DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Duy-Phuc Pham

Leveraging side-channel signals for IoT malware classification and rootkit detection

Thèse présentée et soutenue à Rennes, le 13/01/2022
Unité de recherche : IRISA, UMR 6074

Rapporteurs avant soutenance :

Lejla BATINA	Professeur, Université Radboud de Nimègue, Pays-Bas
Davide BALZAROTTI	Professeur, EURECOM, France

Composition du Jury :

Examinateurs :	Yerom-David BROMBERG	Professeur, Université de Rennes 1, France
	Damien COUROUSSÉ	Ingénieur de recherche, CEA-LIST, Grenoble, France.
Rapporteurs :	Lejla BATINA	Professeur, Université Radboud de Nimègue, Pays-Bas
	Davide BALZAROTTI	Professeur, EURECOM, France
Dir. de thèse :	Pierre Alain FOUQUE	Professeur, Université de Rennes 1, France
Co-encadrant de thèse :	Annelie HEUSER	Chargeé de Recherche, CNRS, IRISA, France
	Olivier ZENDRA	Chargeé de Recherche, INRIA Rennes, France

This page intentionally left blank

Acknowledgement

I would like to express my deepest appreciation to my supervisor, Annelie Heuser, for her energy and for giving me the opportunity and freedom to work on this thesis. I would also like to thank Olivier Zendra for the guidance and advice he provided throughout my PhD journey and for encouraging me to finalize my work.

It was a pleasure and a joy to work with great researchers at TAMIS, EMSEC and CAPSULE of INRIA and IRISA laboratory in Rennes. I want to express my gratitude to everyone on the team, especially Damien Marion, for their encouragement and thoughtful comments during my research. I would further like to thank Ronan Lashermes, Matthieu Mastio and Nicolas Aragon for their insightful support.

A special thanks to Pierre-Alain Fouque and Jean-Louis Lanet, who generously accepted me as their PhD student. I very much appreciate that Lejla Batina, Davide Balzarotti, Yerom-David Bromberg and Damien Couroussé accepted to be part of the jury for my PhD defense. Yet none are more influential than Aurélien Francillon and Benoît Gérard who provided valuable comments in my annual CSI meetings.

This PhD thesis would not have been accomplished without my parents and their constant support and encouragement. The only reason I could have completed this thesis was the love and support I received from my wife and son. My greatest thanks go to them.

Lastly, a very special thanks to my friends, Christophe Genevey-Metat, Alexandre Gonzalvez, Cassius Puodzius, Max Kersten, and Quoc-Hung Nguyen. I am really grateful to Adam Zabrocki for insights from LKRG and Albert Spruyt for enlightening me about practical SCA. I really appreciate our discussions and the feedback offered by all of you.

This page intentionally left blank

Table of Contents

This page intentionally left blank

Acronyms

AHMA Automated Hardware Malware Analysis. 61

APT Advanced Persistent Threat. xiv

AV Anti-virus. 41

C&C Command and Control. 24

CNN Convolution Neural Networks. 29, 33, 34

CPU Central Processing Unit. 13

DDoS Distributed Denial-of-Service. 45, 62

DPA Differential Power Analysis. 13

EM Electromagnetic Emanations. i, 1, 12, 14, 16, 122

FN False Negative. 35, 36

FP Palse Positive. 35, 36

FPR Palse Positive Rate. 37

GPU Graphical Processing Unit. 74, 76

IoT Internet of Things. i, xiii, 1–5, 40

LDA Linear discriminant analysis. 21, 22

MITM Man In the Middle. 4

MLP Multi-layer Perceptron. 29, 34

NAS Network-attached storage. 5

NB Naive Bayes. 26

NICV Normalized Inter-Class Variance. 21

OS Operating System. 58

PC Personal computers. 14

RBF Radial Basis Function. vi, 27, 28

ReLU Rectified Linear Unit. 30

ROC Receiver Operating Characteristics. 37

SCA Side-channel Analysis. i, 1, 12, 14

SDR Software-defined radio. i, 1, 12, 18, 19

SMM System Management Mode. 8, 10

SPA Simple Power Analysis. 13

SVM Support vector machines. 22, 27, 28

TN True Negative. 35, 36

TP True Positive. 35, 36

TPR True Positive Rate. 37

ULTRA Ultimate Rootkit Detection over the Air. 90

VM Virtual Machine. 8, 47

List of Figures

List of Tables

Résumé long en français

Un logiciel malveillant ou maliciel, aussi dénommé programme malveillant de l'anglais malware, est un programme développé dans le but de nuire à un système informatique, sans le consentement de l'utilisateur dont l'ordinateur est infecté.

Ils perturbent intentionnellement un dispositif numérique ou un réseau informatique, font fuir des informations privées, obtiennent un accès non autorisé à des informations ou à des systèmes, privent les utilisateurs de l'accès à des informations ou violent la sécurité informatique et la vie privée de l'utilisateur.

La détection des malwares reposant sur des caractéristiques statiques et dynamiques présente encore diverses difficultés, comme les techniques d'empaquetage ou d'obfuscation, ou encore la possibilité d'échapper à la surveillance des bacs à sable. En particulier, les systèmes cyber-physiques embarqués peuvent manquer de ressources ou d'accessibilité pour des outils anti-malware équivalents à ceux utilisés pour défendre les systèmes informatiques et les serveurs.

Un système embarqué est un système informatique qui remplit une fonction spécifique au sein d'un système mécanique ou électrique plus vaste. Il se compose d'un processeur informatique, d'une mémoire informatique et de périphériques d'entrée/sortie. Les dispositifs embarqués sont utilisés depuis de nombreuses années dans les environnements industriels, les appareils critiques, les systèmes de contrôle militaires et le secteur automobile. Cependant, ce n'est que récemment qu'ils ont commencé à envahir toutes les facettes de notre société en raison de la révolution de l'Internet des objets (IoT) et leur nombre et leur complexité augmentent de façon exponentielle. D'ici à

2025, nous devrions compter plus de 64 milliards de dispositifs IoT [RHK20] et d'autres seront produits à mesure que les technologies au-delà du 5G arriveront à maturité. Les entreprises qui fabriquent ces appareils sont constamment en course pour accroître leur part de marché, et donnent donc la priorité à une mise sur le marché rapide combinée à un ensemble de nouvelles fonctionnalités pour attirer de nouveaux clients [CVD+20]. Cela les conduit à remettre à plus tard les questions de sécurité et de confidentialité, car ils utilisent une variété de logiciels et de micrologiciels personnalisés sans tenir compte des problèmes de sécurité, ce qui en fait une cible attrayante pour les cybercriminels.

Ces dernières années, nous avons assisté à une recrudescence de la quantité et de la sophistication des malwares attaquant les systèmes IoT. Les botnets et les outils DDoS côtoient les crypto-mineurs, les malwares, les ransomwares et les menaces persistantes avancées (APT) conçus pour réaliser des cyberattaques IoT. Selon le Symantec Threat Report de 2018 [Ent18], les attaques IoT ont augmenté de 600% entre 2016 et 2017. En 2019 et 2020, le volume des malwares IoT a augmenté de 218% et 66%, respectivement. Le nombre de frappes de malwares IoT en 2021 a atteint 60 1 millions, le plus grand nombre jamais enregistré par SonicWall [Son22] en une seule année, a entraîné une hausse de 6% d'une année sur l'autre. Les chercheurs ont notamment découvert un record mensuel de 10,8 millions d'échantillons de malwares en octobre 2020. Il est évident qu'il y a une augmentation significative des malwares IoT basés sur Linux au cours des dernières années.

Les appareils IoT posent des contraintes de performance, comme une unité de traitement avec une faible fréquence d'horloge, une mémoire limitée et un faible débit. Par conséquent, il est difficile de mettre en œuvre des mesures de sécurité exigeantes en termes de calcul, telles que des solutions antivirus. De nombreux appareils sont dépourvus d'interface de gestion, ce qui rend difficile la mise en œuvre de correctifs de sécurité ou de mises à jour de signatures. En raison du grand nombre d'échantillons de logiciels malveillants IoT dans la nature, il est essentiel que la communauté de la sécurité soit en mesure de construire des approches modernes pour la détection et la classification des logiciels malveillants IoT.

L'objectif de cette thèse est de comprendre les techniques et les comportements des malwares IoT, et d'explorer de nouvelles approches pour utiliser les informations des canaux secondaires afin d'identifier les types de malwares qui ciblent un dispositif embarqué. Les contributions de cette thèse de doctorat sont séparées en deux par-

ties différentes. Premièrement, nous présentons une nouvelle approche qu'un analyste de logiciels malveillants peut utiliser pour recueillir des informations exactes sur le type et l'identité des logiciels malveillants, même en présence de techniques d'obscurcissement qui peuvent entraver l'analyse. Grâce à cette approche, un analyste de logiciels malveillants est en mesure d'obtenir des connaissances précises sur le type et l'identité des logiciels malveillants, même en présence de techniques d'obscurcissement qui peuvent empêcher l'analyse binaire statique ou symbolique. En outre, nous présentons le cadre ULTRA à faible coût, qui est la première solution "wave-and-play", où il suffit d'agiter une sonde sur le dispositif pour voir instantanément quel rootkit est infecté. ULTRA a une spécificité qui facilite la découverte des rootkits dans un système en temps réel sans qu'il soit nécessaire de modifier le dispositif ou d'exiger des logiciels, en surveillant deux types distincts d'appâts qui sont capables d'exposer le comportement des rootkits furtifs.

Liste de publications

Cette thèse est basée sur les travaux de recherche qui ont conduit aux publications évaluées par les pairs des articles suivants : [PMMH21, PMH21, PMH22]. Une partie du matériel créé pour ces articles a été adaptée et réutilisée dans cette thèse. Une autre publication [PVM19] est le résultat d'une collaboration avec un autre doctorant, et ne constitue donc pas une partie principale de cette thèse de doctorat. Cependant, il est important de se référer à cet article pour avoir une compréhension générale des techniques d'évasion des malwares et du contexte de la détection des malwares.

[PMMH21] Duy-Phuc Pham, Damien Marion, Mathieu Mastio, and Annelie Heuser. Obfuscation Revealed: Leveraging Electromagnetic Signals for Obfuscated Malware Classification. In *Annual Computer Security Applications Conference (ACSAC)*, 2021

[PMH21] Duy-Phuc Pham, Damien Marion, and Annelie Heuser. Poster: Obfuscation Revealed-Using Electromagnetic Emanation to Identify and Classify Malware. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 710–712. IEEE, 2021

[PMH22] Duy-Phuc Pham, Damien Marion, and Annelie Heuser. ULTRA: Ultimate Rootkit Detection over the Air. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2022

[PVM19] Duy-Phuc Pham, Duc-Ly Vu, and Fabio Massacci. Mac-A-Mal: macOS malware analysis framework resistant to anti evasion techniques. *Journal of Computer Virology and Hacking Techniques*, 15(4):249–257, 2019

Introduction

"*Malware*" is a portmanteau for malicious software that intentionally disrupt a digital devices or computer network, leak private information, gain unauthorized access to information or systems, deprive users access to information or violate the user's computer security and privacy. Malware detection relying on static and dynamic features still has various difficulties, such as packer or obfuscation techniques, or sandbox monitoring can be evaded. In particular, embedded cyber-physical systems may lack the computing resources or accessibility for anti-malware tools equivalent to those used to defend computer systems and servers.

An embedded system is a computer system that performs a specific purpose within a larger mechanical or electrical system. It consists of a computer processor, computer memory, and input/output peripheral devices. Embedded devices have been utilized in industrial environments, mission-critical appliances, military control systems, and the automotive sector for many years. However, it is only recently that they have begun to pervade every facet of our society due to the so-called Internet of Things (IoT) revolution and they are exponentially growing in number and complexity. By 2025, we are expecting to have over 64 billion IoT devices [RHK20] and more will be produced as technologies beyond 5G mature. Companies that make these devices are constantly racing to grow their market share, consequently they prioritize a quick time-to-market combined with a set of novel features to attract new customers [CVD+20]. This leads to the postponement of security and privacy issues such as they utilize a variety of customized software and firmware without consideration of security concerns, making

them an appealing target for cybercriminals.

We have seen an upsurge in both the amount and sophistication of malware attacking IoT systems in recent years. Botnets and DDoS tools live side by side with cryptomining, malware, ransomware, and Advanced Persistent Threat (APT) designed to perform IoT cyber attacks. According to the Symantec Threat Report of 2018 [Ent18] indicated 600% increase in the IoT attacks from 2016 to 2017. In 2019 and 2020, the volume of IoT malware increased by 218% and 66%, respectively. The amount of IoT malware strikes in 2021 reached 60.1 million, the greatest number ever recorded by SonicWall [Son22] in a single year, resulted in a 6% year-over-year rise. In particular, researchers discovered a monthly record of 10.8 million malware samples in October 2020. It is evident that there is a significant rise in the Linux-based IoT malware in recent years.

Malware detection techniques often rely on static and dynamic malware analysis, with hybrid being some combination of the two, in order to gain information on the input samples. In particular, static malware analysis investigate code features and signatures whereas dynamic approaches rely on executing actual code, typically in some sand-boxed environment or during operational runtime of a system. Static malware analysis can be trivially evaded by obfuscation techniques such as code metamorphism. Dynamic malware techniques based on software level, on the other side, are OS, resource, and architecture dependent and prone to sandbox evasion. The use of traditional analysis techniques to IoT malware and IoT systems has its limitations. IoT devices pose performance constraints, such as processing unit with low clock rate, limited memory, and low throughput. Therefore, it is difficult to implement security measures that are computationally demanding, such as antivirus solutions. IoT devices lack a universal management interface, making them difficult to implement frequent security patches and updates. Because of the large number of IoT malware samples in the wild, it is critical for the security community to be able to build modern approaches for IoT malware detection and classification.

Malware detection techniques for the IoT devices at the software layer are prone to performance constraints and are vulnerable to sophisticated malware evasion strategies. With advancements in the field of Side-channel Analysis (SCA) to perform side-channel attack, a new area of research has emerged in malware analysis techniques that is coming closer to the hardware perspective. In computer security, a Side-channel

Analysis (SCA) is any analysis based on information gained from the implementation of a computer system, rather than vulnerabilities in the implementation per se (*e.g.* cryptanalysis). Side-channel attacks have been first demonstrated via various leakage channels based on statistical methods (such as [Koc96, KJJR11, KJJ99]). In fact, TEMPEST attacks [NAC82] have historically been used to gather information about systems by leaking emanations, which include unintentional radio or electrical signals, sounds, and vibrations. A common idea to take advantage of side channel information such as Electromagnetic Emanations (EM) and energy consumption in anomalies and malware detecion context is to observe how the system behaves in its normal state, and to raise an alert when a abnormal behavior is triggered. While it is feasible for malware developers to obfuscate malicious function calls and behaviors to evade software detection, modifying their device's power consumption or EM pattern is less likely and more difficult to achieve [ADCC18].

In the malware analysis context, there are typically two fundamental tasks: malware detection and malware classification. The term "malware detection" can refer to either the process of determining whether or not a particular input binary is malicious or benign, or it can refer to the detection of the presence of malicious applications on a host system. On the other hand, malware classification is the process of assigning a malware sample to a specific malware family. Malware within a family has features in common that may be used to create signatures for further detection and classification. In particular, signatures can be static or dynamic depending on how their features are collected by malware analysts. In general, malware detection solutions showed high detection accuracy, but they were unable to correctly classify malware into specific categories since their detection techniques did not take into account the characteristics of each malware category [KKB+21]. Malware analysts are generally able to fully control and customize their analysis environment and device in the most advantageous way in malware classification, whereas in real-time IoT malware detection, the target environment requires stability and high availability, so interruption and downtime should be kept to a minimum.

We are able to address both scenarios of IoT malware classification and detection in this thesis. The goal of this thesis is to understand the techniques and behaviors of IoT malware, and to explore novel approaches of using side channel information to identify malware and rootkit that are targeting an embedded device. So that the

contributions of this PhD thesis can be separated into two different parts that solve the two aforementioned problems of IoT malware detection and classification. First, we present a novel approach that a malware analyst can use to classify IoT malware dataset. Using this approach, a malware analyst is able to obtain precise knowledge about malware type and identity, even in the presence of obfuscation techniques which may prevent static or symbolic binary analysis. Furthermore, we present the low-cost ULTRA framework, which is the first wave-and-play solution, where one can simply wave a probe over the device to instantly see what rootkit is infected. ULTRA is the ultimate goal of this thesis in which malware analysts can leverage to detect sophisticated rootkits on online embedded IoT devices. ULTRA has a specification that facilitates the discovery of rootkits in a specific system in real-time, without the need for device alteration or software requirements by monitoring two distinct types of hardware and software *baits* that are capable of exposing the behavior of stealthy rootkits.

Thesis outline

This thesis is organized as follows. In Chapter 1, we recall definitions and presentation of the necessary subjects of IoT security, malware, and side-channel analysis. We further introduce the background of detection and classification techniques by using machine and deep learning. In Chapter 2, we shall present literature research studies which related to IoT malware analysis evasion techniques with focuses on side-channel solutions. In chapter 3, we describe AHMA - an automated IoT malware classification framework using oscilloscope to monitor side-channel signals. Additionally, this chapter mentioned particular obfuscation techniques to investigate the evasion capability to side-channel method. As described in Chapter 4, we propose a low-cost ULTRA framework which is the first wave-and-play solution, where one can simply wave a probe over the device to instantly see what rootkit is infected.

Publications

This thesis is based on the research work that led to the peer-reviewed publications of the following papers: [PMMH21, PMH21] for AHMA and [PMH22] about ULTRA. Parts of the material created for these articles have been adapted and reused in this work. Another publication [PVM19] was the result of a collaboration with another doctoral student, and hence is not a main part of this PhD thesis. However, it is important to refer to the paper for a general understanding of malware evasion techniques and malware detection background.

[PVM19] Duy-Phuc Pham, Duc-Ly Vu, and Fabio Massacci. Mac-A-Mal: macOS malware analysis framework resistant to anti evasion techniques. *Journal of Computer Virology and Hacking Techniques*, 15(4):249–257, 2019

[PMH21] Duy-Phuc Pham, Damien Marion, and Annelie Heuser. Poster: Obfuscation Revealed-Using Electromagnetic Emanation to Identify and Classify Malware. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 710–712. IEEE, 2021

[PMMH21] Duy-Phuc Pham, Damien Marion, Mathieu Mastio, and Annelie Heuser. Obfuscation Revealed: Leveraging Electromagnetic Signals for Obfuscated Malware Classification. In *Annual Computer Security Applications Conference (ACSAC)*, 2021

[PMH22] Duy-Phuc Pham, Damien Marion, and Annelie Heuser. ULTRA: Ultimate Rootkit Detection over the Air. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2022

This page intentionally left blank

1 Background

Today, malware rises and plays a crucial role in technology's vulnerabilities, from financial transactions to IoT devices in our "smart" homes, and even to potential "cyberwar" such as energy, and transportation infrastructures [Dwy19]. Malware detection has become critical to safeguarding the internet economy, as well as state stability, security, and wealth. In this chapter, we recall the definitions needed within its related domains. This encompasses the presentation of the necessary subjects of IoT security, malware, and side-channel analysis. We further introduce the background of detection and classification techniques by using machine and deep learning.

Contents

1.1 IoT, embedded devices, and their security challenges

IoT devices, which comprise physical objects that are embedded with sensors, software, actuators, and computer devices, connect and exchange data with other devices and systems over the Internet or other communications networks. It allows data to be sent between things or people automatically and without the need for human interaction. It is common that IoT devices are embedded devices that may have constrained resources, limited battery or restricted accessibility. Consequently the use of security policies and processes to reduce the overall risk or impact of cybersecurity threats is not fully implemented on them.

