur les chemins d'exécution d'un programme mais qui en plus soit capable d'exprimer une notion d'ordonnancement dans le temps (e.g. il n'existe pas de chemin tel que l'événement *A* intervient après l'événement *B*). Par ailleurs, nous souhaitons de la souplesse d'utilisation, c'est-à-dire qu'en raison du temps imparti pour des évaluations de code source, il n'est pas raisonnable d'envisager qu'un évaluateur modélise (e.g. via un automate) entièrement le comportement attendu du programme qu'il analyse. Notre objectif est plutôt de fournir un outil facile à manipuler qui via des requêtes permette d'interroger le code sur ses différentes exécutions.

7.1 Approche retenue

Nous voulons développer un outil qui soit en mesure d'exprimer des propriétés sur le comportement d'un programme écrit en langage C (i.e. ses exécutions) au travers de requêtes définies par l'évaluateur. Notre but n'est pas de se baser sur les exécutions réelles du programme ou de décrire le comportement attendu d'un programme (via un automate ou un autre formalisme) car :

- se baser sur les exécutions réelles revient à avoir toutes les entrées possibles du programme et pour chaque entrée calculer son exécution, ce qui est impossible

- en pratique,
- fournir un modèle décrivant le comportement d'un programme est un processus long, réalisé à la main (i.e. par l'évaluateur) et qui nécessite une connaissance poussée du programme (i.e. nom des procédures et comportement exact associé).

7.1.1 Exemples de requêtes

Nous présentons plus en détail quelques exemples de propriétés de sécurité que nous souhaitons vérifier et quels sont les types de requêtes dont nous avons besoin.

Appel au *pare-feu*

Dans les plates-formes natives Java Card™(voir [120]), une partie de la sécurité est assurée par un mécanisme complexe, qui se nomme *pare-feu* ou *firewall* permettant d'assurer la protection des données d'une application vis-à-vis des autres applications. Ainsi, à partir d'une procédure identifiée comme critique, toutes les exécutions doivent contenir un appel au *firewall*.

Effacement du buffer de clef

Lorsqu'une fonction manipule des biens sensibles, il est souhaitable de s'assurer que les endroits de la mémoire où les données sensibles ont été stockées ont été effacés car il existe un risque qu'elles soient révélées lors d'un *dump* de la mémoire. Généralement les effacements se font grâce à une procédure particulière et nous devons vérifier que toutes les exécutions du programme font bien appel à cette procédure après la manipulation des données.

Mot d'état

Dans le domaine des cartes à puce, il est courant que les procédures indiquent si les opérations qu'elles ont effectuées se sont correctement déroulées ou non. Cette indication passe par l'envoi de mots d'état (*status word*) particuliers. Ces mots d'état permettent de se repérer dans le programme et de vérifier si les attaques physiques ont bien perturbées l'endroit qui avait été identifié comme sensible. Les requêtes consistent donc à déterminer si une exécution d'une procédure retourne bien un mot d'état donné.

Compteur de séquence

Les attaques physiques peuvent modifier le flot de contrôle d'un programme. Une contre-mesure usuelle consiste à utiliser des variables qui sont positionnées tout le long du flot de contrôle du programme (voir section 3.6.2). La sécurité repose sur le fait que ces variables sont incrémentées lors d'une exécution et que ponctuellement

le programme vérifie leur cohérence (i.e. si la variable ne contient pas la bonne valeur alors une perturbation a été détectée). Une première approche consiste à vérifier si les procédures censées être protégées possèdent bien une incrémentation du compteur de séquence. Une seconde approche, plus fine, consiste à vouloir vérifier s'il existe une exécution le long de laquelle le compteur n'est jamais incrémenté.

7.2 Étude de l'existant

Notre outil se base sur le graphe de flot de contrôle d'un programme et l'enrichit afin de permettre la spécification d'un comportement particulier en utilisant la logique temporelle CTL. Cette approche nous permet de vérifier si des politiques de sécurité ont été correctement implantées dans un programme C. Au cours de ces dix dernières années, plusieurs projets traitant de la vérification de programmes à base de *model-checking* ont eux aussi des approches basant leurs analyses sur le graphe de flot de contrôle d'un programme. Dans ce qui suit, nous présentons des outils ou techniques qui permettent de vérifier si des programmes écrits en Java ou en C sont conformes à des spécifications.

7.2.1 MOPS

Le premier outil que nous présentons se nomme MOPS (MODEL-CHECKING PROGRAMS FOR SECURITY PROPERTIES voir [121, 122]) et il permet d'analyser des programmes écrits en langage C. Étant donné un programme et une propriété de sécurité, MOPS vérifie si le programme qu'il analyse viole ou non cette propriété. Les propriétés de sécurité que MOPS vérifie sont des propriétés temporelles de sécurité innocuité¹ (i.e. propriétés qui requièrent que le programme fasse certaines opérations relevant de la sécurité dans un certain ordre). Pour ce faire, l'utilisateur de MOPS décrit sa propriété de sécurité au moyen d'un automate à états finis. L'outil vérifie ensuite si le programme viole la propriété de sécurité temporelle en utilisant du *pushdown model-checking*. Cette technique recherche de façon exhaustive dans le graphe de flot de contrôle du programme s'il existe un chemin qui ne vérifie pas la propriété définie ci-avant.

Bien que cette approche soit intéressante, elle ne couvre pas exactement nos besoins de départ. En effet, cet outil vérifie bien des propriétés temporelles en utilisant le graphe de flot de contrôle d'un programme mais le fait que l'utilisateur soit obligé d'utiliser un automate à états finis pour spécifier le comportement attendu rend son utilisation contraignante. Enfin, MOPS permet de définir qu'une procédure doit toujours être appelée après une autre, mais il ne permet pas de vérifier des propriétés sur les variables (e.g. est-ce qu'après l'appel à la procédure p , la variable v est toujours décrémentée et ce sur tous les chemins d'exécution ?).

¹Le terme anglais est *safety*, correspondant à l'aptitude à éviter de faire apparaître des événements critiques ou catastrophiques.

7.2.2 Propriétés de sécurité et graphe de flot de contrôle

L'approche développée par Besson et al. ([123]) consiste à présenter un formalisme basé sur la logique du temps linéaire pour spécifier des propriétés de sécurité globales se rapportant au graphe de flot de contrôle d'un programme. Besson et al. définissent un modèle de programme minimaliste, orienté sécurité qui prend en compte seulement les appels de procédures et des propriétés de sécurité devant être vérifiées lors de l'exécution d'un programme.

Les propriétés de sécurité sont exprimées via des formules écrites en logique du temps linéaire (LTL voir section 7.3.2). Cette formule est ensuite traduite en automate à états finis qui joue un rôle central dans l'algorithme de vérification. L'algorithme de vérification suit tous les chemins possibles du graphe de flot de contrôle en laissant simultanément l'automate évoluer ; il suffit alors de vérifier si l'automate termine dans un état acceptant.

Le modèle ainsi défini est générique et peut être adapté à différents langages ; d'ailleurs les auteurs ont pris comme cadre d'application de leur technique la sécurité du langage Java 2 basée sur l'inspection de la pile Java et des appels de méthodes disposant de privilèges.

Au vu des résultats, la technique employée semble faire ses preuves. Pourtant, la logique employée pour les spécifications nous semble trop restrictive car elle ne permet pas d'exprimer les notions telles que « pour toutes les exécutions » et surtout ne permet pas d'exprimer des propriétés sur les variables. De plus nous ne pouvons pas changer la logique employée car leur algorithme de vérification est basée sur le fait qu'il existe une relation d'équivalence entre LTL et les automates de Büchi.

7.2.3 Aoraï

L'outil AORAÏ est un greffon de l'analyseur statique FRAMA-C (voir [106]) et il permet de vérifier des spécifications exprimées à l'aide de la logique LTL.

Cet outil (voir [124]) propose de décrire le comportement d'un programme en utilisant soit un langage de description d'automate (via le langage YA) soit une formule LTL. Ensuite, fort de la description du comportement attendu du programme et de son code source écrit en C, l'outil génère un nouveau fichier C annoté qui sera vérifié avec l'outil Jessie. Jessie est un greffon pour FRAMA-C qui est basé sur le calcul des *weakest precondition*. Il permet de prouver que des procédures C satisfont leurs spécifications exprimées dans le langage ACSL².

Pourtant, en l'état actuel des choses, nous voyons deux limitations à son utilisation. Premièrement, la déclaration du langage C `switch` n'est pas supporté. Deuxièmement, dans notre cadre d'analyse sécuritaire, le fait d'énoncer des propriétés via la logique LTL ne permet pas de quantifier sur les chemins d'exécution ; qui est la problématique à laquelle nous souhaitons répondre.

²Langage d'annotation qui permet d'exprimer des propriétés de sécurité. Il peut être comparé à JML pour Java.

7.2.4 Goanna

L'outil GOANNA (voir par exemple [125, 126, 127]) présente une approche très intéressante pour analyser des programmes écrits en langage C/C++. Il s'agit d'un analyseur statique qui, à partir du code source d'une application, reconstruit le graphe de flot de contrôle d'un programme puis l'étiquette avec des propositions pertinentes en vue de vérifier une propriété. Une fois le graphe de flot de contrôle étiqueté, GOANNA le traduit dans le langage du model-checker NuSMV (voir [128]) autorisant ainsi l'utilisation de la logique temporelle CTL en vue de vérifier des propriétés sur les nœuds du graphe.

Afin de faciliter la formulation de propriétés de sécurité, cet outil fournit depuis juin 2009 deux langages (voir [129]). Le premier, GPSL, est un langage d'abstraction de haut niveau. Il s'agit d'une collection de *patterns* habituels pour la vérification associé à une bibliothèque d'objets prédéfinis. Ces objets prédéfinis correspondent à des requêtes concernant par exemple les endroits où la mémoire est allouée, utilisée ou désallouée. Grâce à ce langage, l'utilisateur final a la possibilité de définir de nouvelles vérifications basées sur les patterns prédéfinis et les objets de la bibliothèque. Le second langage se nomme GXSL. Il s'agit d'un langage de bas niveau qui est utilisé pour construire les objets prédéfinis de la bibliothèque et qui s'adresse aux experts d'outils d'analyse statique. Ce langage permet d'aller chercher directement des informations dans l'arbre de syntaxe abstraite et d'exprimer en CTL des vérifications sur ces mêmes informations.

GOANNA est un outil commercial, dont les fonctionnalités concernant les langages GPSL/GXSL sont apparues au milieu de l'année 2009. Au moment de l'établissement de l'état de l'art des outils existants, nous n'avions pas retenu GOANNA car d'une part il n'était pas si abouti qu'aujourd'hui et d'autre part, même en l'achetant nous n'aurions pas pu avoir accès au code source pour l'adapter selon nos besoins. Ainsi, le développement de notre outil s'est fait en parallèle de celui de GOANNA et pour cette raison il présente des similarités fortes.

7.2.5 Choix technique

Au vu de ces contraintes nous avons choisi d'utiliser la logique temporelle et plus spécifiquement CTL (*Computation Tree Logic* voir section 7.3.2) qui nous servira à énoncer formellement des propriétés sur le graphe de flot de contrôle d'un programme à analyser (i.e. les exécutions du programme). L'approche que nous avons retenue est proche de celle utilisée par GOANNA.

7.3 Vérification et *model-checking*

Le *model-checking* est une technique de vérification de logiciels (voir [101]). Le principe du model-checking consiste à vérifier certaines des propriétés du modèle du système étudié. Plus précisément, le model-checking passe par la spécification d'un

comportement du système (ou programme) que nous souhaitons vérifier et d'une formule exprimant une propriété que nous souhaitons voir satisfaite par le modèle.

7.3.1 Modélisation

Dans le domaine de la vérification, le modèle est généralement donné sous la forme d'une *structure de Kripke* (voir définition 1).

Définition 1. Étant donné P un ensemble fini de propositions atomiques, une structure de Kripke est un quintuplet $S = \langle Q, q_0, P, \rightarrow, l \rangle$ où Q est un ensemble fini d'états de contrôle, $q_0 \in Q$ est l'état initial, \rightarrow est une relation binaire sur Q , et $l : Q \mapsto 2^P$ est un étiquetage de Q par des propositions de P .

Les structures de Kripke (voir [101, 130]) sont une abstraction du comportement d'un système ou d'un programme dont nous voulons tester certaines propriétés. Les états du système sont représentés par les états de contrôle de la structure de Kripke. La relation binaire \rightarrow représente les possibilités d'évolution du modèle à partir de chaque état.

Cette évolution du modèle nous amène à définir formellement son comportement en introduisant la notion de *chemin*. Un *chemin* dans une structure de Kripke S est un mot $\pi = q_0q_1q_2\dots$, fini ou infini sur Q tel que :

$$\forall i \geq 0, q_i \rightarrow q_{i+1}$$

La longueur d'un chemin π , notée $|\pi|$, correspond au nombre de transitions qu'il contient (éventuellement infini). Le $i^{\text{ème}}$ état de π noté $\pi(i)$, est l'état q_i atteint après i transitions. Il n'est défini que si $i > |\pi|$.

Une *exécution partielle* de S est un chemin partant de l'état initial q_0 tandis qu'une *exécution complète* est une exécution maximale, autrement dit qu'on ne peut pas prolonger. Dans ce qui suit, lorsque nous parlerons d'exécution sans plus de précision, nous ferons référence à une exécution complète.

Nous venons de présenter une façon de modéliser un système (ou le comportement d'un programme), dans ce qui suit nous allons montrer comment spécifier des propriétés sur un modèle.

7.3.2 Spécification et logiques temporelles

Les logiques temporelles correspondent aux langages utilisés pour exprimer des propriétés d'un système (ou programme) évoluant dans le temps. Ces logiques sont particulièrement bien adaptées aux structures de Kripke (voir par exemple [131, 132, 133]). Elles permettent d'exprimer qu'un événement a va se produire à la prochaine étape de l'évolution du système ou que sous certaines conditions c , un événement b se produira certainement, sans que le moment précis soit connu pour autant. Nous présentons ici quelques logiques temporelles parmi celles qui sont les plus utilisées.

CTL*

La logique temporelle CTL* (pour *Computation Tree Logic*) fut introduite par Emerson et Halpern (voir [134]) et elle sert à énoncer formellement des propriétés portant sur les exécutions d'un système.

Cette logique permet de mélanger des propriétés portant sur une exécution particulière d'un modèle (un chemin dans la structure de Kripke auquel cas on parlera de *linear time logic*) et sur les possibilités d'évolution d'une exécution (*branching time logic*). La logique CTL* permet d'exprimer des formules écrites dans d'autres logiques temporelles telles LTL ou CTL qui seront présentées dans ce qui suit.

Syntaxe de CTL*. La syntaxe de CTL est définie par :

$$\begin{aligned}
\phi_S ::= & P_1 \mid P_2 \mid \dots && (\textit{Propositions atomiques}) \\
& \mid \neg\phi_S \mid \phi_S \vee \phi_S \mid \phi_S \wedge \phi_S \mid \phi_S \Rightarrow \phi_S \mid \dots && (\textit{Combinateurs booléens}) \\
& \mid \mathbf{E}\phi_P \mid \mathbf{A}\phi_P && (\textit{Quantificateurs de chemins}) \\
\phi_P ::= & \phi_S \\
& \mid \mathbf{X}\phi_P \mid \mathbf{F}\phi_P \mid \mathbf{G}\phi_P \mid \phi_P \mathbf{U}\phi_P \mid \dots && (\textit{Combinateur temporel})
\end{aligned}$$

Donnons à présent une définition des quantificateurs de chemins et de quelques combinateurs temporels :

- $\mathbf{A}\psi$ (Always) énonce que toutes les exécutions partant de l'état courant satisfont la propriété ψ ,
- $\mathbf{E}\psi$ (Eventually) énonce qu'il existe une exécution partant de l'état courant satisfaisant ψ ,
- $\mathbf{X}P$ (X pour neXt) énonce que l'état suivant vérifie P ,
- $\mathbf{F}P$ exprime qu'un état futur vérifie P (sans préciser quel état),
- $\mathbf{G}P$ exprime que tous les états futurs vérifient P ,
- $\psi_1 \mathbf{U}\psi_2$ énonce que ψ_1 est vérifié jusqu'à ce que ψ_2 le soit (i.e. ψ_2 sera vérifiée un jour et en attendant ψ_1 restera vraie).

Les formules de type ϕ_S sont appelées **formules d'état** car leur sémantique porte sur un état de la structure de Kripke. Celles de type ϕ_P sont appelées **formules de chemin**.

Sémantique de CTL*. Nous allons maintenant aborder la notion de formule satisfaite dans une situation donnée. On écrira $S, \pi, i \models \phi$ et on lira « au temps i de l'exécution π , ϕ est vraie », cela en parlant d'exécutions de S dont on n'exige pas qu'elles soient issues de l'état initial. On définit de manière inductive la sémantique

de CTL* sur un chemin $\pi = q_0q_1q_2 \dots$ dans une structure de Kripke par :

$$\begin{array}{ll}
\pi, i \models P & \text{ssi } P \in l(\pi(i)), \\
\pi, i \models \neg\phi & \text{ssi } \pi, i \not\models \phi, \\
\pi, i \models \phi \wedge \psi & \text{ssi } \pi, i \models \phi \text{ et } \pi, i \models \psi, \\
\pi, i \models \mathbf{X}\phi & \text{ssi } i < |\pi| \text{ et } \pi, i+1 \models \phi, \\
\pi, i \models \mathbf{F}\phi & \text{ssi il existe } j \text{ tel que } i \leq j \leq |\pi| \text{ et } \pi, j \models \phi, \\
\pi, i \models \mathbf{G}\phi & \text{ssi pour tout } j \text{ tel que } i \leq j \leq |\pi| \text{ on a } \pi, j \models \phi, \\
\pi, i \models \phi \mathbf{U}\psi & \text{ssi il existe } j, i \leq j \leq |\pi| \text{ tel que } \pi, j \models \psi, \text{ et} \\
& \text{pour tout } k \text{ tel que } i \leq k \leq j, \text{ on a } \pi, k \models \phi, \\
\pi, i \models \mathbf{E}\phi, & \text{ssi il existe un } \pi' \text{ tel que } \pi(0) \dots \pi(i) = \pi'(0) \dots \pi(i) \text{ et } \pi', i \models \phi, \\
\pi, i \models \mathbf{A}\phi, & \text{ssi pour tout } \pi' \text{ tel que } \pi(0) \dots \pi(i) = \pi'(0) \dots \pi(i) \text{ et } \pi', i \models \phi.
\end{array}$$

LTL

La logique LTL (pour Linear Temporal Logic voir [131]) est le fragment obtenu à partir de CTL* quand on retire les quantificateurs **A** et **E**. Elle est définie par :

$$\begin{array}{l}
\psi, \phi ::= P_1 \mid P_2 \mid \dots \\
\quad \mid \neg\phi \mid \phi \vee \psi \mid \phi \wedge \psi \mid \phi \Rightarrow \psi \mid \dots \\
\quad \mid \mathbf{X}\phi \mid \mathbf{F}\phi \mid \mathbf{G}\phi \mid \phi \mathbf{U}\psi \mid \dots
\end{array}$$

Toutes les formules de LTL sont des formules de chemin. La logique LTL peut donc être utilisée pour décrire le comportement d'une exécution d'un système en particulier. Cependant, pour une exécution donnée, une formule ϕ de LTL ne peut pas examiner les exécutions alternatives qui s'en distinguent à chaque moment où un choix non déterministe est possible.