There are several concerns about the risks associated with the growth of IoT devices and technologies, particularly in the areas of security and privacy. As a result, industry and government efforts to address these concerns have begun, including the development of international and local standards, guidelines, and regulatory frameworks [otCTO21]. Moreover, the network connectivity exposure, diversity of hardware architecture, and diverse operating platforms, all expand the attack surface and target field for malicious actors. The threat likelihood even rises significantly when the attacker takes advantage of less secure, vulnerable or exploited devices to launch a massive-scale attack on critical infrastructure (*e.g.* Mirai DDoS attack [AAB+17]). Malware encouraged the response from both academia and industry to keep digital devices "clean" from malware by creating "anti-virus", and later "endpoint detection" products that are installed on billions of computing devices worldwide [Dwy19]. Anti-malware solutions identify, prevent and eliminate these harmful intruders; and in the process, define, articulate, inform, and detect what is malicious.

Computers and embedded systems, from medical devices to smart homes and industrial controllers, are increasingly connected to the Internet. By 2025, we are expecting to have over 64 billion IoT devices [RHK20] and more will be produced as beyond 5G technologies mature. The connection of these devices can expose the devices to malware and malicious code. Many devices are incompatible with anti-malware software because they run custom firmware or manufactures may forbid end users from installing updates or anti-malware software on the device. For example, the device manufacture may only certify the safety or reliability of a device as long as the end-user does not modify the device by installing third-party software (which includes anti-malware software). [MSB22] discussed many open issues and difficulties with

tools and techniques for IoT malware analysis, and concluded that the research community as a whole must solve those shortcomings.

In the following subsections, we will discuss the IoT attack vectors as well as IoT malware detection challenges.

1.1.1 IoT attack vectors

An IoT attack vector is a specific path, method, or scenario that can be exploited to break into an IoT system, thus compromising its security. [ACH15] classified IoT attacks under four distinct classes: physical, network, software, and encryption attacks, excluding environmental attacks (earthquakes, etc.). An attack surface in IoT is defined as the device vulnerabilities, including both software and hardware vulnerabilities, and vulnerabilities of the associated network infrastructure.

In this subsection, we will discuss some of the attack vectors that can lead to the infection of malware on IoT devices as follows:

- **Internal actors**: An attacker compromises IoT system by physically injecting malicious code into it, granting administrative access to the IoT system; *e.g*, a scenario where an internal actor plugging a USB device with malicious software into the device. Hence, controlling all data flow from and to the device and ensuring its operation. (*e.g.* SCADA malware BlackEnergy 3 attack on the Ukraine grids [Sha16])
- **Weak Authentication**: IoT manufacturers often provide an easy-to-follow installation manual where they provide a basic username and password for logging in and configuring the device, that can be exploited brute-force attack or dictionary-based attack. Because the majority of users do not change their default credentials, weak authentication becomes an attack vector for IoT attackers before granting higher privileges to execute malicious code. Moreover, on shared IoT devices, vendors often increase authentication vulnerabilities through implementation flaws in device access control schemes [JCO20].
- **Hardware/Software vulnerabilities**: An attacker compromises IoT system by exploiting hardware or software vulnerabilities and executing malicious code, further to grant higher privilege on the system and to install malware binary. This attack vector is exploited by 0-day vulnerabilities or when users do not keep their firmware updated. Ubuntu's survey [Rou16] worryingly revealed that only 31%

of consumers that own connected devices perform updates as soon as they become available. A further 40% of consumers have never consciously performed updates on their devices. Nearly two thirds felt that it was not their responsibility to keep firmware updated, and 22% believed it was the job of software developers, while 18% considered it to be the responsibility of device manufacturers. However, most IoT manufacturers do not prioritize security and can not finance the efforts. Furthermore, some hardware vulnerabilities can not be patched unless the user replaces the device with a newer model (*e.g. checkm8* vulnerability affects all devices from iPhone 4s to iPhone X devices and supports any iOS version running on these supported devices). [KVH+20] found that IoT malware increasingly relies on exploiting vulnerabilities as part of their infection strategy, and recent malware is much more likely to exploit multiple vulnerabilities.

- **Physical device tampering**: The IoT device can be tampered during supply chain progress, and threat actors has gotten into the printed circuit board. For example, a known electronic device can be altered at the hardware fabrication level for malicious purposes, such as accessing stored data or eavesdropping on users' activities; or a genuine electronic device can be replaced by a counterfeit electronic device for a malicious objective [SLKS19a]. It is also known as "hardware trojan" [TK10].
- **Insecure Communication**: Secure communication across the network requires the use of cryptography. IoT devices, however have limited resources and are not equipped with high-performance cryptography computing for secure communication [MSB22]. Implementing strong encryption schemes is a low priority since it raises the cost of the production. Insecure communication can lead to Man In the Middle (MITM) attack, where the attacker over the network manages to interfere between two sensor nodes, accessing restricted data, violating the privacy of the two nodes by monitoring, eavesdropping and controlling the communication between the two sensor nodes. Furthermore, they can interfere the device remote update/upgrade process to execute malicious code.
- **Social Engineering** Social engineering attacks, rather than attacking the technical weakness of the system, are known to target the human-computer reaction to allow attackers to persuade a user into completing an action that compromises a system information security. The attacker manipulates users of an IoT system in order to obtain private information or to perform actions to achieve his ob-

jectives. Until recently, social engineering in computer systems was only limited to traditional Internet interactions such as email and website platforms. However, as the threat landscape for the Internet of Things grows, the impacts of this deception-based attack will no longer be limited to traditional cyberspace, but can potentially result in physical impact, ranging from industrial plants to nuclear power plant destruction, as in the case of Stuxnet [FMC11].

- **Device exposure to the Internet**: Some manufacturers of embedded IoT devices keep a few unusual ports and unauthenticated services open for remote access or to update the firmware periodically. Consumers expose these services publicly on the public network, so that the attacker takes advantage to exploit the device or local network using these services.

1.1.2 IoT malware detection challenges

IoT malware analysis and detection have become increasingly vital, but also more difficult, as a result of of rapidly growing IoT technologies, the inherent attack vectors, and the sophistication of malware attacks. In this subsection, we will discuss four of the IoT malware analysis and detection challenges:

- **Detection at edge**: In general, the malware detection and classification training model based on machine learning requires a heavy computing infrastructure. Embedded devices, however, may have limited resources and are not capable of running anti-malware software. Therefore, side-channel malware detection and cloud-based malware detection are potential solutions.
- **Malware dataset availability**: One important challenge is gathering IoT malware dataset as well as creating a honeypot solution to reproduce specific vulnerabilities. Since it can be difficult to deploy a specific version of the software combined with the hardware needed to carry out a specific environment to reproduce the vulnerability. Another difficulty is finding an IoT malware dataset from publicly available sources.
- **Architecture diversity**: IoT devices are diverse in terms of instruction set architecture (Intel, ARM, MIPS, MIPSEL, PPC, etc.), operating system (different distributions of Linux Operating System), devices (smart-home, wearables, routers, medical, surveillance, NAS, etc.), and different versions of headers and libraries

for execution. These variations make it difficult to have an universal solution for IoT malware detection.

- **Inadequate dynamic malware analysis environment**: A variety of dynamic sandbox solutions are available for analyzing Linux malware. However, without root privilege, many IoT malware behave differently [MSB22]. It is not trivial to grant a malicious program root privilege, while avoiding the sandbox evasion techniques as well as persistent techniques that are used by IoT malware.

1.2 IoT malware

This section introduces foundations of generic malware, rootkits and obfuscation.

1.2.1 Generic IoT malware

While general-purpose computers are often built around common architectures such as the x86, embedded IoT devices are being developed within a diverse variety of architectures such as ARM, MIPS, PowerPC, etc. as well as different endianness types such as little or big endian. A discussion of the major issues associated with the diversity of possible target IoT environments can be found in [CGFB18].

Any sort of attack (malware or otherwise) requires the attacker to reach an attack surface. When an attacker discovers the attack surface, they develop an attack vector (outlined in Section 1.1.1), which is the path the attacker takes to locate and exploit susceptible IoT devices on victim's network, causing the device to do something other than what it was designed to do. In general, IoT malware has several characteristics, including the following: IoT malware scans and exploits open ports and services such as SSH, FTP, or Telnet; IoT malware is used to perform DDoS attacks; IoT malware performs brute-force attacks to gain access to other devices.

Cross-compiling their own malware source code into multiple architectures is a well-known technique used by IoT malware developers to get across the target diversity problem in their malware attacks. They infect the target device with all the compiled malware binaries at the same time. As a result, there will be only one binary that is successfully executed within the compatible architecture on the embedded device.

1.2.2 Malware obfuscation

Malicious codes commonly use packers, obfuscators, and polymorphism to hinder static-analysis and evade detection by making analyses difficult to reverse-engineer. Binary file obfuscation is defined as any attempt to conceal the true meaning or behavior of binary software [EN20]. Obfuscation can be used by programmers for a variety of reasons. For example, the protection of proprietary algorithms or the obscuring of harmful intent. There are numerous tools available to help programmers create obfuscated binaries. Obfuscation is used by nearly all types of malware to hinder analysis.

Collberg *et.al* [CTL97] defines obfuscating transformation as follows:

Definition 1: Obfuscating transformation

Let $P \xrightarrow{T} P'$ be a transformation of a source program P into a target program P'. $P \xrightarrow{T} P'$ is an *obfuscating transformation* if

- P and P' have the same observable behavior,
- P' is harder to analyze than P, and
- P' is no more than polynomially slower than P.

More precisely, in order for $P \xrightarrow{T} P'$ to be a valid obfuscating transformation, the following conditions must hold: if P fails to terminate or terminates with an error condition, then P' may or may not terminate. Otherwise, P' must terminate and produce the same output as P.

Previous research classifies code obfuscation schemes into three main categories: data obfuscation, static code rewriting, and dynamic code rewriting. We introduce a collection of obfuscation transformations with static code rewriting that consists of *Opaque predicates*, *Bogus control flow*, *Instructions substitution* and *Control-flow flattening*, and dynamic code rewriting such as *Packer* and code *Virtualization* as follows.

Opaque predicates break up code blocks by inserting obfuscated predicates with an outcome that is difficult to reverse-engineer without running the binary. *Bogus control flow* adds a basic block before the current basic block. This new basic block contains an opaque predicate and then makes a conditional jump to the original basic block, which is also filled with random junk instructions. In contrast, *Control-flow flattening* obscures links between basic-blocks by flattening the control-flow. In general, it puts the basic-blocks of a program into one large switch-statement and a $next$ variable to keep track

of the next block to jump to. *Instructions substitution* replaces standard operators (*e.g.* call, addition, subtraction, or boolean operators) by semantically equivalent but more complicated sequences of instructions.

On dynamic code obfuscation techniques, for instance, *Virtualization* turns a function into an interpreter, whose bytecode language is customized for this function. *Packer* hides code by encoding or encrypting its executable sections that make its result difficult to be interpreted by static analysis. The unpacking routine turns this data back into original machine code at runtime.

Over the last few years, there has been an arm race between programmers who want to keep their code secret, malware authors who hinder their malicious code, against reverse engineers and malware analysts. While there are numerous existing packers for the PE format, only a limited number of ELF packers have been published. There are fewer packers for ARM than for the ELF Intel architecture, and most of them are proof-of-concept projects. The only exception is UPX, a popular open source packer to compress the size of executables.

1.2.3 Rootkits

Rootkits exist to enable long-term covert access to a system so that they can be managed and monitored remotely in an undetectable manner. It is a program that is created to give escalated permissions to a computer whilst also hiding its presence. However not everything that comes to be seen as malicious rootkit in this way, as Sony's Digital Rights Management system demonstrates: Its system installed software (a rootkit) to stop replication of copyrighted content, quickly considered synonymous with malware as it illegally installed its rootkit on the victim machines without their knowledge [Tou16].

In common, types of rootkits are divided into two main categories: user mode and kernel mode rootkits [Bun04, HL07] regarding the level (ring) of privileges obtained *(e.g.* User-space rootkits: beurk [UT17], vlany [mem19] and kernel-space rootkits: diamorphine [m0n21], spy [Jan21], maK_it [McN15]). Some rootkits are designed to perform both modes of operation and thus work at both levels.

[Blu12] discussed further 2 rootkits living beyond kernel level: System Management Mode (SMM) based rootkits and Virtual Machine (VM) based rootkits. System Management Mode (SMM) is a special-purpose operating mode on x86 and x86-64

processors, intended for use by firmware or BIOS to perform low-level system management operations while an OS is running. The primary advantage of SMM is that it provides a distinct and easily isolated processor environment that functions transparently to the operating system and software applications. Examples of SMM rootkits rootkit are [ESZ13] and chipset rootkit Thunderstrike [HR15].

A hypervisor is a low-level layer of software that enables a computer to share its resources in order to run more than one operating system at the same time. A hypervisor is also known as a virtual machine monitor. The main idea behind a hypervisor rootkit is that it could secretly install itself as a hypervisor, conceal the currently running operating system, which is initially running on bare metal, and trap it within a virtual machine. The rootkit can then stealthily modify the operating state from the outside. Examples of hypervisor rootkits are BluePill [Rut06] on x86, and rHV [BVN16] on ARM).

Commonly, a rootkit is designed to hide running modules and processes, mask the existence of files, directories, or users, hide network activities, and capture keystrokes.

System protection levels

To understand the attack surface of rootkits, this subsection introduces the background of system protection levels. The x86 and x64 processor architectures provide 4 privilege levels (or protection rings) to prevent system code and data from being (maliciously) modified by lesser privileges. The privilege levels, as illustrated in Fig.1.1, range from 0 (most privileged) to 3 (least privileged), with memory, I/O ports, and the ability to execute particular machine instructions being protected. Furthermore, [Blu12] discussed further 2 protection levels which are beyond the kernel level: hypervisor and System Management Mode (SMM) based lelves. An x86 CPU's privilege level governs what programs can and cannot do.

Windows only employs privilege ring 0 and 3 for kernel and user modes, respectively. OS code (such as system services and device drivers) executes in kernel mode. Kernel mode allows access to all system memory and all CPU commands. Windows utilizes just two levels because some hardware architectures, like ARM and MIPS in the past, only supported two levels [YSI17].

The earlier ARM architecture defined seven execution modes where a privilege level 0 (PL0) device is unprivileged, whereas the other six are privileged (PL1). Recent ARMv8 and ARMv9 unify the privileged execution modes, simplifying exception and

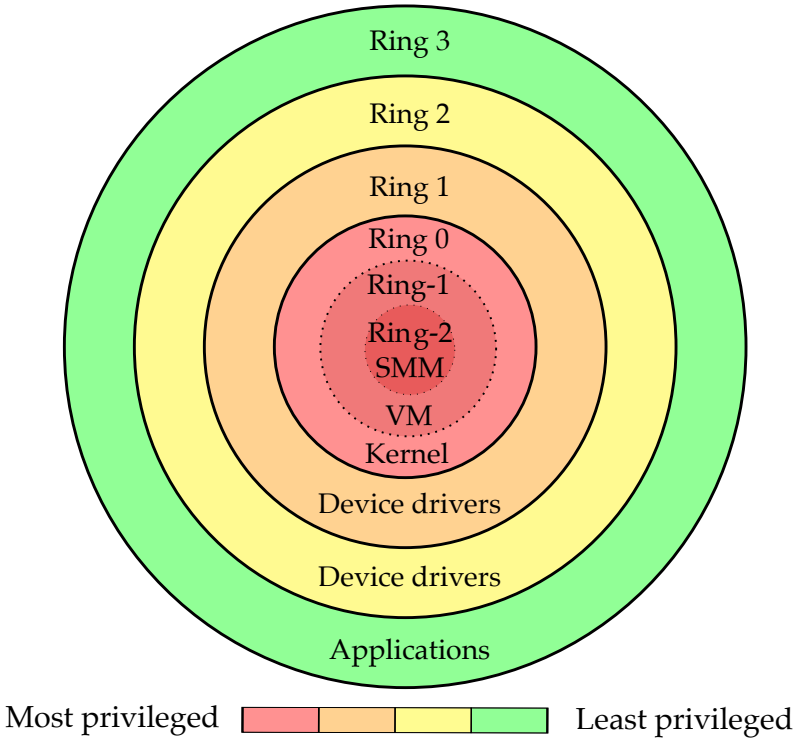


Figure 1.1 – x86 and x64 CPU privilege levels

interrupt handler code. They also modify the nomenclature by changing PL0 to EL0 (exception level) and PL1 to EL1, but maintain the numbers and meaning the same. As shown in Figure 1.2, the exception levels are referred to as ELx, with x as a number between 0 and 3. For example, the lowest level of privilege is referred to as EL0.

As Joanna Rutkowska once stated, "SMM rootkits sound sexy, but, frankly, the bad guys are doing just fine using traditional kernel mode malware (due to the fact that AV is not effective)"[Rut09]. The following subsections introduce the user-level and kernel-level rootkits as far as needed to understand Chapter 4.

User-level rootkit

User-level rootkits operate in the user space and hence do not access the kernel. An example of this type of rootkit is the replacement of the OS's important programs such as *ls, ps* and *login* with altered code that filters standard output according to criteria given by the attacker. Recent user-level rootkits replace or override functionalities in dynamically linked libraries. They manipulate the mechanism of the dynamic linker,

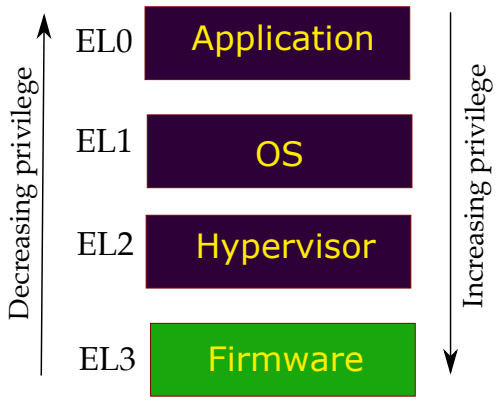


Figure 1.2 – ARM privilege and exception levels

or preload loader to intercept calls to library functions and manipulate their execution. For example, such rootkits may load a malicious DLL into the memory space of a user process, such as explorer.exe in Windows, or exploit *LD_PRELOAD* in Linux, in order to operate in the background. In comparison to kernel-level rootkits, user-level ones often offer richer features and are commonly used in mass attacks since they are easier to develop than kernel-mode rootkits as the design requires less precision and knowledge.

Kernel-level rootkit

Kernel-level rootkits are usually injected into the kernel similar to Loadable Kernel Modules (LKM), which allow rootkits to modify the kernel without having to recompile it, thus can be installed and uninstalled on the fly. They use various techniques to accomplish their goals, such as: syscall hooking, function pointer hooking, direct kernel object manipulation, etc. In common, kernel-level rootkits are more sophisticated and targeted since they are not trivial to detect as well as develop, and any errors in execution can cause systems to panic, which will reveal the intrusion and allow the attack to be thwarted. Such rootkits are not trivial to develop, since any errors in execution can cause system to panic, which will reveal the intrusion and allow the attack to be thwarted. Therefore, kernel-level rootkits are often seen from strategic groups that have sufficient technical qualifications and financial capabilities , such as APT groups that care for information theft, or carry out destructive actions regardless of cost, or financial motivation [pts21].

1.3 Side-channel analysis

A correct implementation of a strong protocol is not guaranteed to be secure. For example, failures can be caused by defective computations and information leaked during protected operations via side-channel. In computer security, a side-channel attack is any attack based on information gained from the execution on a system, rather than weaknesses in the implemented algorithm itself (*e.g.* software remote code execution) or in the underlying mathematical problem (*e.g.* cryptanalysis). In fact, Side-channel Analysis (SCA) must be distinguish from classical cryptanalysis that are purely mathematical while SCA rests on side-channel information. Timing information, power consumption, Electromagnetic Emanations (EM), or sound can provide a side-channel source of information, which can be exploited.

Side-channel attacks have been demonstrated on numerous cryptographic implementations and via various leakage channels that are based on statistical methods pioneered (see [Koc96, MOP07, KJJR11]). Some side-channel attacks require technical knowledge of the system's internal operation, while others, such as differential power analysis, work as black-box analysis. Side-channel attacks do not include attempts to crack a cryptosystem by human deception or coercion with legitimate access, such as social engineering.

The main concept of SCA is to utilize side channel information to recover (secret) information leaked during normal operations. More specifically, in the context of this research, the malware or device activities without analyzing the software algorithm. In the following subsections, we first introduce the background of SCA, then EM leakage, and SDR last.

1.3.1 Introduction to Side-channel Analysis (SCA)

[ZF05] divided SCA into passive attacks and active attacks. Passive attacks are those that do not interfere with the operation of the target system. On the other hand, the adversary in an active attack exerts some influence on the behavior of the target system to gain information about the target operation. At least ten kinds of significant SCA have been explored. For each computation an electronic device is processing, it generates a set of residual unwanted productions (Fig. 1.3) that are potentially leaking sensitive information about its internal state. The following are some classes of side-channel attacks in general:

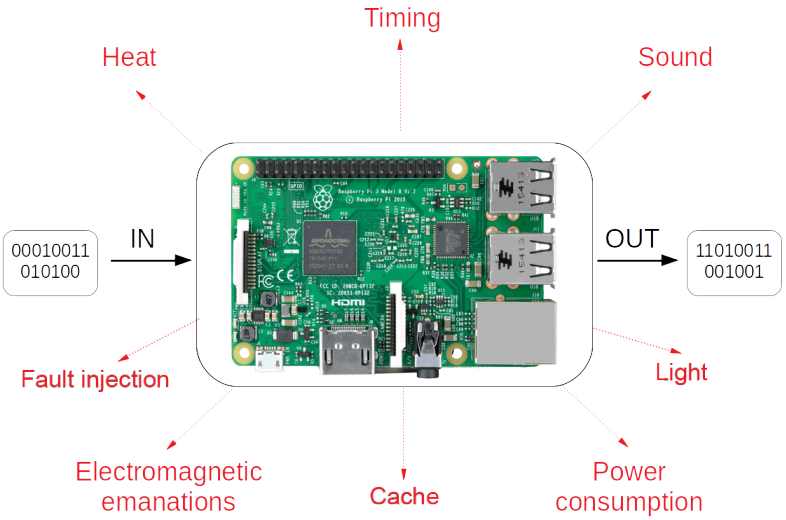


Figure 1.3 – Examples of physical side channels information

- **Timing attacks** are based on calculating how long different computations take such as comparing an attacker's provided password to the victim's unknown password. Timing attacks on critical cryptography implementations, such as find fixed Diffie-Hellman exponents, factor RSA keys, and break Digital Signature Standard (DSS) cryptosystem were first introduced in 1996 by Kocher [Koc96].
- **Cache attacks**: The attacker is able to track the victim's cache accesses on a microarchitecture level, in a virtualized environment, a shared physical system, or a cloud service. Using this side-channel, an attacker may be able to disclose or narrow the possible values of secret information stored on the target device [Pag02]. Most of the recent devices employ a cache between the Central Processing Unit (CPU) and main memory to speed up computing accesses. If the CPU accesses data that is not ready in the cache, a delay will be generated, *i.e.* cache miss, since the target data must be loaded from main memory into the cache. This delay may enable attackers to determine the occurrence and frequency of cache misses leaking information. For example, Bernstein [Ber05] demonstrates complete AES key recovery from known-plaintext using cache timing attack.
- **Power consumption**: Attacks that take advantage of the hardware's power usage during its computation. It was first introduced in 1999 by Kocher [KJJ99]. Power consumption attacks can be divided into Simple Power Analysis (SPA) and Dif-

ferential Power Analysis (DPA). The goal of SPA attacks is to deduce from one power trace which behavior is being executed at a given time and what values the input and output contain. As a result, in order to mount such an attack, the adversary needs exact knowledge of the implementation. DPA attacks, on the other hand, work on black-box scenarios and instead rely on statistical methodologies in the analysis process. Power consumption attacks can also be divided into supervised and unsupervised attacks.

- **Electromagnetic Emanations (EM) leakage**: Attacks based on electromagnetic emanations that can directly provide information leaked from the target device. The electromagnetic side channel research has been first conducted for attacking smart cards, FPGA and other small devices (e.g. [QS01, GMO01, AARR02]). On Personal computers (PC), [ZP14] observed electromagnetic leakage of processor-memory systems from laptops and desktop computers, and the predicted activities can reliably be received at distances that vary from tens of centimeters to several meters, including the signals that have propagated through cubicles or structural walls. [GPT15] demonstrated successful EM attacks on a side-channel protected PC implementation of the square-and-multiply modular exponentiation algorithms, to achieve RSA and El-Gamal key extraction.
- **Fault attacks**: Information is discovered by introducing faults in a computation from software or hardware levels. Generally, a fault model should at least specify the following characteristics according to [ZF05]: *(i)* The precision an attacker can reach in choosing the time and location on which the fault occurs during the execution of a cryptographic module. *(ii)* The length of the data affected by a fault; for example, only one bit, or one byte. *(iii)* The persistence of the fault; whether the fault is transient or permanent. *(iv)* The type of the fault; such as flip one bit; flip one bit, but only in one direction (e.g. from 1 to 0); byte changed to a random (unknown) value; and so on.

In the following subsections, only SCA through EM is considered in this thesis.

1.3.2 Electromagnetic Emanations (EM) leakage

History of EM leakage

The ability to leverage electromagnetic emanations has long been recognized in military circles. For example, the National Security Agency's declassified TEMPEST

files [NAC82], which explore several compromising emanations such as electromagnetic radiation, line conduction, and auditory emissions. The TEMPEST attack is not one that is only confined to cryptographic devices, it is a system problem and is of concern for all equipment which process security data to spy on information systems through leaking emanations, including unintentional radio or electrical signals, sounds, and vibrations. It originally referred to a classified US government program aimed at studying such emission security (EMSEC) concerns and developing protection standards.

During World War I, the German army used the earth loop current of allied battlefield phone lines to successfully eavesdrop on enemy voice communication [Bau99]. To reduce the weight of cable drums that the signal troops had to carry, only a single insulated wire was utilized to connect field phones at the time. Eavesdroppers were able to pick up the resulting voltage drop using valve amplifiers coupled to well-spaced ground spikes since the return current went through the ground. The French and British forces became aware of the problem in 1915 and they finally put in place defenses including installing earth connections hundreds (and then thousands) of meters behind the front trenches, using twisted-pair cables, reducing line currents, and minimizing the sensitivity of information relayed via field phones. In 1985, van Eck [VE85] demonstrated that the screen content of a video display unit could be reconstructed at a distance using low-cost home-built equipment, namely a TV set with manually controlled oscillators in place of sync-pulse generators, brought the concept to the attention of the public.

EM leakage foundations

The components of electrical devices, frequently emit electromagnetic radiation as part of their operation. An adversary who can obtain these emanations and deduce their causal relationship to the underlying, may be able to deduce a surprising amount of information about the computation and data. EM analysis can also be classified into two types, similar to power analysis attacks: Simple EM Analysis (SEMA) and Differential EM Analysis (DEMA). The EM fields are produced when current flows through a conductor, and are composed of an electric field (E-field) component and a magnetic field (H-field) component. The two are mutually dependent, such that no moving E-field can exist without an associated H-field. They radiate from a conductor roughly traveling at right angles to each other (see Fig. 1.4).

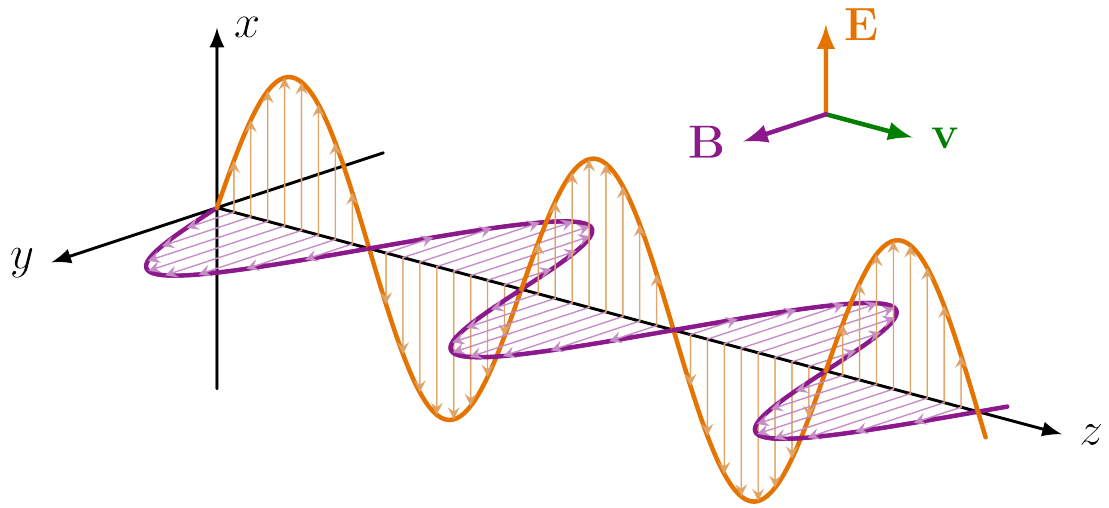


Figure 1.4 – A sinusoidal electromagnetic wave propagating along the positive z-axis, showing the electric field (E) and magnetic field (B) vectors.

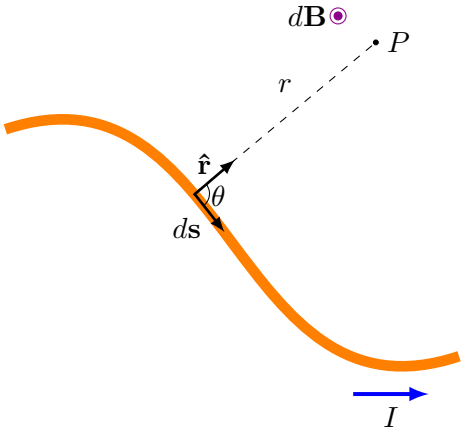


Figure 1.5 – The magnetic field $d\mathbf{B}$ at a point due to the current I through a length element $d\mathbf{s}$ is given by the Biot–Savart law. The direction of the field is out of the page at P.

To determine the total magnetic field generated by an electric current, the Biot-Savart law can be applied (illustrated in Fig. 1.5) as follows:

$$d\mathbf{B} = \frac{\mu_0 I}{4\pi} \int_C \frac{d\mathbf{s} \times r}{|r^3|} \qquad \text{(Eq. 1.3.1)}$$

Where the magnetic field $d\mathbf{B}$ at position $\hat{\mathbf{r}}$ is produced by a static electric current I, the magnetic constant is μ_0 (permeability of free space) and $d\mathbf{s}$ is a vector whose magnitude

is the length r. As variations in the magnetic field occur, an electromotive force *emf* is generated within the H-field, according to Faraday's law of induction:

$$\texttt{emf} = -\int_S \frac{d\mathbf{B}}{dt}.dS \qquad \text{(Eq. 1.3.2)}$$

Where, S is the surface area of the loop at the end of the H-field probe. In Fig. 1.6 a stationary conducting loop is in a time-varying magnetic B field. This `emf` induced by the time-varying current (producing the time-varying B-field) in a stationary loop is referred to as transformer `emf` in power analysis.

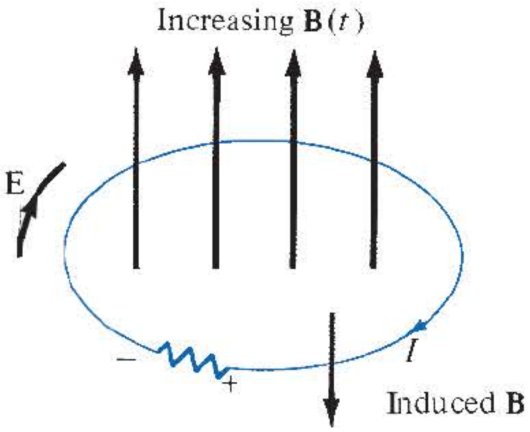


Figure 1.6 – Induced emf due to a stationary loop in a time-varying B-field [SN01].

EM leakage acquisition

In order to capture this EM information, near-field magnetic H-field or electric E-field probes can be utilized (*e.g.* Fig. 1.7). An H-field probe has a conductive loop with a coil at the endpoint, and is usually used to detect magnetic fields produced by clock signals, serial data streams, control signals, and switching power supplies. While E-field probes often make direct contact with circuits to find emissions on individual pins or PCB traces. Because radiated emanation levels can be incredibly low in many circumstances, a pre-amplifier may be required between the probe and the scope. Otherwise, a sensitive scope or internal pre-amplifier is required if a pre-amplifier is not used. When it comes to EM leakage acquisition in our research, selecting the right probe and placing it correctly are critical keys that will be discussed later.

Figure 1.7 – Near field probe set of 4 Langer H-field probes (left) and one E-field probe (right).

Magnetic field probes (H-field probes) are generally formed into a loop. When the loop is 90 degrees to the signal, or when the magnetic field is "flowing" through the loop, maximum response occurs. When the loop is parallel to the signal, the response is the weakest. The H-field probe is typically positioned close to the target device, allowing the magnetic field to induce a voltage through it. An oscilloscope and other devices such as Software-defined radio (SDR) can measure this voltage. There is a trade-off between resolution and sensitivity when it comes to loop size: A large loop is more sensitive, but its spatial resolution is lower. While a smaller loop is less sensitive, but it makes it easier to identify the location of the signal.

Electric field probes (E-field probes) When E-field probes are oriented parallel to the observed electric field, they respond at their maximum. Because the E-field of most conductors is perpendicular to the conductor's surface, E-field probes are held perpendicular to the tested conductors. Electric fields emanating from components with larger surface areas are measured with large area probes. The probe's top is electrically shielded, and measurements are taken on the probe's bottom side. Fields from other nearby structures are suppressed by shielding the smaller near-field E-probes. The spatial selectivity of these probes is typically less than a millimeter. As a result,

they can commonly be used to isolate a location on a printed circuit board to a single narrow trace.

Fig. 1.8 illustrates the EM leakage acquisition workflow. After the EM leakage acquisition, a pre-processing of raw traces is needed in common. Thereafter activities are predicted by using statistical or learning models to recover the information that can be deduced from the EM traces acquisition.

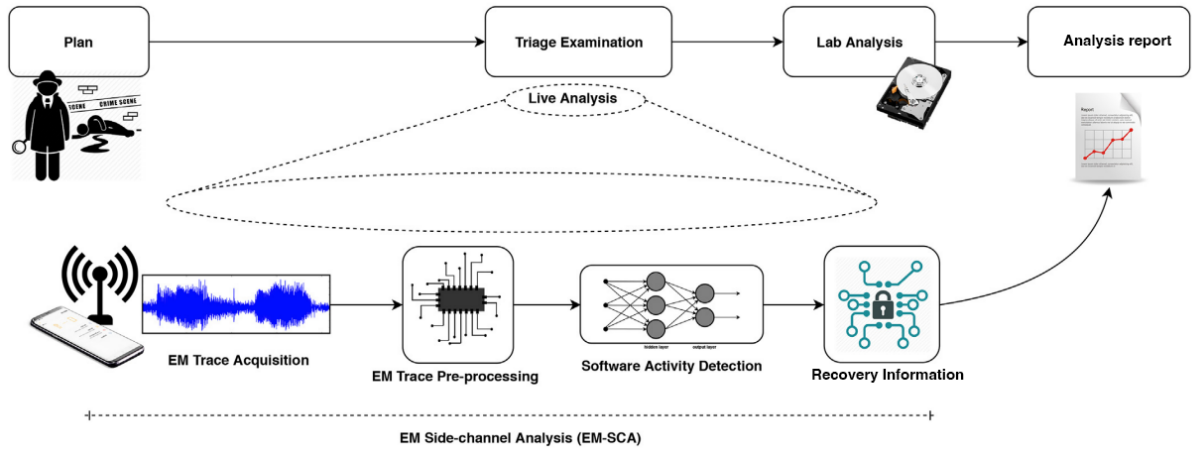


Figure 1.8 – EM-SCA techniques for activity detection acquisition workflow (adopted from [SLKS19b])

1.3.3 Software-defined radio (SDR)

The term “software radio" originated from the Division of E-Systems Inc. (now Raytheon) in 1984 to refer to a digital baseband receiver. This digital baseband receiver enabled programmable interference cancellation and demodulation for broadband signals by utilizing several array processors accessing shared memory [Joh85]. The first SDR device was produced by Ulrich L. Rohde in 1982 using the COSMAC (Complementary Symmetry Monolithic Array Computer) chip for the US Department of Defense [ARR]. Early in the 1990s, J. Mitola separately defined the design principles for “software radio" [Mit93], as well as described a proposal to develop a GSM-base station that would combine a digital receiver with digitally controlled communications jammers to create a real software-based transceiver.

From a high level, SDR is a minimal hardware component with data processing and reconfigurability performed mostly in software, controlling the center frequency, band-

width, gain, and other parameters with a fast analog-to-digital converter (ADC) or, in some cases, offloads the computation via a Field Programmable Gate Array (FPGA). A wide variety of SDR hardware is compatible with digital signal processing frameworks (*e.g.* HackRF One[GAD21], USRP with GNURadio [Val08, GNU21]). With SDRs, the software implementation makes its way to the physical layer, which is highly flexible and allows a system's behavior to be adjusted with a simple software interface rather than redesigning and rebuilding a whole new hardware. This is incredibly useful in terms of research and prototyping.

By leveraging EM signals from SDR, it allows monitoring EM measurement in a longer time window with a smaller storage requirement compared to general crypto side-channel attack or anomaly detection using an oscilloscope. Due to its capability to scan through a wide range of frequencies to locate potential EM leakages, SDR is now becoming a strong candidate for EM side-channel research due to its scanning capability. Recent research shows that SDR is an efficient solution to perform side channel attacks (on AES-128 [FI17], SHA1 [SLKS19b], ElGamal [GPPT15]). Fig. 1.9 illustrates an example of a 3072-bit ElGamal key attack using SDR presented in [GPPT15]. The prototype, named Pita "bread", is built of readily-available electronics with a low-cost setup including a SDR receiver FUNcube Dongle Pro+, small embedded device Rikomagic MK802 IV, WiFi antenna and AA batteries, which suffice for several hours of operation.

Figure 1.9 – The Pita handheld prototype measures the target computer (left) at a specific frequency band and streams the digitized signal over Wi-Fi, in real time, to the attacker's computer (right).

Recent attacks over smaller distances, such as screaming channels [CPM+18] which

succeeded in breaking AES by exploiting wireless communication such as Bluetooth and Wi-Fi. Recent work to detect malware using EM [SNA+20, KSN+19b] demonstrated that digital oscilloscopes and spectrum analyzers are feasible to monitor and capture EM traces to detect and classify malware. However, such equipment is costly, and it is impractical to exploit a more expensive device to monitor a lower-valued target. To the contrary, SDR is another EM monitor technology that provides flexibility and low cost that will be further discussed in Chapter 2.

1.3.4 Side-channel leakage: Dimensional reduction, feature extraction and transformation

Normalized Inter-Class Variance (NICV)

The output of side-channel data acquisition is typically large measurement traces, which need to be reduced for further processing. Extracting features within large measurement traces can be a challenging step. In the field of physical side-channel analysis of cryptographic algorithms, several methods have been published relying on statistical measures such as mean and variance, for example, NICV [BDGN14], SOST/SOSD [GLP06], the Pearson correlation coefficient [MOP07, HZ12], TVLA [SM16]. In our methodology, we will rely on NICV as it is straightforward to implement, time efficient, and not model-agnostic (contrary to the TVLA).

NICV is defined as:

$$\mathrm{NICV}(X, Y) = \frac{\mathrm{Var}[\mathbb{E}[X|Y]]}{\mathrm{Var}[X]} \qquad \text{(Eq. 1.3.3)}$$

with X being the recorded data, Y being the labels and Var (resp. $\mathbb{E}$) the variance (resp. the expectation).

A key advantage of NICV over state-of-the-art is that NICV does not need a clone device nor knowledge of secret features of the target system. NICV has a low computation requirement and detects leakage using public information such as input plaintexts or output ciphertexts only. It can also be used to test the efficiency of leakage models, the quality of traces, and the robustness of countermeasures.