On parle alors de formule de chemin et on parle de *logique du temps linéaire* pour ce genre de formalisme.

CTL

La logique CTL (voir [135, 136]) est un fragment de CTL* définie par :

$$\begin{array}{l}
\psi, \phi ::= P_1 \mid P_2 \mid \dots \\
\quad \mid \neg\phi \mid \phi \vee \psi \mid \phi \wedge \psi \mid \phi \Rightarrow \psi \mid \dots \\
\quad \mid \mathbf{A}\mathbf{X}\phi \mid \mathbf{E}\mathbf{F}\phi \mid \mathbf{A}\mathbf{G}\phi \mid \mathbf{A}(\phi \mathbf{U}\psi) \mid \dots
\end{array}$$

Elle exige que chaque utilisation d'un combinateur temporel (**X**, **F**, **U**, etc.) soit immédiatement sous la portée d'un quantificateur **A** ou **E**. Ce qui fait que toutes les formules de CTL sont des formules d'état.

7.4 Modèle de programme

Dans la section précédente, nous avons vu comment modéliser un système puis comment spécifier des propriétés sur ce système en utilisant une logique temporelle.

Dans ce qui suit, nous allons expliquer comment nous avons utilisé les structures de Kripke pour modéliser le comportement d'un programme (écrit en langage C) en vue de spécifier si certaines politiques de sécurité sont correctement implémentées.

Afin de pouvoir vérifier des propriétés sur un programme, nous allons travailler à partir de son abstraction fournie par son graphe de flot de contrôle. Le graphe de flot de contrôle d'un programme est un graphe dont les nœuds contiennent des assertions (e.g. instructions `x++`, `f(*p)`, etc.) et les transitions entre états correspondent à l'ensemble des exécutions du programme. Le graphe de flot de contrôle fournit donc une abstraction des exécutions d'un programme. Cependant, l'information contenue dans chaque nœud n'est pas suffisante pour spécifier le comportement du programme et le vérifier via une formule. Nous devons donc enrichir cette structure afin de la transformer en une structure de Kripke qui nous permettra de tester certaines propriétés.

Une structure de Kripke désigne un système de transitions entre états où les états sont étiquetés par des propositions. Ainsi, la transformation d'un graphe de flot de contrôle en une structure de Kripke s'opère en étiquetant chacun de ses nœuds par des propositions atomiques correspondant aux propriétés que nous souhaitons vérifier. Notre approche consiste à abstraire toute l'information concernant le flot de données afin de ne garder que des nœuds étiquetés par des propositions atomiques qui correspondent à des propriétés d'intérêt. Autrement dit, le calcul de chemins dans le flot de contrôle se fera en considérant le graphe de flot de contrôle comme un graphe étiqueté. Notre approche est ainsi similaire à celle décrite par Besson et al. dans [123].

Le graphe de flot de contrôle résultant est décrit par un ensemble de nœuds NO , un nœud de départ n_0 , un ensemble de transitions T et des étiquettes ET :

$$G = (NO, n_0, T, ET)$$

où les différents composants ont la signature suivante :

$$\begin{aligned} ET & : NO \rightarrow \{\text{Call Func}, \text{Return}, \text{Incr Var}, \text{Decr Var}, \text{Other}\} \\ n_0 & : NO \\ T & : NO \rightarrow \mathcal{P}(NO) \end{aligned}$$

Nous avons défini un modèle dans lequel les nœuds peuvent être étiquetés par différents types indiqués par ET :

- un appel de procédure (**Call Func**),
- un retour de procédure (**Return**),
- l'incrément (resp. décrémentation) d'une variable (**Incr Var**, **Decr Var**),
- la sémantique d'un nœud qui ne présente pas d'intérêt pour la vérification de propriétés (**Other**).

La liste des étiquettes peut être enrichie en fonction des besoins de l'utilisateur.

La figure 45 illustre le procédé de transformation d'un programme en une structure de Kripke.

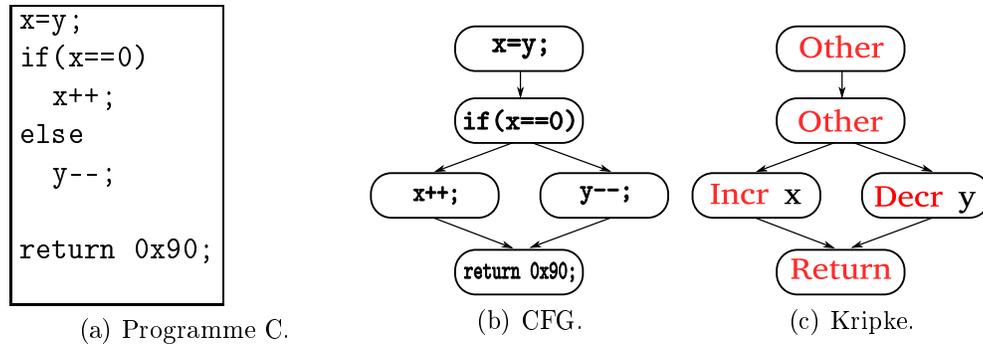


FIG. 45: Graphe de flot de contrôle et structure de Kripke.

Une propriété s'énonce par une formule de la logique temporelle dont la sémantique porte sur les états de contrôle de la structure de Kripke et les chemins de cette structure. Ainsi, le model-checking consiste à prendre en entrée une structure de Kripke S et une formule de logique temporelle ϕ et à répondre « Oui » si et seulement si S satisfait ϕ .

Imaginons que sur notre exemple de la figure 45, nous souhaitons vérifier « s'il existe un chemin comportant un état dans lequel la variable x est incrémentée », nous écririons la formule CTL suivante : $\mathbf{EF}(\text{Incr } x)$ et l'outil nous répondrait « Oui » en exhibant le chemin correspondant.

Traduction de nos requêtes en CTL. Nous présentons ici la traduction des requêtes définies dans la section 7.1.1 en CTL :

- appel au pare-feu : \mathbf{AF} (Call *firewall*)
- effacement du buffer de clef : \mathbf{AF} (Call *clear_buffer*)
- mot d'état : \mathbf{EF} (Return *status_word*)
- compteur de séquence : \mathbf{AF} (Incr *cpt_sequence*) ou $\mathbf{EG} \neg(\text{Incr } \textit{cpt_sequence})$

7.5 Sémantique, notion d'exécution et CTL

Notre outil est conçu pour être pratique (i.e. utilisable dans un processus d'analyse de code source) qui permet de vérifier certaines propriétés sur des grands programmes. Pour cette raison, nous avons fourni un outil qui tente d'établir un équilibre entre validité, précision et mise à l'échelle.

7.5.1 Validité

Un graphe de flot de contrôle comporte toutes les informations qui sont présentes dans le code source d'un programme (i.e. les assertions et leurs liens entre elles). En ce sens, il peut être considéré comme un programme écrit dans un langage où la sémantique peut correspondre à une véritable exécution (i.e. une trace de

programme). Autrement dit, une exécution peut être vue comme un chemin dans le graphe de flot de contrôle.

Toutefois, l'inverse est faux : un chemin dans le graphe ne correspond pas nécessairement à une exécution réelle du programme (voir figure 46).

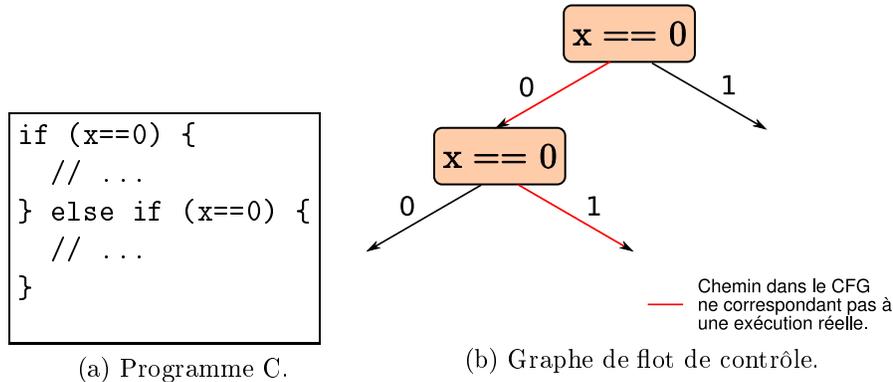


FIG. 46: Exécution et chemin dans le graphe de flot de contrôle.

Un graphe de flot de contrôle modélise tous les chemins d'exécution possibles d'un programme, ce qui fait qu'il sur-approxime le comportement réel du programme. En effet, définir les exécutions possibles du programme comme étant les chemins présents dans le graphe de flot de contrôle suppose que n'importe quel branchement conditionnel peut être pris ou non, comme si les conditions pouvaient avoir n'importe quelle valeur à n'importe quel moment.

Cette constatation a une conséquence directe sur notre analyse : le model-checking se base sur les chemins d'un graphe qui ne correspondent pas forcément à des exécutions réelles : ce qui nous mène tout droit à la notion de chemins infaisables (*infeasible paths* voir définition 2).