Linear discriminant analysis (LDA)

A popular supervised feature transformation algorithm is the linear discriminant algorithm (LDA) which finds a linear combination of features separating two or more

classes [JWHT14]. LDA explicitly tries to model the difference between the classes of data, which makes it a suitable preprocessing algorithm in case of large data. It focuses on maximizing the separability among known categories, and the projection will keep classes as far apart as possible, so that LDA is a good technique to reduce dimensionality before another classification algorithm such as SVM. Note that the features are transformed into another feature space such that original dependencies (shapes, patterns) between features do cause information loss. Even though it will speed up training, it may make the system performs slightly worse. Recent work [MBBB16] shows that LDA can also be used directly as a classifier.

1.4 Detection and classification techniques

This section presents a background of detection and classification techniques in malware analysis, with a focus on IoT systems to address both the advantages and disadvantages of traditional defenses. The proposed techniques include static malware analysis, in which features are extracted statically from samples, and dynamic malware analysis techniques, in which features are extracted from behavioral monitor. Furthermore, we will discuss learning techniques such as machine learning and deep learning.

1.4.1 Malware detection: static and dynamic approaches

In common, malware defenders rely on signature-based analysis or dynamic sandboxes in order to gain information about malicious binaries. In particular, static malware analysis and signature-based malware detection are performance-wise effective, but are trivially evaded by obfuscation techniques. Besides, dynamic malware sandboxes automatically analyze new suspected binaries in special environments (e.g., virtual machines). The results of sandboxes are very likely to be incomplete due to inappropriate environments that are similar to the synthetic ones used in security labs. Dynamic malware analysis has two major problems: sandbox evasion and unstable instrumentation solutions, which heavily rely on operating systems and architectures.

Static malware analysis

Static analysis verifies the sample actions conducted in practice without actually executing them. Additionally, in static malware reverse engineering, analysts must disassemble and decompile the binary manually or automatically using reverse-engineering tools. Fortunately, in some cases, the source code has been leaked or published and is a very genuine source for analyzing malware (*e.g* theZoo [1]).

In reverse-engineering, two widely-used disassembling algorithms are: linear sweep and recursive traversal [WF12]. Linear sweep algorithms start from the first byte of the code section and consecutively analyze each successive instruction. The method is simple and fast, but it has some serious drawbacks that arise due to variable instruction size, data, or garbage embedded into the code stream. This algorithm, however, can mistakenly interpret data as code and, accordingly, propagate disassembling errors throughout all the following regular instructions. Recursive traversal, on the other hand, does not using a strictly sequential approach, but starts to disassemble at the known code entry-points and recursively following each branch instruction. By that, only valid code is observed, and therefore unaligned instructions or embedded data will not disturb the process. The main disadvantage of this method is the assumption that each jump target can be identified by static analysis, which is not always possible in the case of indirect calls. An improvement to this algorithm is speculative disassembly, which tries to also parse the left-out gaps between the reachable code regions, using linear sweep for that. Most of the popular reverse engineering tools that support ELF binaries use recursive traversal, e.g., IDA Pro [2], Hopper Disassembler [3], Binary Ninja [4], or radare2 [5] which supports both algorithms. Based on disassembly output, the binary analysis framework attempts to reconstruct high-level programming language by using decompilation techniques. In fact, it produces approximately pseudo code since the compilation procedure is a lossy process that strips symbols, optimizes the code structure, etc. In particular, the Hex-Rays [6] plug-in for IDA is a powerful decompiler which can produce C-like pseudo code as an output.

The advantages of automated static malware analysis are low resource consump-

1. https://github.com/ytisf/theZoo
2. https://www.hex-rays.com/products/ida/
3. https://www.hopperapp.com
4. https://binary.ninja
5. https://rada.re/
6. https://www.hex-rays.com/products/decompiler/

tion and analysis time for disassembly, which usually depends on the size of the binary. One outstanding feature is showing an overall view of the malware, since both linear sweep and recursive traversal techniques can cover all possible binary execution paths, while traditional dynamic malware analysis can cover only one execution path. However, an experienced analyst still has difficulties understanding packed, obfuscated, or junk code:

- Most malware nowadays is written in high level programming languages, a minor modification in source code will bring on significant changes in the disassembly.
- Malware relying on external or environment dependence values cannot be statically determined correctly (e.g., Command and Control (C&C) server configurations, current system date, indirect jump instructions, CPU cores, etc.).
- The use of obfuscation may hinder the malware static analysis. Caliskan-Islam [CIYD+15] mentioned that with the use of some particular compilers that lack decompilers and produce nonstandard disassembly. For instance, the Movfuscator [Dom21] compiles programs into only `mov` instructions without any self-modifying code or transport-triggered calculation. This may likewise hinder the disassembly approach, particularly if the compiler is not generally available and cannot be fingerprinted.

Dynamic malware analysis

Dynamic malware analysis refers to techniques that execute binary samples, functions, shellcode, etc. to verify actions the malware performs in practice by executing them. To monitor which functions are called usually means intercepting function calls, a process known as "*hooking*." Thereafter, output the results of the invocation to a log file for further analysis. Common techniques for dynamic malware analysis are described as follows:

- Analysis in user or kernel mode: A malware sample is executed under debugger, dynamic binary instrumentation or analysis modules implemented in either user or kernel space, makes it simple to invoke functions or API calls. However, because malware analysis should not be done on a live system, this approach is often performed inside virtual machines.

- Analysis in emulator: An emulator transforms the CPU instructions required for one architecture to run them on another. To examine malware sample behaviors, this technique utilizes memory or CPU emulation (libemu, QEMU, etc.) or full system emulation (Boshs, etc.).
- Analysis in virtual machine(s): Virtual machines provide the functionality necessary to run full operating systems. A hypervisor utilizes native execution to share and control hardware, allowing several environments to operate on the same physical computer while being isolated from one another. To monitor malicious behavior, samples are transferred to VMWare, VirtualBox, or other virtualized containers. Following sample execution, used computers will be restored to a clean snapshot. Used machines will be restored to a *clean* snapshot after sample execution.
- Bare environment analysis: Suspicious samples are transferred to a real hardware environment to investigate malicious behaviors that virtual machine evasion techniques would not be able to evade in this type of environment.
- Network simulation: To monitor network activities, the environment uses a simulated or filtered network instead of the Internet in order to avoid revealing the analysis environment information.

In contrast to static analysis, dynamic malware analysis can give only a subset of all possible execution paths rather than an overview of observed sample. The general work flow of malware analysis, shown in Fig. 1.10, is a hybrid solution of both static malware analysis and dynamic malware analysis.

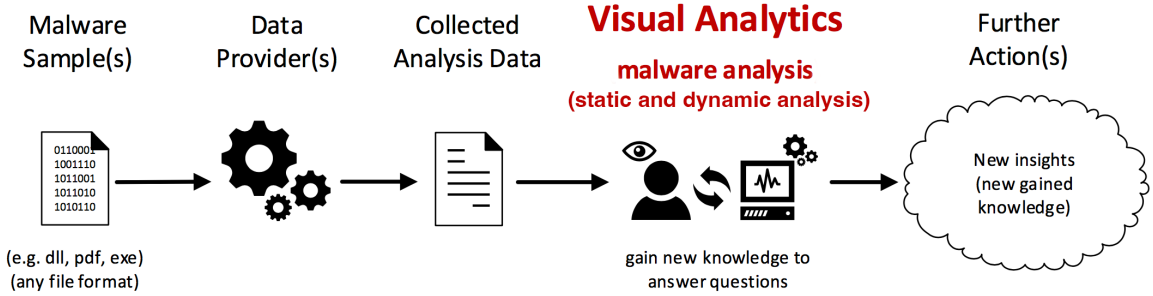


Figure 1.10 – Data collection from malware samples and interactive analysis of these data using visual analytics[WFL+15].

1.4.2 Machine learning

In a machine learning paradigm, a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E [Mit97]. Performing machine learning involves building a model that has been trained on some training data and can subsequently analyze more data to make predictions. For machine learning systems, several types of models have been utilized and investigated. Machine learning approaches are traditionally divided into 3 main categories, depending on the nature of the "signal" or "feedback" available to the learning system: Supervised learning, unsupervised learning and reinforcement learning.

Supervised learning

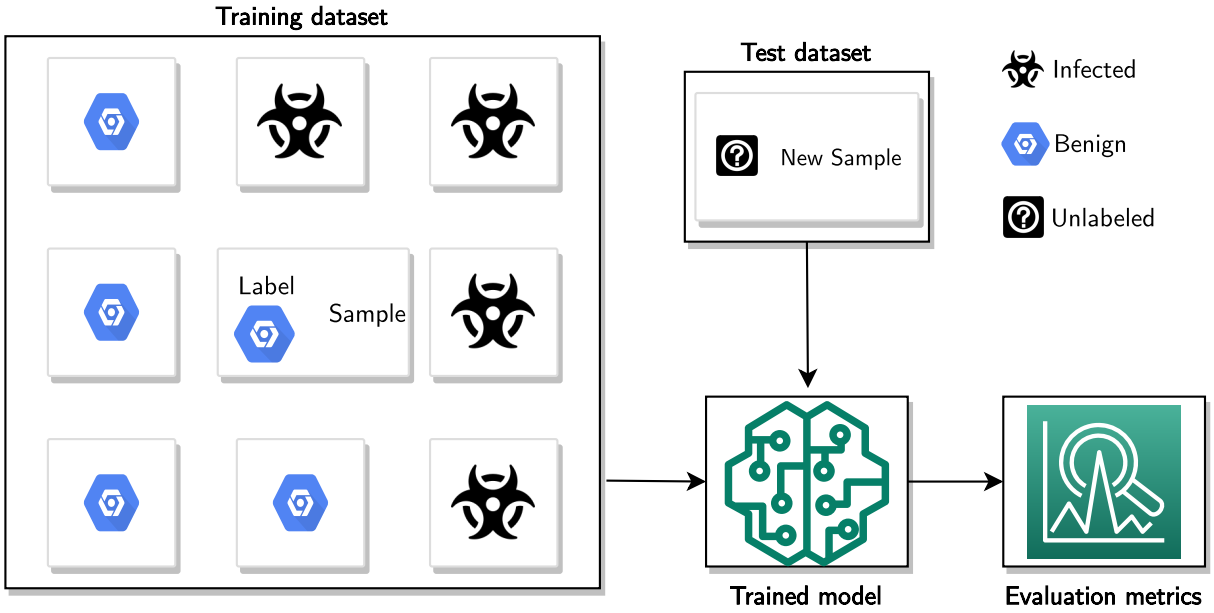


Figure 1.11 – An example of supervised learning: labeled training dataset for malware detection.

A dataset with features is presented to supervised learning algorithms, where each sample is additionally tagged with a “label". A typical supervised learning problem is malware classification. It is trained with many samples along with their labels

(*e.g.*malicious or benign), and it must learn how to classify new samples. Fig. 1.11 shows an example of supervised learning in malware detection.

Naive Bayes (NB)

In supervised learning, a labeled training set is available to build models that are used to make predictions on the testing dataset. The NB classifier is based on applying Bayes' theorem with a strong (naive) independence assumption between the features. It is further based on a Gaussian distribution assumption, which is most often not given in practice, but has still shown comparable performance in the physical side-channel domain when revealing secret keys [PHG17]. The strong benefits of NB are its low resource requirement, fast computation, and no requirement for tunable parameters.

Bayes theorem provides a way of calculating the posterior probability, $P(y|X)$, from $P(y)$, $P(X)$, and $P(X|y)$ as:

$$p(y|X) = \frac{p(y)\prod_{i=1}^{m} p(x_i|y)}{p(X)} \qquad \text{(Eq. 1.4.1)}$$

where, y is class variable and $X = (x_1, x_2, ..., x_m)$ is a dependent feature vector (of size m). $P(y|X)$ is the posterior probability of class (target) given predictor (attribute). $P(y)$ is the prior probability of class. $P(x|y)$ is the likelihood which is the probability of predictor given class. $P(X)$ is the prior probability of predictor.

Naive Bayes classifier assumes that the effect of the value of a predictor (X) on a given class (y) is independent of the values of other predictors. For building a classifier model, we find the probability of given set of inputs for all possible values of the class variable y and take the output with maximum probability. This can be expressed mathematically as:

$$\text{classify}(x_1, x_2, ..., x_m) = \text{argmax}_y P(y)\prod_{i=1}^{m} p(x_i|y) \qquad \text{(Eq. 1.4.2)}$$

Support vector machines (SVM)

Support vector machines (SVM) is a powerful supervised machine learning technique that is widely used in Machine Learning [JWHT14]. It is based on finding hyperplanes that maximize the features' separation in class labels. Using a kernel trick and transforming features into a higher-dimensional feature space (*e.g.* by using the Gaus-

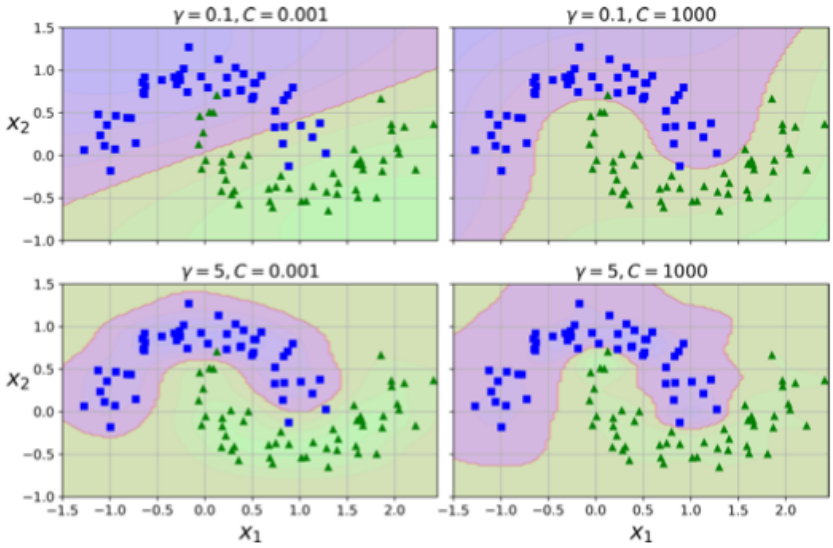


Figure 1.12 – SVM classifiers using an Radial Basis Function (RBF) kernel show models trained with different values of hyperparameters γ and C [Gér19].

sian radial basis function), SVM is able to perform linear or nonlinear classification, regression, and outlier detection.

SVM is similar to logistic regression that solves a regression problem. The goal is to build a system that can take a vector $X \in \mathbb{R}^m$ as input and predict the class identity. We define $w \in \mathbb{R}^m$ is a vector of parameters where w is a set of weights that determine how each feature affects the prediction.

$$\text{classify}(x_1, x_2, ..., x_m) = w^\intercal X + b \qquad \text{(Eq. 1.4.3)}$$

The SVM predicts that the positive class is present when $w^T X + b$ is positive. Likewise, it predicts that the negative class is present when $w^\intercal X + b$ is negative. The kernel technique is a significant invention connected with SVM [GBC16]. It can be demonstrated, for example, that the SVM's linear function can be rewritten as:

$$w^\intercal X + b = b + \sum_{i=1} \alpha_i \phi(X, x_i) \qquad \text{(Eq. 1.4.4)}$$

where x_i is a training example, α is a vector of coefficients and kernel function ϕ. Technique to tackle non-linear problems is to add features using *similarity function*. One

common technique of similarity function is a Gaussian RBF that is defined as:

$$\phi_\gamma(x, x') = exp(-\gamma||x - x'||^2) \quad \text{(Eq. 1.4.5)}$$

where γ is a parameter that sets the "spread" of the kernel ϕ. Either x or x' will be the centre of the radial basis function and γ will determine the area of influence over the data space. In practice, additional regularization hyperparameter C is used to control error and trades off correct classification of training examples against maximization of the margin of decision function. The larger values of C, the smaller margin will be accepted if the decision function is better at classifying all training points correctly. A lower C will encourage a larger margin, so that simpler decision function for the cost of training accuracy (see Fig.1.12).

1.4.3 Deep learning

Artificial neural network was motivated by studying how the human brain functions. Millions of neurons in the human brain use electrical and chemical signals to communicate with one another. The inputs to a neuron are combined in some way, and if they are above a threshold, the neuron fires and an output is sent out to other neurons through the axon. This principle is also used in artificial neural networks [MD11].

Deep learning techniques such as neural networks have been recently applied in many fields. A generic neural network architecture is a Multi-layer Perceptron (MLP) which consists only of dense layers, i.e. layers where each neuron has a weighted connection to all neurons of the next layer. Another popular type of neural networks, especially in image classification, is Convolution Neural Networks (CNN). It initially has been developed specifically for image processing. The main components are convolution layers, which define a set of filters that are convolved to learn the spacial relationship between the input features. The benefit of using CNN is their ability to develop an internal representation of a two-dimensional image. This allows the model to learn position and scale in variant structures of the data, which is significant when working with images. Because of the recursive process of learning that happens when a neural network algorithm detects malware or generates a probability to render software as "infected", an author may not be able to understand and explain how that outcome is achieved[Dwy19]. The weights and dependencies between a neural network's layers, while adjustable by its author, do not always result in a linear output

change. In this thesis, we will concentrate on these two architectures.

Multi-layer Perceptron (MLP)

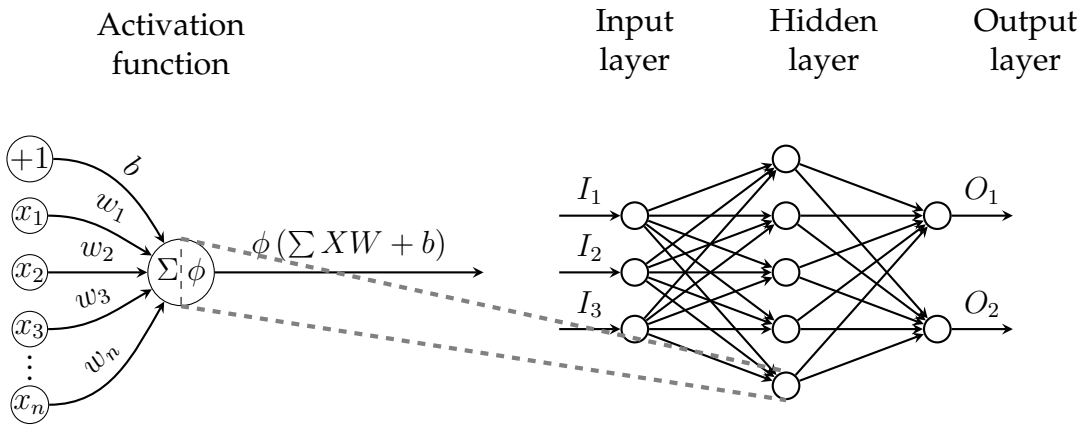


Figure 1.13 – Illustration of a MLP architecture.

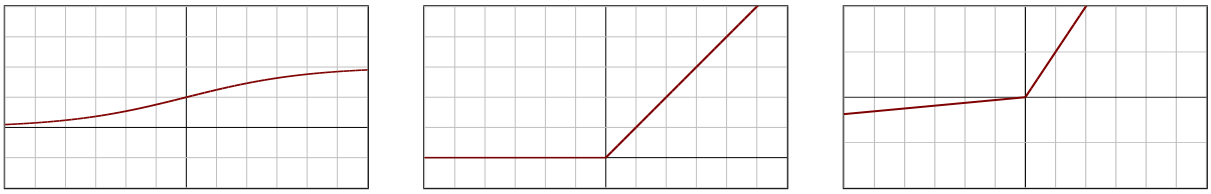
A MLP is composed of several layers (1.13). Each neuron that composes a layer has a weighted connection to all the neurons of the previous layer. As defined in [Gér19], considering the matrix W of these weights, X the matrix input feature, b the bias vector, and ϕ the activation function, the output of a layer is:

$$h_{W,b}(X) = \phi(XW + b) \qquad \text{(Eq. 1.4.6)}$$

During the training phase, the MLP is fed with a set of data divided into mini-batches, and goes through the entire set multiple times (epochs). The weights matrices of each layer are updated every time a forward pass is over for one batch, in order to minimize a loss function that measures how far the output (prediction) is from the desired one (actual value). This is done using the back-propagation algorithm [RHW85]. Description of activation functions used in this thesis will be described next.