Définition 2. Soit b un branchement conditionnel avec un prédicat d'expression p et c un **chemin** (i.e. une séquence d'éléments connexes dans le graphe de flot de contrôle) partant du début du programme jusqu'à l'arête de sortie vraie (resp. fausse) du nœud b . Le chemin c est dit **infaisable** si le prédicat p évalue toujours la condition comme fausse (resp. vraie) quand le flot de contrôle atteint le nœud b le long du chemin c .

Ces chemins infaisables ont un impact sur les résultats fournis par notre outil ; ce que nous allons étudier à présent.

7.6 Programme et satisfaction de formules logiques

Dans notre approche, la vérification de formules logiques exprimées en CTL, s'effectue en se basant sur un modèle abstrait : le graphe de flot de contrôle étiqueté d'un programme. Dans ce qui suit nous donnons une modélisation dont le caractère

n'est pas complètement formel pour exprimer dans quelle mesure une formule logique satisfaite par le graphe de flot de contrôle étiqueté d'un programme reste satisfaite par le programme concret. Nous nous autorisons ce petit manque de rigueur car il s'agit de concepts bien connus dans le folklore des méthodes formelles.

Définition 3. Un programme \mathbb{P} est une séquence d'assertions $\mathbb{P} = \mathbf{a}_1; \dots; \mathbf{a}_n$. Nous considérons qu'un point de programme \mathbf{p}_i est une assertion \mathbf{a}_i où chaque assertion est définie par la grammaire formelle de CIL (voir section 6.4.2)

Définition 4. Une trace de \mathbb{P} correspond à la séquence des assertions \mathbf{a}_i qui seront exécutées ainsi que les données contenant l'état du programme. Nous supposons que \mathbf{a}_1 est la première assertion à être exécutée.

Définition 5. Toute trace d'un programme \mathbb{P} correspond à un chemin dans le graphe de flot de contrôle. La réciproque est fautive car il existe des chemins dans le graphe de flot de contrôle qui ne correspondent pas à une trace d'exécution (voir définition 2).

Note : Lors de la construction du graphe de flot de contrôle, CIL transforme chaque assertion (*statement*) en un nœud du graphe. Nous avons donc une bijection entre les assertions (i.e. les points de programme) et les nœuds du graphe de flot de contrôle.

7.6.1 Satisfaction de formules logiques

Soient \mathbb{P} un programme, CFG_{ET} un graphe de flot de contrôle étiqueté (voir section 7.4), \mathbf{p} un point de programme et ϕ une formule logique exprimée en CTL. La propriété $\mathcal{P}(\phi, \mathbf{p})$ qui affirme « Si ϕ satisfaite dans CFG_{ET} alors ϕ satisfaite dans \mathbb{P} à partir de \mathbf{p} » est définie par :

$$\mathcal{P}(\phi, \mathbf{p}) ::= ((\mathbf{p}, CFG_{ET}) \models \phi) \Rightarrow ((\mathbf{p}, \mathbb{P}) \models \phi)$$

Pour trouver quelles sont les formules ϕ qui vérifient la propriété $\mathcal{P}(\phi, \mathbf{p})$, nous allons procéder de manière récursive sur la construction de ϕ .

Formules atomiques

Appel de fonction. Soit f une fonction et ϕ une formule atomique telles que $\phi ::= \text{Call } f$. Le graphe de flot de contrôle est un graphe où chaque nœud correspond à un point du programme \mathbf{p} de \mathbb{P} , ce qui implique que s'il existe une assertion de \mathbb{P} qui fasse appel à une fonction f , alors il existe un nœud du graphe de flot de contrôle qui correspond à ce point de programme. Le processus d'étiquetage repère dans le graphe de flot de contrôle les appels de fonction et leur appose l'étiquette **Call**, ainsi l'appel à une fonction f sera étiqueté dans CFG_{ET} par **Call** f . Il s'ensuit que $((\mathbf{p}, CFG_{ET}) \models \phi) \Rightarrow ((\mathbf{p}, \mathbb{P}) \models \phi)$.

Note : Les démonstrations de $\mathcal{P}(\mathbf{p})$ pour les étiquettes **Return**, **Incr Var** et **Decr Var** sont identiques à celle de l'appel de fonction **Call** f .

Formules non atomiques

Quantificateur A et combinateur G. Soit $\phi ::= \mathbf{AG}\psi$ et ψ satisfait $\mathcal{P}(\phi, \mathbf{p}) \forall \mathbf{p}$. Soient \mathcal{C} l'ensemble des chemins de CFG_{ET} et $\pi(j)$ avec $\pi \in \mathcal{C}$ le nœud atteint après j transitions partant de \mathbf{p} alors : $(\mathbf{p}, CFG_{ET}) \models \mathbf{AG}\psi$ signifie que $\forall \pi \in \mathcal{C}$, et $\forall j$ tel que $\mathbf{p} \leq \pi(j) \leq |\pi|$ on a $\pi(j) \models \psi$. Comme CFG_{ET} décrit plus de chemins qu'il n'y a de traces réelles du programme (voir définition 2) et que chaque point de programme $\pi(j)$ correspond à un nœud étiqueté de CFG_{ET} , il s'ensuit que $((\mathbf{p}, CFG_{ET}) \models \phi) \Rightarrow ((\mathbf{p}, \mathbb{P}) \models \phi)$.

Quantificateur E et combinateur G. Soit $\phi ::= \mathbf{EG}\psi$ et ψ satisfait $\mathcal{P}(\phi, \mathbf{p}) \forall \mathbf{p}$. Soient \mathcal{C} l'ensemble des chemins de CFG_{ET} et $\pi(j)$ avec $\pi \in \mathcal{C}$ le nœud atteint après j transitions partant de \mathbf{p} alors : $(\mathbf{p}, CFG_{ET}) \models \mathbf{EG}\psi$ signifie que $\exists \pi \in \mathcal{C}$, où $\forall j$ vérifiant $\mathbf{p} \leq \pi(j) \leq |\pi|$ on a $\pi(j) \models \psi$. D'après la définition 2, le chemin π de CFG_{ET} vérifiant ψ peut être infaisable d'où $((\mathbf{p}, CFG_{ET}) \models \phi) \not\Rightarrow ((\mathbf{p}, \mathbb{P}) \models \phi)$.

Quantificateur A et combinateur F. Soit $\phi ::= \mathbf{AF}\psi$ et ψ satisfait $\mathcal{P}(\phi, \mathbf{p}) \forall \mathbf{p}$. Soient \mathcal{C} l'ensemble des chemins de CFG_{ET} et $\pi(j)$ avec $\pi \in \mathcal{C}$ le nœud atteint après j transitions partant de \mathbf{p} alors : $(\mathbf{p}, CFG_{ET}) \models \mathbf{AF}\psi$ signifie que $\forall c \in \mathcal{C}$, $\exists j$ tel que $\mathbf{p} \leq \pi(j) \leq |\pi|$ on a $\pi(j) \models \psi$. Comme CFG_{ET} décrit plus de chemins qu'il n'y a de traces réelles du programme et que le point de programme $\pi(j)$ correspond à un nœud étiqueté de CFG_{ET} , il s'ensuit que $((\mathbf{p}, CFG_{ET}) \models \phi) \Rightarrow ((\mathbf{p}, \mathbb{P}) \models \phi)$.

Quantificateur E et combinateur F. Soit $\phi ::= \mathbf{EF}\psi$ et ψ satisfait $\mathcal{P}(\phi, \mathbf{p}) \forall \mathbf{p}$. Soient \mathcal{C} l'ensemble des chemins de CFG_{ET} et $\pi(j)$ avec $\pi \in \mathcal{C}$ le nœud atteint après j transitions partant de \mathbf{p} alors : $(\mathbf{p}, CFG_{ET}) \models \mathbf{EF}\psi$ signifie que $\exists \pi \in \mathcal{C}$, $\exists j$ tel que $\mathbf{p} \leq \pi(j) \leq |\pi|$ on a $\pi(j) \models \psi$. D'après la définition 2, le chemin π de CFG_{ET} où $\pi(j)$ vérifie ψ peut être infaisable, il s'ensuit que $((\mathbf{p}, CFG_{ET}) \models \phi) \not\Rightarrow ((\mathbf{p}, \mathbb{P}) \models \phi)$.

Quantificateur A et combinateur X. Soit $\phi ::= \mathbf{AX}\psi$ et ψ satisfait $\mathcal{P}(\phi, \mathbf{p}) \forall \mathbf{p}$. Soient \mathcal{C} l'ensemble des chemins de CFG_{ET} et $\pi(j)$ avec $\pi \in \mathcal{C}$ le nœud atteint après j transitions partant de \mathbf{p} alors : $(\mathbf{p}, CFG_{ET}) \models \mathbf{AX}\psi$ signifie que $\forall c \in \mathcal{C}$, $\exists j$ tel que $j < |\pi|$ et $\pi(j+1) \models \psi$. Comme CFG_{ET} décrit plus de chemins qu'il n'y a de traces réelles du programme et que le point de programme $\pi(j+1)$ correspond à un nœud étiqueté de CFG_{ET} , il s'ensuit que $((\mathbf{p}, CFG_{ET}) \models \phi) \Rightarrow ((\mathbf{p}, \mathbb{P}) \models \phi)$.

Quantificateur E et combinateur X. Soit $\phi ::= \mathbf{EX}\psi$ et ψ satisfait $\mathcal{P}(\phi, \mathbf{p}) \forall \mathbf{p}$. Soient \mathcal{C} l'ensemble des chemins de CFG_{ET} et $\pi(j)$ avec $\pi \in \mathcal{C}$ le nœud atteint après j transitions partant de \mathbf{p} alors : $(\mathbf{p}, CFG_{ET}) \models \mathbf{EX}\psi$ signifie que $\exists \pi \in \mathcal{C}$, $\exists j$ tel que $j < |\pi|$ et $\pi(j+1) \models \psi$. D'après la définition 2, le chemin π de CFG_{ET} où $\pi(j+1)$ vérifie ψ peut être infaisable, il s'ensuit que $((\mathbf{p}, CFG_{ET}) \models \phi) \not\Rightarrow ((\mathbf{p}, \mathbb{P}) \models \phi)$.

7.6.2 Précision

Un outil précis est un outil qui reporte peu (voir très peu) de faux positifs. Rappelons nous que nous souhaitons prouver des propriétés bien précises sur un programme (voir 7.1.1) et que ces propriétés peuvent se ramener à :

- savoir si tous les chemins d’une procédure Y comporte un appel à une procédure donné H (e.g. $Y \Rightarrow \mathbf{AF}(\mathbf{Call} H)$),
- savoir s’il existe un chemin d’une procédure Y qui comporte un état avec la propriété K (e.g. $Y \Rightarrow \mathbf{EF}(\mathbf{Incr} x)$).

Toute la question est de savoir quelle assurance nous pouvons avoir dans les résultats en cas de réponse « Oui » ou « Non » de l’outil.

Formules « sûres »

Considérons les formules suivantes :

$$\begin{aligned}\phi_1 & ::= AF\psi_1, \\ \phi_2 & ::= AG\psi_2, \\ \phi_3 & ::= AX\psi_3\end{aligned}$$

avec ψ_i satisfait $\mathcal{P}(\mathbf{p})\forall\mathbf{p}$.

Nous venons de montrer que si ces formules sont satisfaites sur le graphe de flot de contrôle étiqueté alors elles sont satisfaites par le programme. Autrement dit, en cas de réponse « Oui » de notre outil sur ces formules nous avons l’assurance que la réponse est exacte. Nous qualifierons donc ces formules de « sûres ».

Faux positifs

Le fait de considérer plus de chemins dans le graphe de flot de contrôle qu’il n’y a d’exécutions réelles dans le programme conduit notre outil à exhiber des faux positifs (i.e. émettre un avertissement en disant que telle propriété n’est pas vérifiée alors qu’elle ne correspond à aucune réalité). Nous devons distinguer deux cas : en cas de réponse « Oui » et en cas de réponse « Non ».

Réponse « Oui ». Nous venons de démontrer que les formules de la forme :

$$\begin{aligned}\phi_1 & ::= EF\psi_1, \\ \phi_2 & ::= EG\psi_2,\end{aligned}$$

avec ψ_i satisfait $\mathcal{P}(\mathbf{p})\forall\mathbf{p}$ ne vérifient pas la propriété $\mathcal{P}(\mathbf{p})$. Il s’ensuit que si notre outil répond « Oui » concernant la satisfaction de telles formules, nous n’avons aucune garantie sur le résultat fourni : il peut s’agir d’un faux positif.

Cependant, en cas de réponse positive, l’outil exhibe un chemin vérifiant la propriété exprimée par la formule logique. Ce chemin peut par la suite être validé par l’évaluateur afin de s’assurer s’il s’agit réellement ou non d’un faux positif.

Réponse « Non ». Ci-avant, nous avons expliqué que les formules :

$$\begin{aligned}\phi_1 &::= AF\psi_1, \\ \phi_2 &::= AG\psi_2, \\ \phi_3 &::= AX\psi_3\end{aligned}$$

étaient valides en cas de réponse « Oui » de notre outil. Cependant, si notre outil répond, « Non », nous n'avons aucune assurance que ce résultat soit vrai.

Prenons un exemple avec une requête qui exprime « qu'à partir du début du programme, toutes les exécutions comportent un état futur e ». Cette requête est valide mais en réalité fait référence à toutes les exécutions *réelles* du programme. Or, notre vérification se fait sur le graphe de flot de contrôle qui sur-approxime les exécutions réelles d'un programme, ce qui implique que nous pouvons parcourir des chemins infaisables. Autrement dit nous pouvons avoir une réponse « Non » si un chemin infaisable ne comporte pas l'état e et dans cas précis il s'agit d'un faux positif.

Faux négatifs

Un faux négatif signifie que le résultat d'un test est déclaré négatif alors qu'il devrait être positif. Comme nous venons de le voir, notre analyse peut afficher des faux positifs, cependant la recherche dans les chemins est exhaustive car elle parcourt tous les états de tous les chemins. Par conséquent l'analyse n'oublie pas de propriétés sur les états.

Nous dirons donc que notre analyse est valide (*sound* en anglais) car elle peut calculer des faux positifs mais en aucun cas des faux négatifs.

Vers plus de précision

Les chemins infaisables posent un véritable problème quant à la précision de l'analyse. Tout l'enjeu est donc d'essayer d'éliminer au maximum ces chemins ce qui permettrait d'avoir une assurance plus importante dans les résultats obtenus.

Un axe possible d'amélioration concerne l'annotation de branchements conditionnels. Notre approche serait similaire à celle décrite dans la section 6.8. L'évaluateur, grâce à son expertise et sa connaissance du système, peut via un système d'annotation indiquer si des branchements ne seront jamais empruntés par le programme.

Complétude

Comme n'importe quelle propriété non triviale concernant la reconnaissance d'un langage par une machine de Turing est indécidable (voir théorème de Rice [79]). Notre outil essaye d'être valide (*sound* en anglais) mais il n'est pas complet dans le sens où il peut générer des traces de programme qui ne correspondent pas à de véritables exécutions.

7.7 Présentation de l’outil

Cette partie a pour vocation de présenter comment s’articule notre système de requêtes au travers d’un exemple concret. Avant de présenter de façon pratique comment utiliser notre outil, nous souhaitons préciser qu’il est capable de faire des analyses dites *intra-procédurales* et *inter-procédurales*. Dans une analyse intra-procédurale, on ne s’intéresse qu’à des propriétés locales à une procédure sans analyser le graphe de flot de contrôle des procédures qui sont éventuellement appelées. En revanche, dans une analyse inter-procédurale, tous les appels de procédures sont pris en compte et chaque graphe de flot de contrôle de chaque procédure appelée est examiné.

Dans ce qui suit nous allons illustrer nos propos en spécifiant des propriétés sur une procédure écrite en C nommée `compute_checksum`. Cette procédure calcule une somme de contrôle et a été protégée par des compteurs de séquence (modélisés par la variable `CPT_SEQUENCE`). Nous omettons volontairement dans le programme analysé les vérifications portant sur cette la variable `CPT_SEQUENCE` car cela ne fait pas l’objet des propriétés que nous souhaitons vérifier.

Étant donné que ce programme comporte des protections, tout le travail de l’évaluateur est de déterminer si oui ou non elles sont efficaces. Ainsi, la propriété que nous souhaitons vérifier consiste à savoir si sur tous les chemins de `compute_checksum` (i.e. au travers d’une analyse intra-procédurale) comportent une incrémentation du compteur de séquence `CPT_SEQUENCE` (modélisé par `CPT_SEQUENCE++`). Une description de ce programme est donné sur la figure 47 dans la fenêtre numérotée ①.

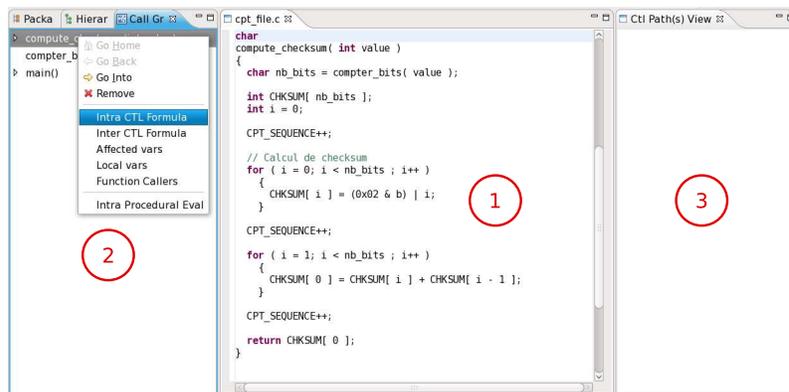


FIG. 47: Programme à analyser.

La fenêtre numérotée ② correspond aux procédures présentes dans le fichier en cours d’analyse et exhibe leur graphe d’appel. C’est d’ailleurs à partir de cette fenêtre qu’en cliquant sur le nom des fonctions, un menu déroulant apparaît proposant d’effectuer une requête intra-procédurale ou inter-procédurale. Dans notre exemple, nous sélectionnons le mode intra-procédural et une fenêtre apparaît permettant de saisir sous forme textuelle une requête CTL (voir figure 48).