Activation functions

The activation function of a node defines the output of that node given an input or set of inputs. Numerous activation functions have been studied in the literature. We only describe some of them that are widely used in modern neural network architectures (Fig. 1.14) as follows for any input $x \in \mathbb{R}$:



(a) sigmoid (b) Rectified Linear Unit (ReLU) (c) Leaky ReLU

Figure 1.14 – Plots of widely used activation functions

- **Sigmoid**

$$f(x) = \frac{1}{1 + e^{-x}} \tag{Eq. 1.4.7}$$

This function bounded in range $(0, 1)$, has non-negative derivate and differentiable. It is used in the output layer and mostly for binary classification.

- **RELU**

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{Eq. 1.4.8}$$

This function continuously unbounded in range $[0, \inf)$, keep positive values untouched and disregard negative ones. Recently this activation function has become one of the most widely used in machine learning community because it well performs and being fast to compute.

- **Leaky RELU**

$$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{Eq. 1.4.9}$$

Leaky ReLU is a variant of ReLU. It allows for a small, non-zero gradient when the unit is saturated and not active [MHN+13].

- **Softmax**

This activation functions is not of a single fold x from the previous layer or layers, but on a vector of values. So that for any input $x = (x_1, \ldots, x_J) \in \mathbb{R}^J$:

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \text{ for } i = 1, \ldots, J \tag{Eq. 1.4.10}$$

The softmax function is often used in various multiclass classification methods for the output layer, where the function classify input into one of more than 2 classes.

Neural network layers

Typically a neural network model consists of 3 successive layers: input layer, hidden layer(s) and output layer. An input layer made of the network's input values as well as output layer composed of the network's output values. Hidden layer(s) are the internals between the input and output layers, hence they are not directly accessible from outsiders. The following provides a description of common types of layers:

- **Fully connected layer** in this kind of layer, all the inputs from one layer are connected to every activation neuron in the other layer(s).

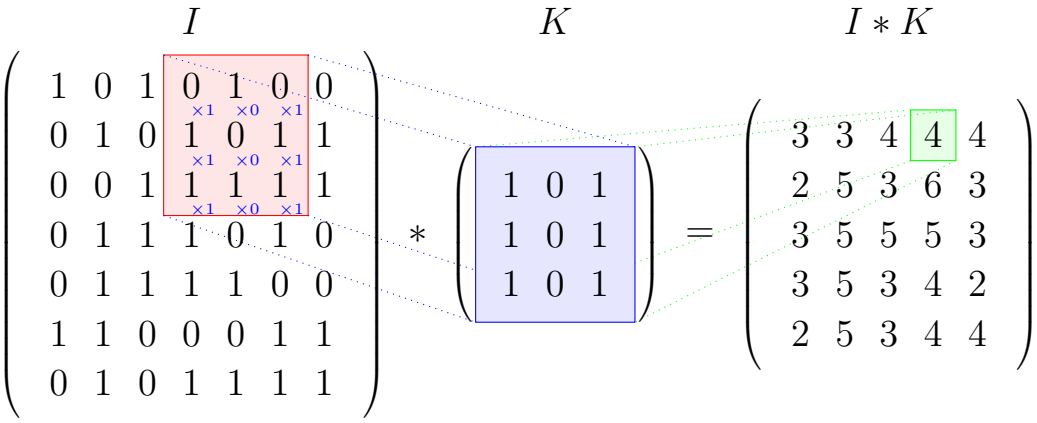


Figure 1.15 – Convolution operation example between input I and kernel K with no padding and stride value of 1.

- **Convolutional layers**: in this kind of layer, not all elements of the inputs are connected to every neuron of the convolutional layer. It contains a set of convolutional kernels (or filters), which get convolved with the inputs (N-dimensional metrics) to generate an output feature map. A kernel is nothing more than a small matrix of values, where each value is known as the weight. By sliding this kernel along the input, we can measure the convolution operation product to compute the output of the layer. A convolution operation is a mathematical calculation on two functions (f and g) that produces a third function by measuring the integral of their point-wise multiplication $f * g$ (see Fig. 1.15). The convolution operation

has deep relationships with the Fourier transform and is heavily used in signal processing.

In practice, we use the convolution operation with additional parameters such as *padding* to the input and with *stride* to the kernel. By increasing stride, it will result in a lower-dimensional feature map. The padding handles the border size information of the input, otherwise without using padding the border side features are disregarded.

- **Pooling layer**: this layer is similar to convolutional layers where not all elements of the inputs are connected to every neuron of the layer. The goal of the pooling layer is to shrink the input to a smaller size for summary statistics of the nearby outputs. Pooling layer parameters have their own size, stride, and padding type but no weight. All it does is to use an aggregation function to combine the data. The pooling units can perform various tasks, such as max pooling, average (or mean) pooling, and L2-norm pooling. Mean pooling was formerly popular, but lately the max pooling method has been demonstrated to perform better in practice. Figure 1.16 shows a max pooling layer, which is the most common type of pooling layer at the time of writing. In this example, we use a max pooling layer which down-samples the volume spatially, independently in each depth slice of the input volume with filter size 2 and stride 2 with no padding.

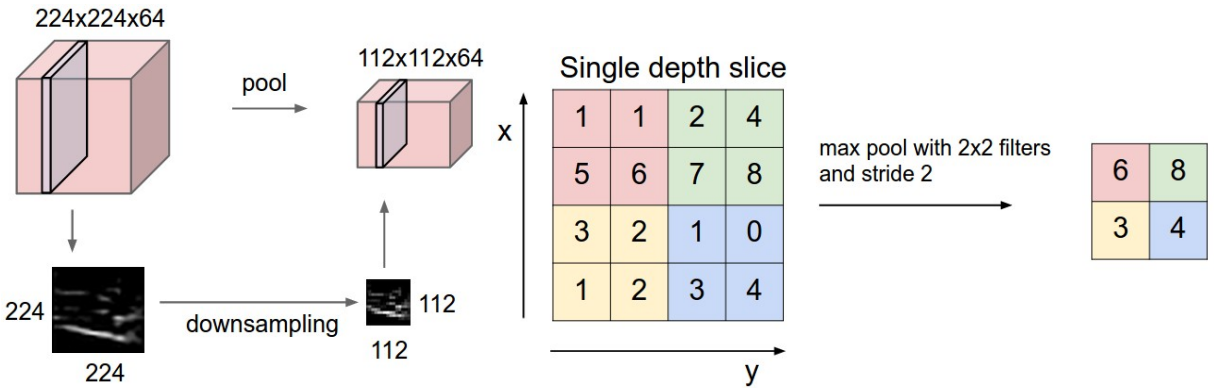


Figure 1.16 – Visual example of max pooling. Left: Pooling the input volume of size (224,224,64) into output volume of size (112,112,64). Right: Max pooling with a stride of 2, each max is taken over 4 numbers (little 2x2 square) [CS221].

Convolution Neural Networks (CNN)

CNNs are based on an architecture inspired by the visual cortex, and thus specialized in image processing. They follow the same principles as MLPs for training, but the neurons in CNN are more sparsely connected between the layers. They take advantage of the spacial information, and each neuron is connected only to those located in a small rectangle (the filters) of the previous layer. This allows us to detect more and more complex patterns the deeper we go through the network. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [GBC16].

CNNs are constituted of different kinds of layers. The main components are the convolution layers themselves, which define a set of filters that are convolved during the forward pass, and produce feature maps that highlight spacial relationship between the input pixels. The weights of the filter are learned and updated at each back-propagation pass. They are generally followed by a pooling layer, that uses an aggregation function (the most widely used nowadays is the max function) and apply it by sliding a rectangle through the input image. Using pooling layers tends to reduce overfitting, and allows increasing the computation speed by shrinking the feature map. A typical CNN architecture is a stack of an input layer, hidden layers, and an output layer where each hidden layer consists of several convolution and pooling layers, followed by a regular MLP (few fully connected layers), and normalization layers to perform the final classification.

1.4.4 Classifiers and evaluation metrics

Learning classifiers

Machine learning and deep learning models are very powerful for classification tasks. [Gér19] divided learning classifiers into three main categories: binary classification, multi-label classification, and multi-class classification.

For a binary classification problem, one just needs a single output neuron using the *sigmoid* activation function so that the output will be a number between 0 and 1 (*i.e.* "infected" or "clean"), which can be interpreted as the estimated probability of the positive class p. The probability of the negative class is $n = 1 - p$.

A multilabel classification system is a classification system that outputs multiple binary tags. There are a variety of metrics to use when evaluating a multilabel classi-

fier, and they depend on the goal of the project. One method is to compute the average score after measuring the F1-score for each individual label or any other binary classifier metrics that will be discussed at the end of this subsection. Multi-label binary classification tasks are likewise solved easily by learning classifiers. We only need two output neurons in this case, both utilizing the *sigmoid* activation function. In general, one output neuron would be assigned to each positive class. It's worth noting that the output probabilities might not always add up to 1 so that this allows the model to output any combination of labels.

Multioutput or multiclass classification is the last form of classification that is basically a generalization of multilabel classification, in which each label can be classified into many classes (i.e., it can have more than two possible values rather than a simple binary tag). In this case, each instance can only belong to one of three or more classes (e.g., more than 2 possible malware variants classification, or classes 0 through 9 for digit image classification), then one output neuron per class is required, and the *softmax* activation function should be used for the entire output layer. The *softmax* function ensures that all estimated probabilities are in the range of 0 to 1 and that they add up to 1 which is required so the classes are exclusive.

Evaluation metrics

The goal of classification is to predict class labels based on input data, and commonly, there are two potential output classes in binary classification. In multi-class case scenarios, there are more than two possible classes Malware detection problem is an example of binary classification, where the input data may include binary samples or behaviors (*e.g.* API calls, file accesses, network logs, etc.), and the output label is either "infected" or "benign." (see Figure 1.11) The two classes are sometimes referred to as "positive" and "negative." There are several methods for assessing categorization performance. Some of the most prominent measures are accuracy, confusion matrix, log-loss, and AUC.

For binary classification problems during this thesis, the following evaluation metrics notations will be proposed:

Condition positive (*p*): The number of real positive cases in the data

Condition negative (*n*): The number of real negative cases in the data

True Positive (TP): A test result that correctly indicates the presence of a condition or

characteristic

True Negative (TN): A test result that correctly indicates the absence of a condition or characteristic

Palse Positive (FP): A test result which wrongly indicates that a particular condition or attribute is present

False Negative (FN): A test result which wrongly indicates that a particular condition or attribute is absent

Accuracy

The accuracy of a classifier is simply the frequency of classifier which produces the correct prediction. The number of right predictions divided by the total number of predictions (the number of data points in the test dataset) is the Accuracy ratio:

$$\textbf{Accuracy} = \frac{\text{\# correct predictions}}{\text{\# total data points}} = \frac{TP + TN}{TP + TN + FP + FN} \qquad \text{(Eq. 1.4.11)}$$

Confusion matrix

Accuracy does not differentiate between classes, where the results for classes infected and benign are regarded equally. Because the cost of wrong classification may differ for the two classes, for example one may have a lot more number of samples in dataset of one class than the other, so that one would want to look at how many cases failed for class infected vs class benign. A confusion matrix (Table 1.1) shows a more detailed analysis of correct and incorrect classifications for each class.

Table 1.1 – Confusion matrix for two classes.

	Predicted class		
True Class	Positive	Negative	Total
Positive	TP	FN	p
Negative	FP	TN	n
Total	p'	n'	N

We distinguish between the two classes in some two-class scenarios, and hence the two types of errors: FP and FN. Consider a false negative scenario of malware detection

where a malicious sample is incorrectly whitelisted to execute, while a false positive occurs when benign software is refused. Obviously, the two types of errors are not equivalent, with the false negative case being far more serious. The true positive rate, also known as the hit rate, is the percentage of real malware we detect, whereas the false positive rate, also known as the false alert rate, is the percentage of samples we denied incorrectly.

Precision-Recall

The confusion matrix provides a lot of information, but one might prefer a more simple metric. The classifier precision, which is the accuracy of positive predictions, is a useful tool to use in classification problems. The precision (Eq. 1.4.12) for a class is the number of true positives divided by the total number of elements labeled as belonging to the positive class (*i.e.* the sum of true positives and false positives).

$$\textbf{Precision} = \frac{TP}{TP + FP} \tag{Eq. 1.4.12}$$

Making only one positive prediction and ensuring it is right (precision = 1/1 = 100%) is a simple way to achieve perfect precision. And this would be insignificant since the classifier would discard all except one good example. As a result, accuracy is sometimes combined with another statistic known as recall, also known as sensitivity or the true positive rate (TPR). Recall (Eq. 1.4.13), in a classification task, is defined as the number of true positives divided by the total number of elements that actually belong to the positive class (*i.e.* the sum of true positives and false negatives).

$$\textbf{Recall}\ (= TPR) = \frac{TP}{TP + FN} \tag{Eq. 1.4.13}$$

In binary classification, precision is also known as positive predictive value, while recall is also known as sensitivity. When one needs a straightaway method to compare these two metrics, it's typically easier to combine precision and recall into a single statistic called the F1 score. F1 score (Eq. 1.4.14) is calculated using the harmonic mean of precision and recall . As a result, the classifier will only earn a high F1 score if it has high recall and accuracy.

$$\textbf{F1} = \frac{2 * \textbf{Precision} * \textbf{Recall}}{\textbf{Precision} + \textbf{Recall}} = \frac{2 * TP}{2 * TP + FP + FN} \tag{Eq. 1.4.14}$$

Other related measures used in classification include true negative rate and balanced accuracy. True Negative Rate (TNR) is also called specificity.

$$\textbf{TNR} = \frac{TN}{TN + FP} \qquad \text{(Eq. 1.4.15)}$$

For imbalanced data sets, accuracy might be a misleading metric. Consider a dataset that contains 99 negative samples and 1 positive sample. In this situation, classifying all values as negative yields a 0.99 accuracy score. Balanced Accuracy (BA) is not affected by this issue. It normalizes true positive and true negative predictions by the number of positive and negative samples, respectively, and divides their sum by two:

$$\textbf{BA} = \frac{TPR + TNR}{2} \qquad \text{(Eq. 1.4.16)}$$

2 State of the Art

This chapter presents a review of the related studies on malware evasion techniques to address both advantages and disadvantages of traditional malware defenses. Thereafter, we focus on the existing work of malware detection and rootkit detection techniques, specifically on the side-channel perspective.

Contents

2.1 Malware evasion techniques

2.1.1 Evasion of static code analysis

Malware developers often attempt to pack or compress their samples for a smaller size malware, to hinder the real malicious code section, or to obfuscate malware to make static analysis process more time-consuming using a variety of methods. A typical example is static code obfuscation using packers to generate numerous mal-

ware variants automatically, whilst using the same malware source code. One effective obfuscation technique against static analysis has been studied by Moser et al. [MKK07b] that presented an obfuscation scheme based on opaque constants which are *hidden* in processor registers, and manipulate the control flow of a program. Hence, it is becoming non-trivial to investigate malware behaviors by static analysis and reverse engineering. Many IoT malware is packed by popular packer `UPX` or its variant [Lab21]. Static analysis was the preferred technique for malware analysis until researchers demonstrated that the widespread use of packing and obfuscation made it inadequate in the malware domain.

2.1.2 Evasion of dynamic analysis

Dynamic malware analysis sandboxes generally operate samples within a *jail* environment such as debugger, virtualized or emulated environment and therefore exposed some information which is different from normal user machine. The interest in dynamic malware analysis is not only limited to malware researchers, but also spreads to malware developers to find ways to mitigate malware analysis. Chen *et al.* [CAM+08] grouped evasion techniques by the system abstractions of how they operate: Hardware, Environment, Application and Behavior. Table 2.1 derives a survey of common sandbox fingerprint techniques from previous research [CAM+08, LKMC11, BCK+10, Fer07, YIT+16, Lin11] that compares multiple evasion techniques, their accuracy in detecting dynamic malware analysis environments, and the level of anti-evasion difficulties.

Several research studies [CAM+08, LKMC11, BCK+10] have been carried out to detect whether a malware program behaves differently between an analysis environment and a normal user environment. This involves comparing the runtime behaviors of each analysis system. Chen et al. [CAM+08] analyzed malware samples on native machines, virtual machines, and debuggers separately. They detect any difference in persistent behavior to indicate malware evasion. In their experiments amongst 6222 Windows samples, 40% showed less malicious behavior with a debugger and 4% demonstrated less malicious behavior in a virtual machine. However, Lindorfer *et al.* [LKMC11, Lin11] argue that this approach might lead to a high number of samples being incorrectly classified as evasive. Their *Disarm* executes samples 3 times in each of the 4 sandboxes to establish a baseline for the sample's variation in behavior. Their

Table 2.1 – Common sandbox fingerprinting features (adopted from [PVM19]) consist of multiple evasion techniques, their accuracy to detect dynamic malware analysis environment, and the level of anti-evasion difficulties

Abstraction	Artifacts	Accuracy	Counter-measure
Hardware	*Device information* Includes device ID and manufacturer of Ethernet, VGA adapter, mouse etc. Device specifications which often have distinguished capability such as low disk space, display resolution, processor cores, RAM, etc.	high	easy
	Particular Drivers Drivers initially installed with virtualized guest such as `QEMU`, `VMWare Tools`, `VirtualBox Guest Additions`, `kvmnet`, debugger kernel driver, etc.	high	medium
Environment	*Network configuration* Default gateway, external IP address, ARP list, MAC address, DNS servers	high	easy
	System Artifacts located in system memory such as hostname, OS information, owner name,	medium	easy
	CPU instructions Leak information by using `CPUID`, Interrupt Descriptor Table –`IDT`.	high	hard
Application	*Storage* Look for special files only belongs to virtualized machine, debugger, analysis tools in drive storage or registry keys.	high	easy
	Process name Look for debuggers, analysis running process	high	hard
Side-channel	*Timing attacks* Check for local time source, such as Time Stamp Counter– `TSC`; Time from boot to start	low	medium
	Side-channel information Sensor's information *e.g.* temperature, fan speed, etc.	high	hard
Behavioral	*User interaction* Mouse clicks, keyboard capture.	medium	easy

experiments show that 25% of malware samples produced different persistent changes between multiple executions in the *same* sandbox.

Moser *et al.* [MKK07a] proposed a multiple path exploration system that allows exploring multiple execution paths and identifying malicious actions that are executed only when certain conditions are met. This tool recognizes a branching point whenever a control flow decision is based on data outside the monitored process. Then it is detected if a control flow decision is based on the return value of a system call (*e.g.*, the current system time). Multiple path exploration is a countermeasure to logic bombs and can obtain a more complete picture of malware actions.

A. Klein *et al.* [KK17] have demonstrated that Anti-virus (AV) sandbox execution can be exploited to exfiltrate data from endpoint machines by "burning" the data into the binary to be scanned in the cloud. As one part of this research, the authors made observations on the sandboxes and discovered several fingerprints of sandboxes, such as computer name, performance counter frequency, CPU, MAC address, etc., amongst 3 cloud-based analysis services and 4 cloud AV sandboxes. Obviously, these artifacts are useful indicators for malware evasion.

2.2 Malware detection

We will first describe the traditional methods of performing automatic malware classification by means of software. We will then focus on methods closer to the hardware perspective, to present the relatively new area of analyzing side channels to observe malicious and abnormal activity.

2.2.1 Malware detection from software analysis

Using the power of machine learning to identify and classify the behavior of malevolent software is a common approach in the literature. Schultz *et al.* [SEZS01] were the first to introduce big data analysis to detect Windows PE malware, using various classifiers such as Naive Bayes and Ripper. It paved the way for later work evaluating the efficiency of different learning algorithms, supervised or not, for the same purpose.

The majority of the works discussed here use well-known machine learning algorithms, such as k-means, random forests, and support vector machines. The first attempts at machine learning malware detection and classification were mainly based on

analyzing software information. Some techniques dissect the monitored binary itself For example, in [SEZS01, KM06, RZC+18], the authors use directly the byte code using n-grams as features for the classification. Another class of works [NKJM11, HLI13, LWYZ17] that revealed itself very efficient is based on visualization of the binary as a gray-scale image, before applying classification algorithms. Structural representations of the binary have also been successfully implemented. In [KY13], the authors present a framework for automated malware classification based on function call graph, while in [EH11], they extract control flow graphs and use them as a feature vector. Both of them compared the efficiency of this approach by testing different classifiers.

Instead of analyzing the code of the binary directly, it is also possible to dynamically observe how its execution interacts with the system to try to catch suspicious activity. This is the path followed, for example, by [TIBV10] and [CTY13], where the authors examine the traces of the API calls to automatically classify the monitored executables. Following a similar idea, [AQN+11] presented a classification method using Markov chains constructed from dynamically collected instruction traces.

In [BSRB15], they showed that it is also possible to exploit the network traffic to detect malicious behaviors. If we look at the instruction traces or network activity of the executables that are being monitored, like in [TIBV10], [CTY13], [AQN+11] and [BSRB15], we can automatically classify them. More recent works are using similar techniques, but focus on taking advantage of the developments made in the deep learning field, that can surpass limitations encountered by other machine learning algorithms. For example, [KZWE16] outperforms previously used methods using convolution and recurrent networks to analyze system call sequences. [LBMNS18] and [KRM+18] also uses convolution networks, but apply it to one or two-dimensional representation of the binary, which is similar to the grayscale visualization.

2.2.2 Malware detection from hardware analysis

Since the aforementioned works perform at the software level, they are vulnerable to advanced malware evasion techniques, as discussed in section 2.1. This following section reviews a summary of techniques used in related works on malware analysis using side-channel (Table 2.2) over the years. This new area of research is investigating approaches closer to the hardware perspective. We focus on methods that leverage side-channel analysis and present the relatively new area of analyzing side channels

Table 2.2 – Highlights of techniques used in related works on malware analysis leveraging side-channel analysis.

Articles	Year	Target dataset and detection techniques
WattsUpDoc [CRR+13]	2013	Detection of 12 malware variants. Power consumption, MLP, NN, Random Forest.
Crypto-ransomware in IoT [ADCC18]	2017	Ransomware detection. Power consumption, k-Nearest Neighbor.
EDDIE [NSA+17]	2017	Code injection detection. EM, STFT, Kolmogorov-Smirnov test.
DL and anomaly detection [WZH+18]	2018	Anomaly detection of botnet. Power consumption, MLP, LSTM.
HLMD [BAT19]	2019	Malware classification of 14 variants. HPC, singular values, signature-based.
EM and Neural networks [KSN+19b]	2019	Detection of DDoS, ransomware, control flow hijack. EM, MLP.

to observe malicious or abnormal activity. Various works, such as [DMS+13, SPP+18, BAT19, ODG+15], showed that combining the observation of micro-architectural events collected by Hardware performance counter (HPC) registers with machine learning techniques can allow the detection of malware. Some studies [SEE+17, DA18] were even able to detect malware threats that was previously impossible or difficult to catch with software methods such as kernel-level rootkits and Spectre attack. But although the last discussed methods are working on hardware level, they still require access to the system, and they will inevitably induce an overhead and require isolation constraints from the malware, which can be problematic, particularly, on constrained systems.

Those are the reasons why researchers have recently become more and more interested in physical side channel information as a novel technique to detect malware One of the first works on malware detection [CRR+13], even though limited due to its constrained scenario, showed that the collection of power consumption on medical embedded devices is suitable to detect malware. [KSN+19b] presents a malware detection solution by exploiting EM side-channel signals from embedded devices through

Multi-Layer Perceptron (MLP) to detect handcrafted implementations mimicking malware, malicious parts of DDoS, ransomware and control flow hijack. A common idea to take advantage of side channel information to detect anomalies is to observe how the system behaves in its normal state, and to raise an alert when a new behavior is recorded. In [SNA+20, KSN+19a], the authors propose to detect malware by observing EM signals. During the monitoring, if the observed EM emanations deviate from the previously observed patterns, this is reported as an anomalous or malicious activity. [NSA+17] uses Short Time Fourier Transform (STFT) and Kolmogorov–Smirnov test to detect anomalies inside and between the loops through peaks in the EM spectrum. In [RGF+19], the authors put a wide-bandwidth radio frequency probe over the processor of the device and used a support vector machine to infer the values of the registers. They monitor if the hamming distance of the registers deviates from the known signature, and use this information to detect cyberattacks.

In [WZH+18], the authors use Autoencoders, Long Short-Term Memory (LSTM) units, and MLP on power consumption data to detect anomalies such as Distributed Denial-of-Service (DDoS) attack on 2 target devices: Arduino and Raspberry Pi. [DLL+20] shows an approach to detecting malicious activities on IoT devices via analyzing power side-channel signals using Convolution Neural Networks (CNN). In particular, they conducted a study of 2 architectures, ARM and MIPS, on multiple target devices, including webcams and network routers. The authors experimented with five malicious real-world families by analyzing fine-grained malware activities using their power consumption traces. The experimental results demonstrate that DeepPower is able to detect infection activities of different IoT malware with a high accuracy, however the results get worse for long trace and activities such as *wget*, *grep*, etc. The authors of [ADCC18] are successfully detecting ransomware using machine learning on time series of the power consumption of the device. [CKM21] considered CPU benchmark applications for Android benign dataset to detect malware using EM, however it is very specific stress processes that are easy to detect and classify rather than a wide-range dataset of cleanware, long-running programs and device background services. The use of physical hardware information, and particularly side-channel information, represents a great advance for malware detection. A detailed comparison of the work in different features using side-channel to analyze malicious activities is provided in Table 2.3.

Table 2.3 – Comparison with related works on side-channel malware (SCM) analysis using EM or power consumption. (*): Chapter 3 aims at SCM classification, however it also achieved good results in SCM detection scenario.

Article	SCM detection	Anomaly detection	SCM classification	Real-world SCM	Real-world analysis environment	Samples size	Variations	Benign dataset	Window size	Open data, source code	Device under test
WattsUpDoc [CRR+13]	✓	-	-	✓	-	15	-	-	5s	-	Windows XP Embedded 664 MHz
IDEA [KSN+19a]	-	✓	-	-	-	3	-	-	<40μs	-	AT328p 16MHz, Cortex A8
REMOTE [SNA+20]	-	✓	-	✓	-	3	-	-	<10ms	-	Single-core ARM 1Ghz
Wang *et al.* [WZH+18]	-	✓	-	-	-	1	-	-	10s	-	Raspberry Pi, Arduino, Siemens PLC
Khan *et al.* [KSN+19b]	✓	-	-	-	-	3	-	-	<150μs	-	Cyclone II FPGA & NIOS II soft-processor
DeepPower [DLL+20]	✓	-	✓	✓	-	5	-	-	1s	-	MIPS/ARM OpenWRT
Chawla *et al.* [CKM21]	✓	-	✓	✓	-	137	-	✓	10s	-	Android Intrinsyc Open-Q 820
Chapter 3	(✓)*	-	✓	✓	✓	35	✓	✓	2.5s	✓	Multi-core, 900 Mhz ARM

2.3 Rootkit detection through side-channel

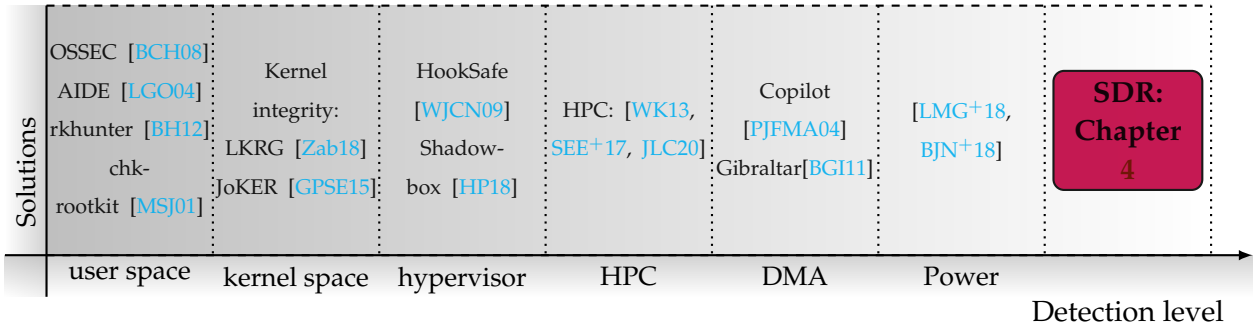


Figure 2.1 – Taxonomy of rootkit detection approaches and positioning our approach in the state of the art and open source tools.

On the effectiveness of Linux rootkit detection tools, Junnila J. [Jun20] carried out an empirical evaluation of 5 prominent anti-rootkit tools: OSSEC [BCH08], AIDE [LGO04], rkhunter [BH12], chkrootkit [MSJ01] and Linux Kernel Runtime Guard (LKRG) [Zab18] against 15 rootkits. Surprisingly, the results showed that only 37.3% of the detection tests provided any indication of infected systems, in particular detection rate was 46.7% for user mode rootkits and 31.1% for kernel mode rootkits. Traditional rootkit detection approaches such as OSSEC, AIDE, rkhunter and chkrootkit generally use signature or rule-based mechanisms to detect rootkits by looking for threat-specific information: either known rootkit binaries or known modifications of system binaries, configuration files, or system states [BLRS10]. Obviously, they cannot detect new rootkits or modified variants of existing rootkits as they are similar to signature-based virus scanning.

On rootkit detection from the kernel-level space, JoKER [GPSE15] utilizes the Joint Test Action Group (JTAG) hardware interface for trusted memory to detect rootkits. LKRG is intended to safeguard OS kernel-level integrity against kernel-level rootkits and exploits. It performs post-infection detection and responds to unauthorized changes of process credentials in OS kernel memory regions. However, such an approach requires compilation of kernel objects with additional kernel flags that must be activated during kernel compilation, thus necessitating kernel recompilation and posing a challenge for divergent, constantly evolving embedded systems.

Another study approach is to detect rootkits by putting the kernel and user space under the monitor of a virtual machine (VM). Wang Z. *et al.* [WJCN09] have pre-

sented a hypervisor-based system called HookSafe that monitors kernel hooks and prevents them from being hijacked by kernel rootkits. It is a lightweight hypervisor that enumerates a collection of vulnerable areas in kernel space such as regions that kernel-mode rootkits would attempt to modify, and relocates them to a more tightly controlled place. Their experimental results show their effectiveness against 9 rootkits with about 6% performance overhead. However, what if the rootkit is a hypervisor where typically their protection level must descend one level further, rather than a standard kernel-mode rootkit? Yuriy Bulygin [BS08] stated that you may rely on embedded microcontrollers included in the motherboard's north-bridge to monitor for virtual machine-based rootkit. The authors demonstrated a proof of concept named DeepWatch. It was further extended into HyperGuard, which uses a custom System Management Mode handler (SMI) in conjunction with onboard chipsets to protect against malicious hypervisors. Nevertheless, it raised a concern of any technology that is entirely reliant on software solutions only to ensure system integrity.

Shadow-box v2 [HP18] proposed a monitoring framework for x86 and ARM processors, which utilizes Open Platform Trusted Execution Environment (OP-TEE) to verify signatures and remote attestation from kernel executables. Rootkits living at the same protection level (hypervisor) and lower (*e.g.* [BVN16, HR15, ESZ13]) have the opportunity to evade this approach. Additionally, a VM-based solution could circumvent only the known tactics and is vulnerable to novel evasion techniques of VM fingerprinting. Furthermore, Bratus *et al.* [BLRS10] suggested that modern applications could not integrate VM techniques as a detection mechanism, and managing the VM becomes a major challenge due to its complexity and overhead.

Several studies propose combining the values of micro-architecture Hardware Performance Counters (HPC) with learning models to identify malware. Numchecker [WK13], Singh *et. al.* [SEE+17], and LKRDet [JLC20] detect Linux rootkits by looking for HPC deviations during the execution of the kernel through virtualization to determine the presence of a rootkit In particular, Singh *et. al.* [SEE+17] designed 5 tailor-made rootkits, each providing a single piece of rootkit functionality, and execute each sample while collecting HPC traces of its impact on specific benchmark application. They apply machine learning feature selection techniques in order to determine the most relevant HPCs for the rootkit detection. They identified 16 HPCs that are useful for the detection, and also find that direct kernel object manipulation (DKOM) rootkits do not significantly impact HPCs. However, [BVN16] demonstrates that ARM would

allow exception hypervisor level 2 to trap all micro-architecture instructions, including performance counters, allowing the victim OS to continue to use the performance monitor infrastructure while the presence of the rootkit remained hidden. Furthermore, recent studies [ZGJ+18, DWA+19] claim and experimentally support that using the micro-architecture information from HPCs cannot distinguish between benign and malware.

WattsUpDoc [CRR+13] was one of the earliest efforts in malware detection through hardware side-channel that demonstrated the measurement of power usage on medical embedded devices. On rootkit detection through side-channel, [KZLR12] described a network time analysis approach for monitoring performance changes caused by hardware virtualization, with the goal of detecting the hardware virtualization rootkit. It leverages the benchmark software to run on both systems with and without virtualization and if it can be shown that the computer system time is different from the virtualized instance and the non-virtualized instance, then it can be said there is potential for detecting hardware virtualized malware. [LMG+18, BJN+18, MCHR22] identify rootkits by using power-based malware detection on general-purpose computers and [LMG+18, DLL+20, WZH+18] examine machine learning to perform a behavioral detection method based on CPU power consumption. In particular, [LMG+18] investigates a number of machine learning techniques such as Nearest Neighbor, Decision Trees, Neural Networks, and Support Vector Machines, as well as presents a behavioral detection approach based on CPU power consumption that requires access to the physical computer and power clamp. The approach was tested on Windows 7, Windows 10, Ubuntu Desktop, and Ubuntu Server, for 4 distinct rootkits. Relevant data features are extracted to find the overall top performing algorithms.

Gibraltar[BGI11] and Copilot [PJFMA04] leverage direct memory access (DMA) via physical PCI to separately detect rootkit in kernel memory from another machine. However, system overhead, asynchronous kernel read/write, race conditions, and timing attacks are major challenges to this solution.

Table 2.4 – Comparison with related works on rootkit (RK) detection using different side-channel analysis techniques: HPC, DMA, Power consumption (Power) and EM.

	Article	WnP	Classi-fication	Baits	ML	DL	Sample size	Open source	Benign	User RK	Window size	Device under test
HPC	Numchecker [WK13]	-	-	✓	-	-	8	-	-	-	262.3 ms	32-bit Ubuntu PC
HPC	[SEE+17]	-	-	-	✓	-	5	-	-	-	45s	VMWare Windows 7 Intel
HPC	[JLC20]	-	-	✓	✓	-	4	-	-	-	2.91s	ARM Cortex-A53
DMA	Copilot [PJFMA04]	-	-	-	-	-	12	-	-	-	30s	PCI-compatible Intel PC Linux
DMA	Gibraltar [BGI11]	-	-	-	-	-	23	-	✓	-	20s	PCI-compatible Intel PC Linux
Power	[LMG+18]	-	-	-	✓	✓	5	-	-	✓	>5m	PC Windows 10 & Ubuntu 14
Power	[BJN+18]	-	-	-	✓	-	5	-	-	-	>1m	Dell OptiPlex 755 Windows 7
EM	**Chapter 4**	✓	✓	✓	✓	✓	9	✓	✓	✓	1.3s	ARM Raspberry Pi & MIPS Ci20

2.4 Research problems statement

Previously in this chapter, we discussed relevant detection solutions for malware, from classic to side-channel approaches, as well as their shortcomings. While some above-mentioned related works are successfully detecting malicious activity, there is a lack of research in the field on in-the-wild malware detection instead of proof-of-concept samples that may reflect only particular parts of realistic malware samples. Even more, none of the related works investigated the scenario of benign datasets and variants such as packed or obfuscated malware to test the robustness of their systems. Moreover, most of these works are using anomaly detection with low sample size, which has the advantage of detecting unknown threats, but is generally prone to raise numerous false positives. Indeed, anytime a new feature is introduced to the system, it is detected as malicious. Some only exploit an isolated malware execution environment (*e.g.* disabled outside connections), or an undefined malware execution environment, making it prone to evasion techniques and unclear if the malware actually executes malicious behaviors. Finally, none of them, to the best of our knowledge, are able to perform wide-ranging classification models in real-world malware analysis, *i.e.*, determine precisely the type, obfuscation, or variant of the malware infecting the system, due to their restricted malware dataset or analysis methods.

Most of the work on rootkit detection utilizes benchmark software to collect data from the system in both states: with and without rootkits. However, the purpose of benchmark software is to assess the relative performance of the system, normally by running a number of stress tests that are not particularly aimed against rootkits. The benchmarking tool consumes unwanted system overhead, and benchmarking data against rootkit will be less accurate and realistic. In our work, we present the methodology of using *baits* that are carefully crafted to trigger specific system behaviors. We show that this approach produces better accuracy in detecting and classifying rootkits. Previously, side-channel techniques for either malware detection or malware classification were developed, but none of them discussed solutions for both scenarios.

An autonomous malware detection and classification methods on embedded devices is apparently a necessity for malware analysts, forensic investigation and incident response. The primary intention of this thesis is to design methods that allow analysts to automatically investigate malware on embedded devices. In particular, it is a hybrid solution based on side-channel analysis and dynamic malware analysis to investigate

useful information detect and suspicious behaviors from IoT binaries. This has led to these research questions for detection and classification of malware in general and rootkit specifically.

RQ1 *How can we build and setup an IoT malware analysis and detection on embedded device?*

Since automated malware analysis techniques are matured on other platforms, but they have shortcomings that cannot be applied straightforward for IoT systems. Answering this question provides novel and enhanced methods of malware analysis on IoT system comparing to techniques proposed in literature.

RQ2 *If a malware analyst has a dataset of unlabeled binaries. Would it be possible to classify the dataset into labeled types, families, variants of malware or rootkits, obfuscation techniques used etc.?*

Answering this question will be apparently useful for malware analyst in practice. It allows analysts to automatically investigate malware on embedded devices.

RQ3 *Is it feasible to utilize EM for stealthy rootkit detection on embedded devices?*

Answering this question in order to solve the problem of stealthy rootkits that do not expose any malicious activities, thus often hinder from side-channel analysis.

3

Obfuscation Revealed: Leveraging EM Signals for Obfuscated Malware Classification

The content of this chapter, which is based on a joint work with Damien Marion, Matthieu Matsio, and Annelie Heuser, was published in the Annual Computer Security Applications Conference (ACSAC) 2021 [PMMH21] and Euro S&P 2021 [PMH21]. Presentations [1] [2] [3] have been made at both academic and industrial conferences, illustrating an overview of the paper.

Contents

1. ACSAC 2021
2. SemSecuElec 2021 https://videos-rennes.inria.fr/video/VJq91KPL6
3. Hardwear.io Security Trainings and Conference USA 2022 https://youtu.be/oCohqwfUpsQ

3.1 Introduction

In this chapter, we will present a novel approach of using side channel information to identify the kinds of malware threats that are targeting embedded devices. Using this approach, a malware analyst is able to obtain precise knowledge about the IoT malware type and identity, even in the presence of obfuscation techniques that may prevent static or symbolic binary analysis. We recorded 100,000 measurement traces from an IoT device infected by various in-the-wild malware samples and realistic benign activity. We preprocessed these traces and used them to train machine learning and neural network models. Our method does not require any modification to the target device. Thus, it can be deployed independently of the resources available without any overhead. Moreover, our approach has the advantage that it can hardly be detected and evaded by the malware authors. In our experiments, we were able to predict three generic malware types (and one benign class) with an accuracy of 99.82%. Even more, our results show that we are able to classify altered malware samples with unseen obfuscation techniques during the training phase, and to determine what kind of obfuscations were applied to the binary, which makes our approach particularly useful for malware analysts.

3.1.1 Motivation

In the new area of Internet of things (IoT), embedded cyber physical systems (CPS) are blooming. IoT and its applications is influencing the majority of our life's infrastructure, ranging from health/ food production to smart cities and urban management. Our IoT world is growing at a breathtaking pace, from 2 billions objects in 2006 to a projected 200 billion by the end of 2020, which is approximately 26 smart objects for every human being on Earth [4].