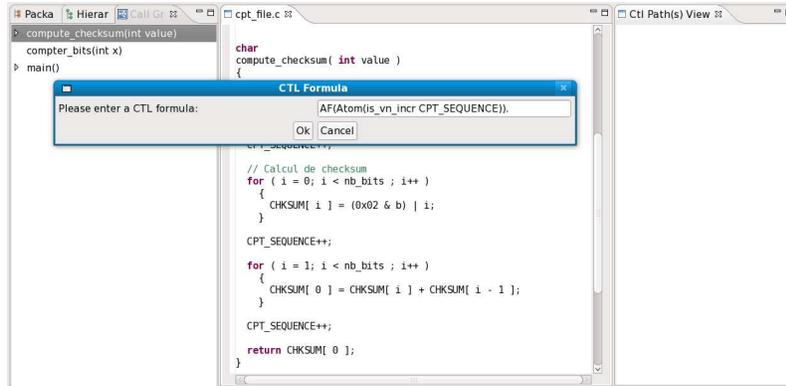
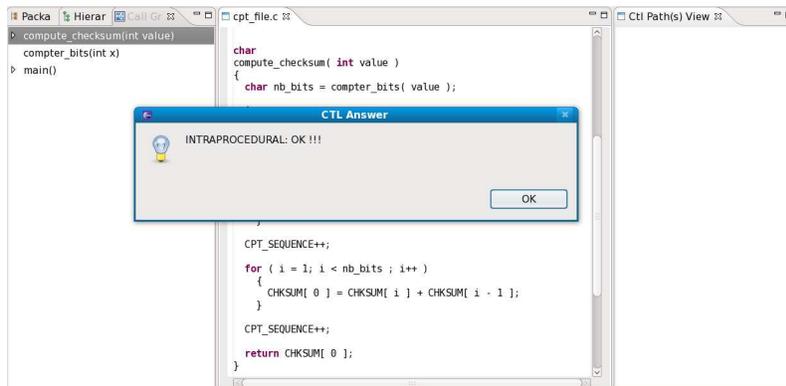


FIG. 48: Interpréteur de formules CTL.

Ici la requête exprime le fait que tous les chemins doivent comporter une incrémentation de la variable `CPT_SEQUENCE` et est décrite par :

- le combinateur temporel **F** sous la portée du combinateur **A**,
- **Atom** qui indique que ce qui suit est une propriété atomique et qui est composée de l'étiquette `is_vn_incr` qui correspond à une incrémentation de variable.

Cliquer sur OK a pour effet de valider la requête et de lancer l'analyse dont la réponse s'affiche sous une forme textuelle (voir figure 49). Nous noterons que pour

FIG. 49: Réponse du *model-checker*.

cette requête, aucun chemin ne s'affiche dans la fenêtre numérotée ③ sur la figure 47. Ceci provient du fait qu'il serait peu judicieux de faire afficher tous les chemins vérifiant la propriété énoncée.

En revanche, si nous remplaçons le quantificateur **A** par **E**, l'outil exhibe un seul chemin qui satisfait la propriété d'incrémentation du compteur de séquence. Ce chemin a comme point de départ le début de la procédure `compute_checksum` et comme fin le premier état rencontré qui comporte une incrémentation du compteur. Cet chemin apparaît dans la fenêtre intitulée *Ctl Path View* sur la figure 50.

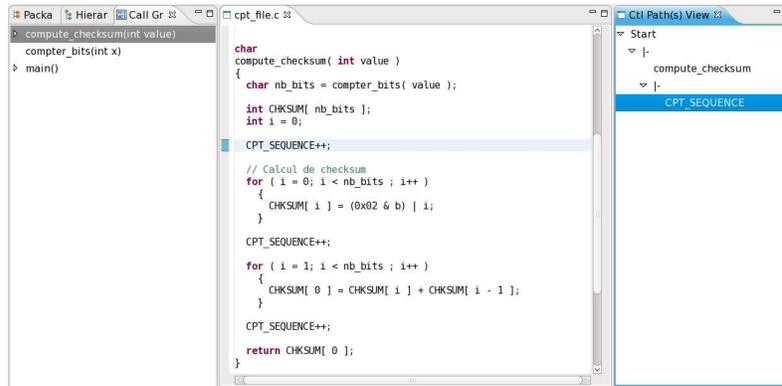


FIG. 50: Chemin calculé par l'outil.

7.8 Conclusion

Dans ce chapitre, nous venons de présenter un outil qui permet de vérifier des propriétés sur des programmes écrits en langage C. La formulation de ces propriétés passe par l'écriture de requêtes exprimées au moyen de la logique CTL.

Chercher à vérifier des propriétés sur un programme n'est pas une tâche aisée et passe souvent par une modélisation de son comportement via des automates (ou un autre formalisme). Pourtant, nous voulions absolument éviter cette étape de modélisation afin de fournir un outil qui soit flexible et simple d'utilisation. Notre approche nous a donc amené à faire des compromis et à travailler à partir du graphe de flot de contrôle d'un programme qui est une abstraction de ses exécutions réelles. Cette façon de procéder nous a amené la flexibilité recherchée (i.e. aucune modélisation préalable) mais cependant elle a un coût : les analyses peuvent générer des faux positifs.

La génération de traces de programme ne correspondant pas à une exécution réelle n'est pas vraiment surprenante car en toute généralité, il n'est pas possible d'avoir des outils concernant l'analyse de programme qui soient valides et complets ; ils sont donc obligés de faire des approximations du comportement réel d'un programme. Tout l'enjeu de futurs travaux consistera donc à essayer de rendre les analyses aussi précises que possibles tout en gardant la flexibilité première de l'outil.

Conclusion

Afin de s'adapter aux exigences du marché, la technologie des cartes à puce n'a cessé d'évoluer au cours de ces dix dernières années et avec cet essor, de nouveaux enjeux sécuritaires sont apparus. Ainsi, rendre les cartes plus sécurisées est un processus complexe impliquant différents mécanismes aussi bien de bas niveau que de haut niveau visant à protéger les composants matériels et les logiciels embarqués. Cette thèse s'inscrit dans cet objectif en proposant des outils qui permettent de vérifier si des logiciels destinés à être exécutés sur des cartes présentent des faiblesses face à des attaques spécifiques.

La première partie de nos travaux a consisté à mettre au point un environnement logiciel dédié à l'analyse de la résistance d'implémentations d'algorithmes cryptographiques face à des attaques par analyse de la consommation de courant. Notre environnement permet de définir des modèles de fuite associés à des modèles de consommation et de vérifier si les hypothèses de départ concernant les fuites sont exactes.

Nous utilisons des modèles de consommation théorique qui, comme tous les modèles, possèdent leurs propres limites. Les modèles que nous utilisons font par exemple abstraction de la longueur des pistes des bus et ne cherchent pas à modéliser d'éventuelles sources de bruit. En particulier, si par la simulation nous n'arrivons pas à déterminer des sources de fuites dans une implémentation (i.e. elle résiste à la DPA théorique), nous ne pouvons pas affirmer qu'il en sera de même quand le programme sera exécuté sur une puce réelle. En effet, il se peut que dans la réalité certains phénomènes physiques, non pris en compte par la modélisation, soient des sources de fuites (e.g. longueur des bus).

Cependant, si des fuites apparaissent lors de la simulation, notre environnement permet d'affiner leur recherche en se focalisant sur des parties très précises comme une instruction particulière ou un registre donné. Cette approche permet à des évaluateurs de déterminer en quelques minutes si une implémentation est vulnérable à une attaque de type DPA, facilitant ainsi la mise en place d'une attaque réelle.

La deuxième partie de nos travaux s'inscrit dans la continuité de la recherche de vulnérabilités dans des implémentations de bas niveau. Nous nous sommes intéressés à l'analyse de programmes écrits en langage d'assemblage AVR dans le but de vérifier s'ils sont vulnérables aux *timing attacks*. Fournir un outil complètement

automatique pour résoudre ce problème n'est pas possible en raison de sources d'indécidabilité comme le nombre d'itérations que doit effectuer une boucle. Face à cette contrainte, nous nous sommes orientés vers une solution semi-automatique qui consiste à faire appel aux connaissances d'un évaluateur (i.e. à son expertise) afin de lever ces sources d'indécidabilité. La solution que nous avons retenue, consiste à reconstruire le graphe de flot de contrôle d'un programme afin de s'en servir comme support pour la description de traces d'exécution. Ces traces d'exécution d'un programme sont décrites au moyen d'expressions régulières qui seront par la suite interprétées par notre outil en vue de donner leur temps exact d'exécution (en terme de cycles d'horloge). Grâce à cette approche, notre outil permet à la fois de rechercher des vulnérabilités mais aussi de valider qu'un éventuel déséquilibre dans les temps d'exécution est bien exploitable. Enfin, l'ergonomie de l'outil n'a pas été mise de côté car nous l'avons interfacé avec l'environnement de développement Eclipse. D'ailleurs, cette intégration a débouché sur une représentation agréable du temps d'exécution des blocs de base du graphe de flot de contrôle et par la même occasion a rendu possible la modélisation d'attaques par injection de fautes. Enfin cette intégration aide à visualiser l'impact de telles attaques sur le flot de contrôle, ce qui permet de les valider (ou de les invalider) avant de les mettre en œuvre sur les bancs de tests.

La troisième partie de nos travaux a trait à l'analyse de programmes écrits dans le langage de haut niveau C. Nous nous sommes concentrés sur la vérification de politiques de sécurité en donnant à l'évaluateur d'une part les moyens de comprendre plus facilement comment est structuré le programme en cours d'analyse et d'autre part en lui offrant une façon de vérifier des propriétés sur ce programme.

Ayant bien identifié quels étaient les besoins des évaluateurs en terme d'outils de navigation, nous avons pris le parti de fournir des assistants de navigation qui au travers d'informations concernant les variables et procédures rencontrées, facilitent la compréhension du programme. D'ailleurs, de ces assistants a émergé une façon originale de naviguer dans le flot de contrôle d'un programme grâce au positionnement d'expressions conditionnelles rencontrées dans des procédures. Cette assistant, guidé par l'évaluateur, génère un raffinement du graphe d'appel, facilitant la recherche de vulnérabilités mais aussi la visualisation des impacts d'éventuelles attaques par injection de fautes.