Naturally, they are increasingly targeted by cyber criminals due to their occurrences, availability, and the ability to use infected devices for further attacks on victim's architecture. IoT devices are given higher processing power, and some of them are running fully functional OS with multicore processors. This increases the attack surface by making them vulnerable to similar threats as general purpose computers, in particular, malware exploitation.

4. https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html

As the number of sophisticated malware samples constantly increases, malware analysts rely on mostly automated analysis systems for detection, classification, and forensic. While malware analysis and mitigation studies are on the rise, various challenges and unsolved problems are still remaining, some concerning the security of all computing systems, and some specific to embedded devices.

Analysis systems relying on static and dynamic features still have various shortcomings for malware analysts (previously discussed in 2.2). For example, static features can be easily manipulated by packing or obfuscating techniques [OSM11], whereas dynamic software-based monitoring may be detectable (*e.g.* by sandbox fingerprinting [YIT+16]) to terminate the malware execution, and thus hinder the capability of behavioral analysis [Sut13]. Moreover, unlike computer systems and servers, embedded cyber physical system may not have enough resources (such as computing power or battery) or accessibility (*e.g.* restricted access) to implement basic malware analysis solutions. All these factors make it difficult for malware analysts to automatically gain proper information about collected IoT malware samples (*i.e.* nature, evolution, etc.) to be able to mitigate the security risks.

In the scenario of anomaly and malware detection, a recently new direction is using hardware features to detect malicious behavior. Indeed, for each computation an electronic device is processing, it generates a set of residual productions that are potentially leaking side channel information (as discussed in Chapter 1, Figure 1.3), giving away a substantial amount of knowledge about the internal state of the device itself.

In the following sections, we concentrate on the EM field of an embedded device as a source for malware analysis, which offers several advantages. In fact, the EM emanation that is measured from the device is practically undetectable by the malware. Recent work [HHM+14] in cryptographic side channel analysis has demonstrated that it is able to detect the presence of electromagnetic monitoring using feedback from LC oscillators on the hardware level of the device. However, applied to our scenario, this requirement is impossible to be reached by malware on a software-level. As a result, common malware evasion techniques cannot be implemented directly. Another advantage is that monitoring EM emanation does not require the alternation of the device in order to analyze it. In other words, it does not rely on device architecture and OS or without any computational overhead.

Previous works using EM emanation and power consumption investigated the detection of malware [KSN+19b, ADCC18, RGF+19], abnormal behavior [WZH+18], or

distinct control flow tracking [ZPKA18]). These works are in very constrained and static systems (in the case of anomaly detection) and mostly analyzed only laboratory-grown malware samples without any variations. This naturally raises the question of realistic evaluation:

If a malware analyst has a dataset of unlabeled binaries. Would it be possible to classify the dataset into labeled types, families, variants of malware or rootkits and obfuscation techniques used?

While malware detection concerns with the process of detecting the presence of malware whether a specific program is malicious or benign [KKV11], malware classification refers to the process of distinguishing the unique types of malware from each other based on the identified malicious patterns. In this work, we derive a framework that is capable to *classify real-world* malware samples including protection mechanisms against static and dynamic malware analysis using only EM emanation from the device. Furthermore, we aim at classifying into malware types, family, possible protection mechanism, previously unseen variants, or even distinct executable classification, which makes our framework particularly suited for malware analysts.

3.1.2 Our contributions

In summary, this chapter makes the primary contributions:

1. **Obfuscated ARM malware dataset.** We put in place a representative set of malicious ARM binaries, on which we applied various obfuscation techniques. By integrating obfuscation techniques against software-based malware analysis systems, we are able to investigate if these techniques also hinder analysis based on EM emanation, and if we can distinguish the applied obfuscation procedures independent of the executed binary. To the best of our knowledge, this has never been studied before, provides the largest distinct malware sample dataset, and is crucial for practical malware analysis.
2. Our **real-world malware testbed** allows executing malicious binaries while having internet access, spoofing C&C servers (in case they have been taken down), and protecting the host environment from infections of the malware. To not bias our experiments and not allow user-fingerprinting by the malware to evade detection, our framework includes a randomized user environment. We set up an experimental framework to measure the electromagnetic activity of an embedded

bare metal multiprocessor hardware environment running a fully-functioning Linux OS in a random initial state. This includes a data acquisition and malware testbed platform. Using this platform, we are able to record a vast number of electromagnetic traces from a device that was either infected by a real-world malware sample belonging to different types, or in a clean state performing random activities (i.e. not only in an idle state).

3. **Generic side-channel analysis environment.** Our approach does not make any alteration to the target device. In particular, we do not utilize software monitoring, precise triggering, or produce any additional overhead on the device. In our experiments, a multiprocessor hardware environment was used to run a fully-functioning Linux OS to be applicable to realistic IoT systems in the wild, use a random initialized analysis environment, and perform *practical* benign activities.

4. **Robust and resistant analysis techniques.** We derived a methodology on how to effectively extract suitable information about the binary, taking as input the raw EM traces. Our approach consists of preprocessing by selecting the most relevant frequency bands over time and then classifying in various scenarios with neural network models and simplistic machine learning models. Results show that our methodology is resistant to obfuscation techniques such as virtualization, packing, and static code rewriting.

5. **Experimental scenarios compliant to malware analysts.** We compile various scenarios, each of them represents a real world malware analysis use case: type and family malware classification, exact malware executable profiling, virtualization and packer identification, obfuscation classification, and the classification of *unseen* obfuscated variants. These scenarios go way beyond the detection scenarios considered in the state-of-the-art. Also, using our analysis on obfuscation, we are the first to discuss the difficulties of malware evasion against our methodology.

6. **Open-source.** The resources related to this work are publicly available[5]. We provide our source code, datasets, malware classification models, and raw results of our experiments.

5. https://github.com/ahma-hub/analysis/wiki

3.1.3 Roadmap

The following sections are organized as follows. The generation of malware dataset and its obfuscated variants will be described in Section 3.2. We detail our framework in Section 3.3, which includes our approach towards dynamic malware execution, data acquisition and preprocessing, and malware classification based on neural networks, Naive Bayes, and SVM. In Section 3.4 we detail our experimental setup. Results and discussion on malware classification scenarios are given in Section 3.5, and Section 3.6 concludes.

3.2 Real-world IoT malware collection

One of the most important blocks in building malware analysis systems, is the construction of input datasets. In this section, we aim at being realistic and to include common obfuscation techniques that are used by today's malware designers, frequently used to avoid detection by hiding known signatures, behaviors, or by making it more difficult to reverse engineer. While general purpose computers usually run on common architectures such as x86, embedded IoT devices are developed for a broad range of architectures, such as ARM, MIPS, PowerPC, etc. Major problems related to the diversity of the possible target IoT environments are described in [CGFB18].

This work, for the sake of simplicity, only supports ELF ARM 32-bit architecture which can be executed properly on Linux OS.A well-known technique IoT malware authors use to get over the target diversity is to cross-compile malware source code in multiple architectures. Then, they deploy all malware binaries on the target device at once. Thus only one binary will be executed properly with the correct supported architecture. In the following, we discuss the creation of our malicious and benign dataset.

3.2.1 Malware dataset

To understand the scope of ARM malware on IoT devices, we conduct a study on 4,790 32-bit ELF ARM malware samples collected from Virusign. [6]. Thereafter, we extract AV labels for each sample from VirusTotal [7] reports to obtain malware variant

6. https://virusign.com
7. https://virustotal.com

name. To get normalized labels that can be used for classification, we use AVClass. It selects the top ranked corresponding family name through plurality vote. On collected dataset, AVClass was able to associate the collected malicious ARM samples to 19 different families. *Mirai* (43.5%) and *Bashlite* (35.8%) dominate the dataset. These 2 DDoS malware variants have been publicly open sourced, and resulted in large use, various modifications and variations of increasing complexity. This result is consistent with previous epidemiologic studies of IoT malware [CGFB18, MBM+18] and online real-time statistics such as Malware-bazaar [8].

To construct a representative malware dataset, we use 3 different well-known malware variants: DDoS (*mirai*, *bashlite*), Ransomware (*gonnacry*), and kernel rootkits (*spy*, *maK_It*). In our study, we reviewed their codebase to understand their *modus operandi* described as follows:

Bashlite creates TCP communication to the C&C server, then exchanges IRC commands and messages. Control commands and common behaviours of *bashlite* consist of scanner, password bruteforce, TCP and UDP Flooding.

Mirai adopted concepts from previously discussed *bashlite*, with improved features such as anti-debugging, self-hidden, data obfuscation and botkiller which terminates bots from other families.

Gonnacry is an active ransomware variant that is open sourced in Python and C for research purpose. It finds all files in user's home directory, then encrypt those belong to a hard-coded list of extensions. The malware starts its encryption routine and creates a desktop file that will be useful for the decryptor to access the path, key and IV. In addition, we generate multiple malware variants from original *gonnacry* by extending with other crypto schemes such as AES and DES, in addition to the original Blowfish encryption algorithm.

Keysniffer is a Linux kernel module which has functionalities to hook and record keys pressed in the keyboard events of *debugfs*.

MaK_It shares the same rootkit ability to *spy*, with addition of kernel module self-hidden, packet sniffer and reverse-shell backdoor.

On obfuscation of the malware dataset, we applied each binary with every obfuscation transformations to enrich our datasets with static code rewriting that consists of *Opaque predicates*, *Bogus control flow*, *Instructions substitution* and *Control-flow flattening*, and dynamic code rewriting such as *Packer* and code *Virtualization*. To evalu-

8. https://bazaar.abuse.ch/

ate the robustness of our methodology and to explore possible protection techniques against side-channel monitoring, we apply state-of-the-art packers and obfuscators like UPX [OMR], Tigress [CMM+] which implemented stack-based interpreter VM, and Obfuscator-LLVM [JRWM15].

It is worth mentioning that while there are numerous packers existing for PE formats, only a limited number of ELF packers have been published. There are fewer packers for ARM than ELF Intel architecture, and most of them are unmaintained proof-of-concept projects. The only exception is UPX, a popular open source packer to compress the size of executables (*e.g.* Stuxnet malware also uses DLLs packed with UPX [Blu12]).

3.2.2 Benign dataset

The selection of a benign dataset is significant to not only increase the difficulty of detection but also ensure the quality of classification. The benign samples must generate random activities such as computations, background processes with malware-free, or usual activities on embedded IoT devices. We generate benign datasets by collecting ARM executables from a fresh installation of Linux system. This similar approach of constructing benign dataset has been conducted from other generic malware studies [LAS15, BLS13] outside from EM analysis. Furthermore, we complement benign executables under a layer of UPX packer to blend benign samples with packer. Additionally, the usual benign activities for an embedded IoT device were recorded such as Linux utilities, device sleep, photo capture, network connections, as well as long duration of executable runtime such as media player, camera capture, video encoder, data backup, data (de)compression (Table 4.1). This collection varies from short to long duration of executable runtime, and from low to high CPU consuming processes.

Notably in previous studies using EM emanation, the construction of benign dataset is not considered, or benign activity is only associated with either free-malware activities or benchmark software [NSA+17, SEE+17, KSN+19a, SNA+20, CKM21]. It simplifies detection drastically and is not realistic where malware, update services as well as other IoT activities may already have distinct behaviors.

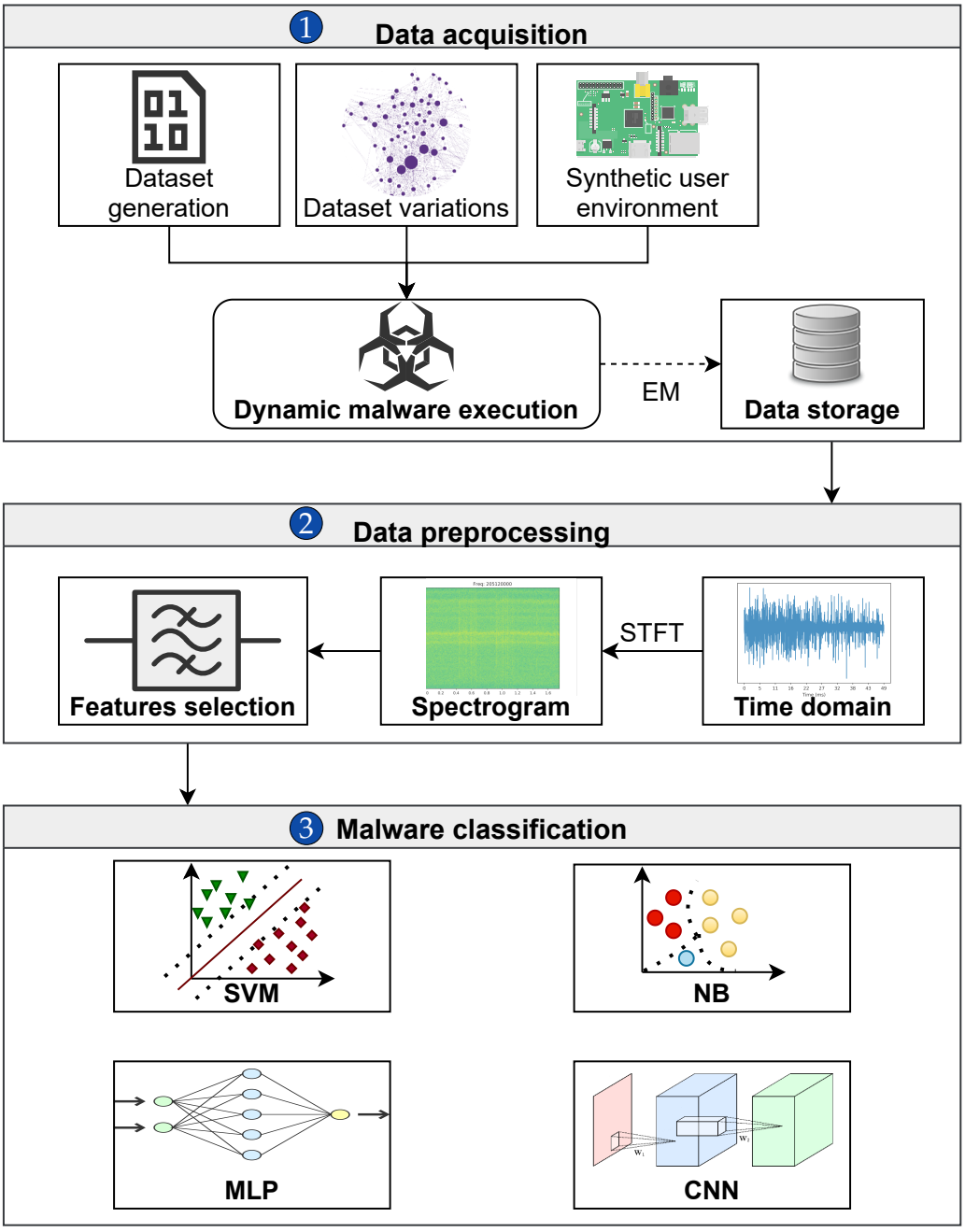


Figure 3.1 – Illustration of the proposed IoT malware classification framework Automated Hardware Malware Analysis (AHMA). 1 Data acquisition: from malicious binary execution to (noisy) EM measurements 2 Data preprocessing: from (noisy) EM measurements to exploitable data. 3 Malware classification: from exploitable data to malware labels.

Table 3.1 – Linux binaries and activities used in the benign dataset

Activities	**Executables**			
Linux Utilities	mknod	vdir	more	find
	zgrep	ls	cat	findmnt
	zmore	as	ed	rm
	touch	dmesg	sleep	cd
	less	grep	objdump	
Network	wget	hostname	ss	ip
Compression	gunzip	bunzip2	bzip2	tar
	uncompress			
Data backup	dd			
Scripting	python			
Photo & Video	raspistill	raspivid		
Video Encoding	MP4Box			
Audio player	mpg321			

3.3 Real-world malware analysis framework AHMA

Our analysis consists of three main phases:

1. *Malware execution and measuring EM emanation*: Within our secured and randomized setup, we execute the malware samples while measuring EM emanation from the outside of the device without manipulating any internal operation.
2. *Data analysis and preprocessing*: The raw captured measurement traces include a large amount of noise. Thus, we transform our data by the time-frequency domain, and select most suitable frequency bands.
3. *Malware classification*: Given the 2D data, we derive deep neural models and compare them to more simplistic machine learning algorithms.

Our malware selection encompasses three types, which are representing common malware targeted on IoT devices in the wild: DDoS, ransomware, and kernel rootkits. To be compliant with real-world scenarios and to investigate the robustness of

our approach, we extend our dataset by applying various software analysis protection mechanisms to the malware binaries. Including obfuscation techniques gives us new outcomes that have never been studied in the state-of-the-art. First, we determine if code obfuscation techniques (*e.g.* code rewriting, virtualization, etc.) can actually hinder our approach. Second, we derive the robustness of our approach against unseen malware samples, by conducting a scenario where our system tries to predict samples with unknown obfuscation. This evaluation is of great importance due to the rapid evolution of malware variations and obfuscation created by attackers. Finally, we investigate if we are able to predict if an obfuscation has been applied, and to which technique it belongs.

We propose an IoT malware classification framework (AHMA), that takes as an input an executable and outputs its predicted label by solely relying on EM side-channel information. Figure 3.1 illustrates our workflow, which will be detailed within the next few subsections. First, we define our threat model, and then we describe the collection of EM emanation while the malware is executed on the target device. We setup an infrastructure to be able to execute malware with a realistic user environment while preventing any infection of our host controller system. Thereafter, as the collected data is very noisy, a preprocessing step is required to isolate relevant informative signals. Finally, using this output, we train neural network models and machine learning algorithms in order to classify malware types, binaries, obfuscation methods, and detect if an executable is packed or not.

3.3.1 IoT malware classification threat model

In general, malware analysts collect datasets of malware from online feeds of intrusion detection systems and community database as illustrated in generic malware analysis workflow (Figure 3.2). The very first and important step is *Malware analysis* where feeds are filtered, analyzed and classified.

In this threat model, malware analysts possessed real-world malware sets and physical target devices. Real-world malware feeds presumably contain unknown variants which exploit evasion techniques and attack a wide range of Linux devices (*e.g. Mirai* variants actively infect Linux IoT devices and obfuscate its encoded strings). By leveraging the combination of bare-metal analysis and EM (Fig. 3.1), it avoids the necessity of software analysis tools update such as sandbox, hooking and anti evasion

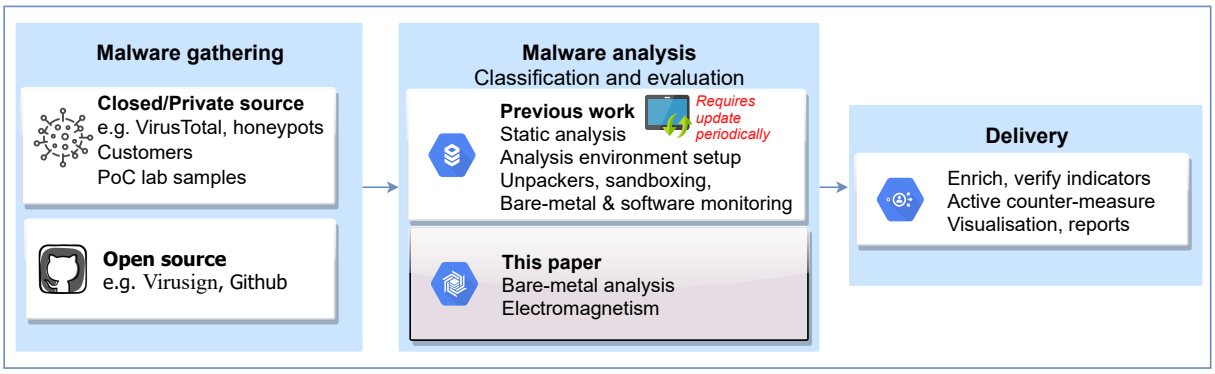


Figure 3.2 – Generic malware analysis workflow

techniques. Moreover, malware analysts are fully able to control and customize their analysis environment in the most advantageous way by simulating network traffic and setting up user synthetic environment. Therefore, simulating side-channel noise, distant EM monitoring, random user activities, multiple malware attack on the same device simultaneously are not their consideration. The proposed analysis framework supports a generic OS, so that it is applicable to any kind of malware with UNIX, from simple BusyBox utilities and Bash commands to ELF malware and high-level scripting (*e.g.* Python). Unlike FPGA-based systems malware detection approach that executes the samples on restricted bare-metal environment. The framework should not use any system modifications like the traditional malware analysis framework such as hooking on the target device to avoid being evaded by the malware.