Concernant la vérification de propriétés ayant trait à l'implémentation de politiques de sécurité, nous avons mis au point une manière de les vérifier sans modélisation préalable (e.g. avec un automate à états finis) au moyen de requêtes exprimées au moyen de la logique CTL. Éviter une modélisation préalable nous a amené à faire des compromis et à travailler à partir du graphe de flot de contrôle d'un programme qui est une abstraction des exécutions réelles d'un programme. Ce compromis nous a donné la flexibilité recherchée mais cependant il a un coût concernant la précision des analyses car des faux positifs peuvent apparaître.

Perspectives

Nous identifions plusieurs axes de poursuites possibles des travaux qui ont été menés dans cette thèse.

Concernant le simulateur de microcontrôleur de carte à puce, il serait intéressant de le tester sur d'autres implémentations, sécurisées ou non, de l'algorithme cryptographique DES. Ceci nous permettrait d'étudier plus en profondeur les faiblesses des implémentations mais aussi de déterminer dans quelle mesure ces différences d'implémentations créent un frein à une attaque par DPA.

De plus, le simulateur a été écrit de sorte à ce qu'il puisse être utilisé dans un assistant de preuve. Au travers de cette approche, nous souhaiterions apporter une preuve formelle de la résistance d'une implémentation d'un algorithme cryptographique face à une attaque de type DPA.

Nous avons posé dans le chapitre 5 les bases d'un outil semi-automatique d'analyse de codes écrits en langage d'assemblage. En vue d'étendre l'outil vers d'autres types d'analyses mais aussi d'obtenir un graphe de flot de contrôle aussi précis que possible, nous aimerions développer une approche combinant analyse dynamique et analyse statique. Cette approche prendrait la forme d'une communication entre le simulateur et cet outil ce qui, nous l'espérons, devrait lever certaines sources d'indécidabilité telles que les adresses de branchement des instructions IJMP et ICALL.

Pour l'outil ayant trait à la navigation/compréhension de programmes écrits en langage C, nous envisageons plusieurs axes d'améliorations possibles. D'une part, notre intuition est qu'une analyse de code manuelle se trouve simplifiée si le code source analysé est représenté de manière plus structurée; son interprétation s'en trouve facilitée. Ainsi, nous souhaiterions améliorer la compréhension du programme en poussant encore plus loin l'ergonomie de l'outil notamment au travers d'un système de navigation dans le graphe de flot de contrôle entier.

D'autre part, nous souhaiterions réduire au maximum le nombre de faux positifs que peut générer notre model-checker tout en gardant la flexibilité première de l'outil. Nous pensons qu'un travail d'analyse statique poussé concernant l'analyse du flot de données permettrait d'éliminer un grand nombre de chemins infaisables.

Bibliographie

- [1] Michel Ugon. Support d'Information Portatif Muni d'un Microprocesseur et d'une Mémoire Morte Programmable. Technical report, Brevet n° US4211919 (numéro original : FR2401459), déposé le 26 août 1977, publié le 8 juillet 1980. Inventeur : Michel Ugon. Déposant : CII-HB (Compagnie Internationale pour l'Informatique - Honeywell Bull, 1977.
- [2] Louis C. Guillou and Michel Ugon. Smart card, a highly reliable and portable security device. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 464–479, London, UK, 1987. Springer-Verlag.
- [3] International Organization for Standardization (ISO) and International Electronic Committee (IEC). International Standard ISO/IEC 7816 : Identification Cards - Integrated Circuit(s) Cards with Contacts, 1995-2001.
- [4] International Organization for Standardization (ISO) and International Electronic Committee (IEC). International Standard ISO/IEC 10 373 : Identification Cards - Test Methods, 1995-2001.
- [5] International Organization for Standardization (ISO). International Standard ISO 7810 : Identification Cards - Physical Characteristics, 1995.
- [6] National Institute of Standards and Technology (NIST). Fips-197 : Advanced Encryption Standard (AES).
"<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>, November 2001.
- [7] Damien Deville, Antoine Galland, Gilles Grimaud, and Sébastien Jean. Smart card operating systems : Past, present and future. In *Proceedings of the 5 th NORDU/USENIX Conference*, 2003.
- [8] Java Card.
<http://java.sun.com/products/javacard/>.
- [9] Maosco Ltd. Multos.
<http://multos.com>.
- [10] Zeitcontrol Cardsystems. Basic Card.
<http://www.basiccard.com/>.
- [11] COmité FRançais d'ACcréditation (COFRAC).
<http://www.cofrac.fr/>.

- [12] Agence Nationale de la Sécurité des Systèmes d'Informations (ANSSI). <http://www.ssi.gouv.fr/>.
- [13] Common Criteria. <http://www.commoncriteriaportal.com>.
- [14] International Organization for Standardization (ISO) and International Electronic Committee (IEC). International Standard ISO/IEC 15408-1 :2005 Information technology – Security techniques – Evaluation criteria for IT security – Part 1 : Introduction and general model, 2005.
- [15] International Organization for Standardization (ISO) and International Electronic Committee (IEC). International Standard ISO/IEC 15408-2 :2005 Information technology – Security techniques – Evaluation criteria for IT security – Part 2 : Security functional requirements, 2005.
- [16] International Organization for Standardization (ISO) and International Electronic Committee (IEC). International Standard ISO/IEC 15408-3 :2005 Information technology – Security techniques – Evaluation criteria for IT security – Part 3 : Security assurance requirements, 2005.
- [17] Boutheina Chetali and Claire Loiseaux. FORMAVIE : Formal Modelling and Verification of Java Card 2.1.1 Security Architecture. In *E-SMART 2002*, pages 213–231, 2002.
- [18] ATMEL. AT90S8515 datasheet, doc0841 : 8-bit microcontroller with 8k bytes in-system programmable flash.
- [19] ATMEL. Atmega128(1) datasheet, doc2467 : 8-bit microcontroller with 128k bytes in-system programmable flash.
- [20] Andrew Tanenbaum. *Architecture de l'ordinateur*. Pearson Education France, 5 edition, 2005.
- [21] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08 : Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26. ACM, 2008.
- [22] D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens. Transaction security system. *IBM Syst. J.*, 30(2) :206–229, 1991.
- [23] Sergei P. Skorobogatov. Semi-invasive attacks – A new approach to hardware security analysis. Technical Report 630, Univeristy of Cambridge, Computer Laboratory, april 2005.
- [24] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. In *Proc. IEEE*, volume 94, pages 370–382, february 2006.
- [25] Christophe Giraud and Hugues Thiebauld. A survey on fault attacks. In *CARDIS*, pages 159–176, 2004.
- [26] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *In IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

- [27] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. *Lecture Notes in Computer Science*, 1294 :513–525, 1997.
- [28] M. Lomas and al. Low cost attacks on tamper resistant devices. In *Security Protocols, 5th International Workshop, Paris, France*, volume 1361 of *LNCS*, pages 125–136. Springer, 1997.
- [29] Thanh-Ha Le, Cécile Canovas, and Jessy Clédière. An overview of side channel analysis attacks. In *ASIACCS '08 : Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 33–43, New York, NY, USA, 2008. ACM.
- [30] Side channel attacks database.
<http://www.sidechannelattacks.com>.
- [31] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO'96, Santa Barbara, California*, volume 1109, pages 104–118. Springer-Verlag, 1996.
- [32] A. Hevia and M. Kiwi. Strength of two data encryption standard implementations under timing attacks. In *Lecture Notes in Computer Science*, volume 1380, pages 192–205, 1998.
- [33] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *CARDIS*, pages 167–182, 1998.
- [34] Daniel J. Bernstein. Cache-timing attacks on AES.
<http://cr.ypt.to/antiforgery/cachetiming-20050414.pdf>, 2005.
- [35] Stefan Mangard, Thomas Popp, and Maria Elisabeth Oswald. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.
- [36] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *CHES '00 : Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 78–92, London, UK, 2000. Springer-Verlag.
- [37] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21 :120–126, 1978.
- [38] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666 :388–397, 1999.
- [39] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Introduction to differential power analysis and related attacks, 1998.
- [40] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigations of power analysis attacks on smartcards. In *In USENIX Workshop on Smartcard Technology*, pages 151–162, 1999.
- [41] Manfred Aigner and Elisabeth Oswald. Power analysis tutorial. Institute for Applied Information Processing and Communication, University of Technology Graz, 2000.

- [42] Marc Joye, Pascal Paillier, and Berry Schoenmakers. On second-order differential power analysis. In *CHES*, pages 293–308, 2005.
- [43] Thomas S. Messerges. Using second-order power analysis to attack dpa resistant software. In *CHES '00 : Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 238–251, London, UK, 2000. Springer-Verlag.
- [44] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema) : Measures and counter-measures for smart cards. In *E-SMART '01 : Proceedings of the International Conference on Research in Smart Cards*, pages 200–210, London, UK, 2001. Springer-Verlag.
- [45] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis : Concrete results. In *CHES '01 : Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 251–261, London, UK, 2001. Springer-Verlag.
- [46] Medhi-Laurent Akkar. *Attaques et méthodes de protections de systèmes cryptographiques embarqués*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2004.
- [47] David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology Proceedings of Crypto 82*, pages 199–203. Springer-Verlag, 1998.
- [48] John P. Uyemura. *CMOS Logic Circuit Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.
- [49] Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In *CHES '01 : Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 309–318, London, UK, 2001. Springer-Verlag.
- [50] M.-L. Akkar and L. Goubin. A Generic Protection Against High-Order Differential Power Analysis. In *Proceedings of FSE'2003*, LNCS. Springer-Verlag, 2003.
- [51] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In *Cryptographic Hardware and Embedded Systems*, pages 158–172, 1999.
- [52] National Soviet Bureau of Standards. Information Processing Systems. Cryptographics Protection Cryptographic Algorithm. GOST 28147-89, 1989.
- [53] National Institute of Standards and Technology (NIST). FIPS PUB 46-3 : Data Encryption Standard (DES). <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>, October 1999.
- [54] Kumar Sandeep, Paar Christof, Jan Pezl, Pfeiffer Gerd, and Schimmler Manfred. Breaking Ciphers with COPACABANA - A Cost-Optimized Parallel Code Breaker. In *CHES*, 2006.

- [55] IAR Systems.
<http://iar.com/>.
- [56] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. *Avrora : scalable sensor network simulation with precise timing*. In *IPSN '05 : Proceedings of the 4th international symposium on Information processing in sensor networks*, page 67, Piscataway, NJ, USA, 2005. IEEE Press.
- [57] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. *Tossim : accurate and scalable simulation of entire tinyos applications*. In *SenSys '03 : Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.
- [58] Jonathan Polley, Dionysys Blazakis, Jonathan Mcgee, Dan Rusk, and John S. Baras. *Atemu : A fine-grained sensor network simulator*. In *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.
- [59] Simulavr : an AVR simulator.
<http://www.nongnu.org/simulavr>.
- [60] The GNU Project debugger.
<http://www.gnu.org/software/gdb/>.
- [61] J. den Hartog and E. de Vink. *Virtual analysis and reduction of side-channel vulnerabilities of smartcards*. In T. Dimitrakos and F. Martinelli, editors, *Proc. FAST 2004*, page 14pp. Wolters-Kluwer, 2005.
- [62] Den Hartog Verschuren, J. Den Hartog, J. Verschuren, J. De Vos, and W. Wiersma. *Pinpas : A tool for power analysis of smartcards*. In *Proc. SEC 2003, page 5pp. Wolters-Kluwer, 2003. IFIP WG 11.2 Small Systems Security*, 2003.
- [63] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
- [64] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
- [65] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2 edition, June 1996.
- [66] Leroy Xavier, Doligez Damien, Garrigue Jacques, Rémy Didier, and Vouillon Jérôme. *The Objective Caml system release 3.09 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, October 2005.
- [67] Objective caml (ocaml) programming language website.
<http://caml.inria.fr/ocaml>.
- [68] Joshua B. Smith. *Practical OCaml*. Apress, Berkely, CA, USA, 2006.

- [69] Céline Thuillet, Philippe Andouard, and Olivier Ly. A smart card power analysis simulator. *Computational Science and Engineering, IEEE International Conference*, 2 :847–852, 2009.
- [70] ATMEL. Atmega8515(1) datasheet, doc2512 : 8-bit microcontroller with 8k bytes in-system programmable flash.
- [71] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *LNCS*, pages 51–62. Springer, 2003.
- [72] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, pages 16–29, 2004.
- [73] Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Power and electromagnetic analysis : improved model, consequences and comparisons. *Integr. VLSI J.*, 40(1) :52–60, 2007.
- [74] The GNU Privacy Guard.
<http://www.gnupg.org>.
- [75] AVR GCC compiler.
<http://www.nongnu.org/avr-libc/>.
- [76] 8-bit AVR Instruction Set.
www.atmel.com/atmel/acrobat/doc0856.pdf/.
- [77] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX : What You See Is Not What You eXecute. In *VSTTE 2005*.
- [78] Pierre Wolper. *Introduction à la calculabilité*. InterEditions, 1991.
- [79] H. Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1967.
- [80] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.
- [81] Philippe Andouard, Olivier Ly, and Davy Rouillard. VisAA : Visual Analyzer for Assembler. In *CRiSIS '08. Third IEEE International Conference on Risks and Security of Internet and Systems*, pages 221–225, Tozeur, Tunisia, 2008.
- [82] Joshua C. Estep and Christopher A. Healy. A flexible tool for visualizing assembly code. *J. Comput. Small Coll.*, 20(3) :55–67, 2005.
- [83] Patrick Borunda, Chris Brewer, and Cesim Erten. Gspim : graphical visualization tool for mips assembly programming and simulation. In *SIGCSE '06 : Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 244–248, New York, NY, USA, 2006. ACM.
- [84] Ida pro - the interactive disassembler.
<http://www.hex-rays.com/idapro/>.

- [85] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *LCTES/SCOPES '02 : Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 46–55, New York, NY, USA, 2002. ACM.
- [86] Henrik Theiling. *Control Flow Graphs for Real-Time system Analysis Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis*. PhD thesis, Universität des Saarlandes, 2002.
- [87] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [88] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
- [89] Janusz A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4), October 1964.
- [90] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.
- [91] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.
- [92] Richard Y. Kain. *Automata Theory : Machines and Languages*. McGraw Hill, New York, 1972.
- [93] D.N. Arden. Delayed logic and finite state machines. In *Theory of Computing Machine Design*, pages 1–35. U. of Michigan Press, Ann Arbor, 1960.
- [94] Eclipse.
<http://www.eclipse.org>.
- [95] Eric Clayberg and Dan Rubel. *Eclipse : Building commercial quality plug-ins*. Addison-Wesley, Boston, MA, USA, 2006.
- [96] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2) :84, 1997.
- [97] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.
- [98] Zili Member-Shao, Chun Xue, Qingfeng Zhuge, Meikang Qiu, Bin Xiao, and Edwin H. M. Sha. Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software. *IEEE Trans. Comput.*, 55(4) :443–453, 2006.
- [99] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

- [100] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyzer. In *ESOP'05*, 2005.
- [101] Ouvrage collectif - Coordination Philippe Schnoebelen. *Vérification de logiciel - Techniques et outils du model-checking*. Vuibert, 1999.
- [102] Commerce Protocols, Edmund Clarke, Somesh Jha, and Will Marrero. A machine checkable logic of knowledge for specifying security properties of electronic commerce protocols. *LICS Security Workshop*, 1998.
- [103] Source-navigator[™].
<http://sourcnav.sourceforge.net/>.
- [104] Source insight.
<http://www.sourceinsight.com>.
- [105] Understand source code analysis and metrics.
<http://www.scitools.com/products/understand/>.
- [106] Frama-C.
<http://frama-c.cea.fr/>.
- [107] Polyspace.
<http://www.mathworks.com/products/polyspace/>.
- [108] Klocwork.
<http://www.klocwork.com/>.
- [109] Philippe Andouard, Olivier Ly, and Davy Rouillard. Securing embedded code threatened by physical attacks. In *The 10th International Workshop on Information Security Applications (WISA 2009)*, Busan Korea, August 25-27 2009.
- [110] International Organization for Standardization. *ISO/IEC 9899-1999 : Programming Languages—C*, December 1999.
- [111] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil : Intermediate language and tools for analysis and transformation of c programs. In *CC '02 : Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.
- [112] CIL - Infrastructure for C Program Analysis and Transformation (v. 1.3.7).
<http://hal.cs.berkeley.edu/cil/>.
- [113] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7) :107–115, 2009.
- [114] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. Éditions O'Reilly, Paris, 2000.
- [115] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley Professional Inc., One Jacob Way, Reading, Massachusetts 01867, Third edition, 2001.

- [116] Code Warrior. Freescale Semiconductor.
<http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=012726>.
- [117] CalmShine. SAMSUNG.
http://www.samsung.com/global/business/semiconductor/products/microcontrollers/Products_Microcontrollers_SamsungDevelopmentTools.html.
- [118] Keil.
<http://www.keil.com/>.
- [119] GNU gprof.
http://http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html.
- [120] Sun microsystems. Java Card™ Specification.
<http://java.sun.com/javacard/specs.html>.
- [121] Hao Chen Drew, Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of c code. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, 2004.
- [122] MOPS (Model-checking Programs for Security Properties).
<http://www.cs.ucdavis.edu/~hchen/mops/>.
- [123] Frédéric Besson, Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Model checking security properties of control flow graphs. *J. Comput. Secur.*, 9(3) :217–250, 2001.
- [124] Aoraï.
<http://frama-c.cea.fr/download/aorai/aorai-manual-Beryllium.pdf>.
- [125] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna – A Static Model Checker. In *proceedings of 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2006)*. Springer-Verlag, 2006.
- [126] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Model checking software at compile time. In *TASE '07 : Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 45–56, Washington, DC, USA, 2007. IEEE Computer Society.
- [127] Goanna.
<http://redlizards.com/>.
- [128] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2 : An OpenSource Tool For Symbolic Model Checking. In *Proceeding of International Conference on Computer-Aided Verification (CAV 2002)*, pages 359–364, Copenhagen, Denmark, July 27-31, 2002. Springer.

- [129] M. Vistein, R. Huuck, F. Ortmeier, A. Fehnker, and W. Reif. An Abstract Specification Language for Static Program Analysis. In *4th International Workshop on Systems Software Verification (SSV 2009)*. ENTCS, 2009.
- [130] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, 1999.
- [131] Amir Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77)*, pages 46–57, Oct.Nov. 1977.
- [132] Amir Pnueli. The temporal semantics of concurrent programs. *Theor. Comput. Sci.*, 13 :45–60, 1981.
- [133] Leslie Lamport. "sometime" is sometimes "not never" : on the temporal logic of programs. In *POPL '80 : Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185, New York, NY, USA, 1980. ACM.
- [134] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited : On branching versus linear time temporal logic. *Journal of the ACM*, 33(1) :151–178, January 1986.
- [135] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
- [136] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82 : Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 169–180, New York, NY, USA, 1982. ACM.
- [137] Stinson Douglas. *Cryptographie Théorie et Pratique*. International Thomson Publishing France, 1996.
- [138] A. Arnold and I. Guessarian. *Mathématiques pour l'informatique*. Masson, 1994.
- [139] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction à l'algorithmique*. Dunod, 2002.