3.3.2 Data acquisition by dynamic malware execution

Typically, malware analysts rely on static/signature-based analysis or dynamic sandboxes in order to gain information about malicious binaries. We propose a solution which we believe is immune to obfuscation and evasion techniques, by black-box monitoring side channel information to detect and classify malware samples at the hardware-level of a bare-metal IoT device running a fully-functioning operating system. Since network traffic analysis was considered as side-channel leak and already studied in previous work [MBM+18], this chapter will focus on the aspect of EM emanation specifically. The first part of our framework relates to the data acquisition that can be divided into *dynamic malware execution* and *electromagnetic monitoring*.

Realistic malware execution environment

Traditional dynamic malware analysis solutions were built upon virtualization machines or emulation, which leave numerous system artifacts for evasive malware to exploit [RKK07]. In particular, sophisticated malware authors exploit fingerprints inside analysis system (*e.g.* number of cores, network MAC address, etc.) to avoid malware analysis or detection. Besides, in-guest monitoring components to observe malware behaviors (*e.g.* syscall/API hooking, registry monitoring, etc.) also leave artifacts for malware evasion. One way to prevent these artifacts is to patch the exploitable components of the virtual system such that they are indistinguishable from a real machine. However, this approach only guarantees known evasion techniques. Another way is to implement a transparent analysis system that performs hardware virtualization extensions, to keep the CPU execution semantics of the host. It is fundamentally infeasible to make it perfectly transparent, since this system can be detected by timing attacks and CPU identification (previously discussed in 2.1.2).

Furthermore, one can analyze malware on a fresh operating system installed on a real machine which is referred to as bare-metal dynamic malware analysis. A pure bare-metal system is limited to the disk access and network activities in general. However, such an analysis system, is not suited to perform an in-depth behavioral analysis of stealthy malware such as passive rootkits and fileless malware.

To overcome these difficulties, we propose an infrastructure leveraging side-channel information from a bare-metal sandbox rather than emulators or virtual machines. In this malware execution environment, we propose a particular use case for embedded devices where all outgoing network activities will be routed to testbed servers that resides in a local network. Additionally, an unrealistic configuration will not be able to trigger malware activities, so we propose spoofed C&C servers which receive network connections and randomly return control commands, as well as a synthetic user environment dedicated to embedded malware, which will be detailed in Section 3.4.1. Due to a common problem in dynamic malware analysis, the C&C servers of the botnet in the wild are taken down after a short period of time. Moreover, in order to prevent analysis information leakage or infection to the analyst's host machine, a local switch router and firewall under a controlled network are implemented. We have confidence that bare-metal malware analysis does not expose any instrumentation indicators, and side-channel information will give us a snapshot of malware behavioral analysis.

Electromagnetic monitoring

In the data acquisition procedure (step 1 in Fig. 3.1), the controller machine sends binary samples to the target device (in our case we transfer samples to the Raspberry Pi via SSH connection). The controller server is responsible for distributing samples to one or more embedded devices as well as collect recorded EM traces. We use a *malware initiator* to send trigger signals from the target device to oscilloscope through GPIO pin-outs and let the oscilloscope start its recording session. Note that the intended users are malware analysts who are unrestricted to set up malware initiators or choosing the appropriate device. The *malware initiator* is only executed at the beginning of the analysis to trigger the oscilloscope once while malware and benign samples as well as the OS are kept untouched. This monitor scheme also means that our approach does not require data synchronization which is common in side channel analysis. Thereafter, the *malware initiator* spawns the according sample immediately and the controller machine collects recorded EM traces from oscilloscope in block mode through USB protocol. Recorded data traces will then be preprocessed and analyzed with malware classification models (step 2 and step 3 in Fig. 3.1 respectively).

3.3.3 Data analysis and preprocessing

In this subsection, we will describe the procedure we used to collect and process electromagnetic traces from a target device executing malware. First, we obtain the raw signal directly from the oscilloscope. Then, we calculated the Short Time Fourier Transform (STFT), which represents the evolution of the frequency content of the signal over time.

The second step in our complete framework preprocesses the collected (noisy) EM measurements (see Figure 3.1). This step is mandatory as the CPU of an electronic device executes programmed instructions every clock cycle, which will provoke variations in its internal circuitry. Moreover, modern target devices have multicore architectures, so the recorded EM activity is a mixture of various processes, and it is impractical to identify the process responsible for each observed variation from the electromagnetic trace itself. The strong signals existing in the system, like the processor or memory clock, will act as a carrier, that will be amplitude or frequency modulated by the executed instructions [PZCW16]. This modulation will cause EM emanation, that leaks from any element of the device.

It has been shown [SNZP16] that it is possible to monitor the EM spectrum to profile a program execution on the system. Indeed, each loop present in the program will generate repetitive activity that will produce "spikes" at frequencies that correspond to the time the CPU of our monitored system will need to execute one iteration. Each executed program has a specific loop pattern, that is revealed by peaks in frequency.

This is why we preprocess the raw EM data to represent the fluctuations of the frequency content during the measurement time of the traces. For this purpose, we computed the spectrogram of the signal by taking the magnitude squared of the STFT:

$$spectro\{x(n)\}(m,\omega) = |\sum_{n=0}^{N} x(n)w(n-m)e^{-j\omega n}|^2. \quad \text{(Eq. 3.3.1)}$$

A STFT breaks the signal into small segments of equal length, and performs a Fourier transform on each of the segments. Here, $x(n)$ represents the input signal at time n, ω the frequency, m the segment index, N the number of recording points, and w the window function. In our case, the window function splits the signal in chunks of length M, with an overlap O using default SciPy configuration $O = M/8$. The evolution of a signal frequency over time can easily be represented as an image.

Even though the spectrogram improves our data representation in terms of noise reduction, it also greatly increases the amount of data. Using the full spectrogram will drastically increase the amount of time and space resources needed for classification, if even possible. We therefore apply feature extraction to the spectrogram using NICV (Section 1.3.4). In particular, we apply the NICV to the spectrograms in order to identify the frequency bandwidths that may reveal behavioral information about the binary. In fact, applying the NICV to the spectrogram gives the variance of each spectrogram's feature between labels.

Let us denote X as a spectrogram of dimension $D \times F$ with D being the number of time features and F being the number of frequency bandwidths. Let Y be the label, *e.g.* the type of the malware. The computation of NICV(X, Y) results in a matrix of dimension $D \times F$ (see Eq.1.3.3). Next, from the NICV matrix, we select the ϵ frequency bands corresponding to the highest mean over D:

$$F_{\text{extract}} = \{\text{argmax}_{\epsilon}(\frac{1}{D}\sum_{d=0}^{D-1}[(\text{NICV}(X,Y))_d^F]\} \quad \text{(Eq. 3.3.2)}$$

with argmax_{ϵ} being a function that returns the ϵ indexes with the highest values and

$(\cdot)_d^F$ represents the dth column of the matrix over all frequencies. Accordingly, F_{extract} contains the list of the ϵ indexes of the conserved frequencies. Note that, we extract the complete frequency band of the spectrogram rather than its multiple chunks, which is mainly motivated by possible time delays or desynchronization of unseen data in feature extraction process due to the absence of an exact triggering process.

3.3.4 Malware classification model architectures

Table 3.2 – Proposed MLP architecture for ARM malware classification

Layer	Size	Filter	Activation
Flatten	spectrogram_size	_	leaky RELU
Dense	500	_	leaky RELU
Dense	200	_	leaky RELU
Dense	100	_	leaky RELU
Dense	N	_	softmax

Given the most informative spectrogram bands, our main objectives are to analyze to what extent a malware analyst is able to: (*i*) retrieve the type or family of the malicious binary, (*ii*) identify precisely which binary was being executed, (*iii*) classify the obfuscation technique, and (*iv*) classify the malware family even with a previously unknown obfuscation technique. Based on that, we assume that the analyst has a dataset of labeled malware binaries on which he can build supervised classification models. These models are then used to classify new unknown binaries.

Deep learning recently achieved remarkable results in numerous fields, particularly thanks to the improvements brought to the neural network paradigm (previously introduced in subsections 1.4.2 and 1.4.3). Neural networks are particularly effective for computer vision and pattern recognition, and that is the reason for investigating their efficiency in classifying the spectrograms of monitored devices' EM activity. We defined two distinct neural networks architectures in Table 3.2 and 3.3, then compared their efficiency on our classification tasks.

The first architecture is a simple MLP, which takes as input flattened spectrogram bandwidths. We use three hidden layers with a decreasing number of neurons as we go

Table 3.3 – Proposed CNN architecture for ARM malware classification

Layer	Size	Filter	Activation
Convolution	64	7×7	leaky RELU
Max Pooling	64	2×2	_
Convolution	128	3×3	leaky RELU
Convolution	128	3×3	leaky RELU
Max Pooling	128	2×2	_
Convolution	256	3×3	leaky RELU
Convolution	256	3×3	leaky RELU
Max Pooling	256	2×2	_
Dense	128	_	leaky RELU
Dense	64	_	leaky RELU
Dense	N	_	softmax

deeper into the network. We chose a *Leaky RELU* activation function for each hidden layers as it revealed itself slightly more efficient than the *RELU* and *sigmoid* functions in our experiments. Since we wanted to perform a multi-class classification, the output layer ends with a *softmax* function. The loss used to calculate the error in prediction is a categorical cross-entropy.

The proposed CNN architecture (Table 3.3) is more complex, but still rather simple compared with the state-of-the-art networks used for image recognition. It is constituted of a stack of three atomic blocks where each block is made with one or two convolution layer(s), followed by a Max Pooling layer. The number of filters used in the blocks is increasing (but their size is decreasing) as we go deeper in the network. To produce the prediction, we end the network with a multilayer perceptron composed of 2 hidden layers, and a softmax output layer. Once again, we use categorical cross-entropy as a loss function.

We furthermore study the effectiveness of more simplistic and less resource demanding machine learning techniques such as Naive Bayes (NB) and Support Vector Machines (SVM). As NB and SVM (or in general most machine learning algorithms)

are prone to the curse of dimensionality [Bel57], meaning that they do not scale well with the number of input features, we do not take as input the selected bandwidths as for the neural network models, but perform feature transformation on the spectrogram using LDA (previously discussed in 1.3.4). We apply these two machine learning algorithms to our data in order to get preliminary results quicker than with deep learning algorithms. The data used by the SVM, the NB and the deep learning algorithms is the same. We only apply an additional pre-treatment, a Linear discriminant analysis (LDA), on the data provided to the machine algorithm. LDA is a commonly used tool together with machine learning algorithms that allows one to (drastically) reduce data dimension. On the other hand, in most cases, LDA (or any feature transformation) is not suitable for neural network models, as features are transformed into different feature spaces and dependencies between features (in particular, shapes and patterns) may be disconnected and harder to learn.

3.4 Experiments

3.4.1 Data aquisition components

Target device

Evaluation of target device is critical for EM side-channel analysis. We determine three main requirements:

Definition 2: Main requirements for a target device

— It must be a multi-purpose embedded device to support as many collected malware samples as possible, rather than a specific set of malware or device

— Its CPU must be a prominent architecture to avoid the lack of compatibility of emerging IoT malware

— It must be vulnerable to EM side-channel attacks.

We select Raspberry Pi 2B as a target device with 900 MHz quad-core ARM Cortex A7, 1 GB memory. Since its ARM architecture, size, power consumption, and cost-effectiveness make it a good candidate for any kind of embedded and IoT scenarios, including prototypes and developments. Our research focuses on a very general malware classification challenge rather than a narrowed solution to any specific device,

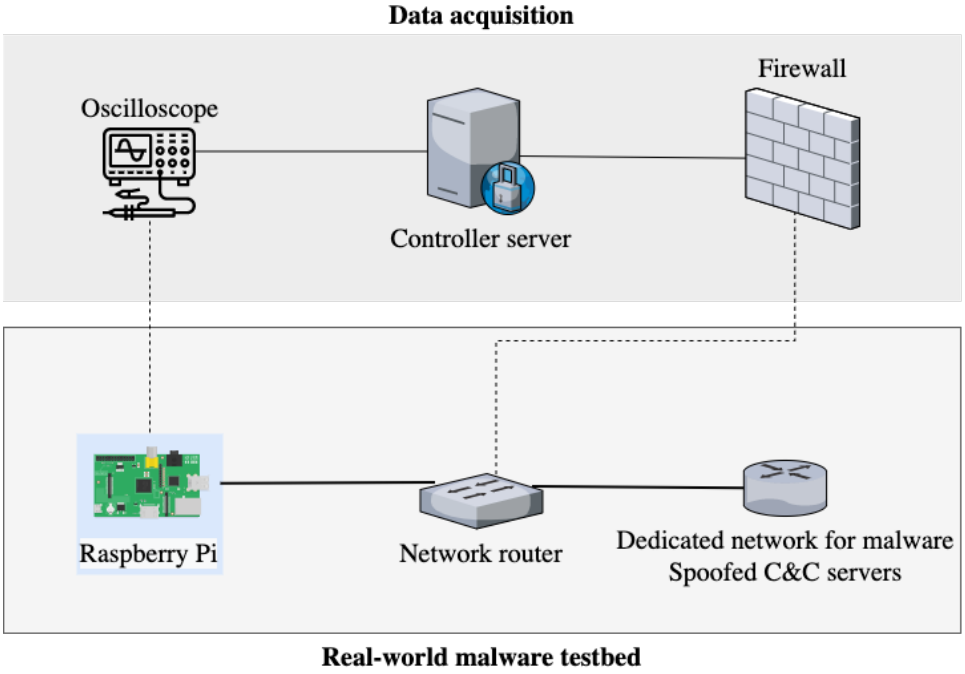


Figure 3.3 – Overview of the proposed infrastructure: Experimental setup of malware testbed and data acquisition.

in particular, as related works did not show diversity in results or analysis techniques across multiple devices (*e.g.* [SNA+20]).

It has been shown in previous works [WZH+18, FI17] that cryptographic and anomaly activities are successfully distinguished by leveraging EM signals from the Raspberry Pi 2B. Later versions of Raspberry Pi have several changes to Wireless LAN as well as metal shielding on the processor, that may affect the EM observation.

By not limiting the capabilities of the infrastructure with restricted bare-metal firmware, the Raspberry Pi is deployed with a fully-functioning Raspbian Buster OS of Linux *4.19.57-v7+ armv7l*. A device under test with a fully functioning OS and multiple cores is to determine if it is possible to handle malware in more complex scenarios where malware is mixed with background processes, services, and interrupts, resulting in noisy EM traces.

To prevent the detection of any artificial artifacts by evasion techniques and maintain a realistic environment, all background services are kept to their default with more than 100 concurrent processes and services. Additionally, no adjustments, overclock-

ing, or tuning of the processor clock rate are applied to the device.

Malware testbed environment

To generate a practical analysis environment that can trigger real malware, we applied different tools and techniques. We created honeypot directories under the root folder, home folder, etc. Each malware execution will have a random initialized environment consisting of different valid files and extensions to assure that ransomware will be properly executed while not biased towards the recorded traces. Further, the *Evemu* library was adapted to emulate random keyboard input events which trigger the keylogger functionality.

Besides, most IoT botnets architectures consist of one Command and Control (C&C) which is continuously connected (except peer-to-peer botnet). To support our malware dataset, consist of *Mirai* and *Bashlite*, we implement a synthetic environment of central spoofed C&C server model. C&C servers are adopted to randomly deliver different commands to the botnet client in multiple attack scenarios (Fig. 3.3). To trigger a broad range of malicious activities, in each experiment the following commands are delivered: network scanner, flood targeted victim network in TCP/UDP, hibernation, or self-terminate etc. Furthermore, we installed multiple virtual machines in the same local network for absorbing network attacks coming from malware. The nature of IoT malware samples and the execution coverage on software level in the device under testing are not altered, so that we presume no anti-analysis evasion techniques can survive during the bare-metal malware analysis.

Electromagnetic signal acquisition

We monitor the Raspberry Pi under the execution of benign and malicious datasets using a low-to-mid range measurement setup. It consists of an oscilloscope, Picoscope 6407, with 1GHz bandwidth connected to a H-Field Probe, Langer RF-R 0.3-3, where the EM signal is amplified using a Langer PA-303 +30dB (Figure 4.4.2). To capture the long-time execution of malware in the wild, the signals were sampled at only 2MHz sampling rate.

The activity of the Raspberry Pi, when executing malware or generating benign activity was recorded at a sample rate of 2MHz for 2.5 seconds. It has been chosen empirically based on (but not limited to) the constraints of the data acquisition compo-

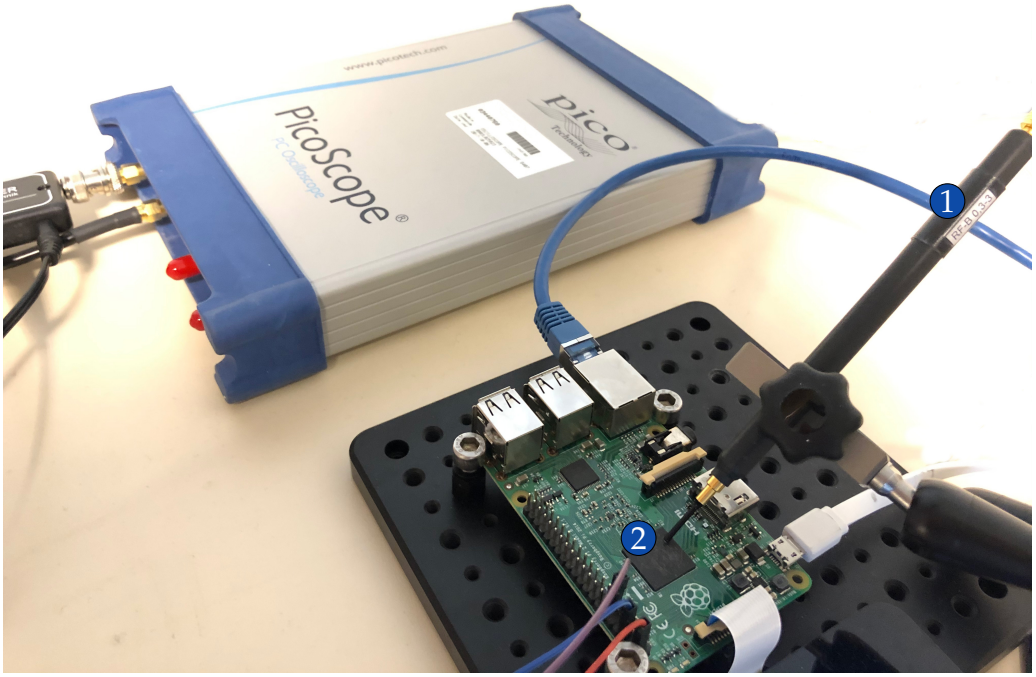


Figure 3.4 – Probe setup consists of a H-Field probe (1) placed 45 degree above the Raspberry Pi processor (2).

nents: imprecise trigger, and malware characteristics (*e.g.* sleep time with no activity of *Mirai*). The duration of 2.5 seconds is enough to obtain exploitable features for classification. We collected 3 000 traces each for 30 malware binaries and 10 000 traces for benign activity. Thus, a total 100 000 traces were recorded, then we computed their short-term Fourier transformation, as described in Section 3.3.3.

The feature selection process using NICV on one spectrogram is illustrated in Figure 3.5. The left side shows the NICV where the right side highlights the selected frequencies that correspond to the 20 frequencies with the highest NICV mean over time (D). The selection of complete bandwidth, and not discrete features of the spectrograms, makes the classification process more resistant to potential de-synchronisation issues.

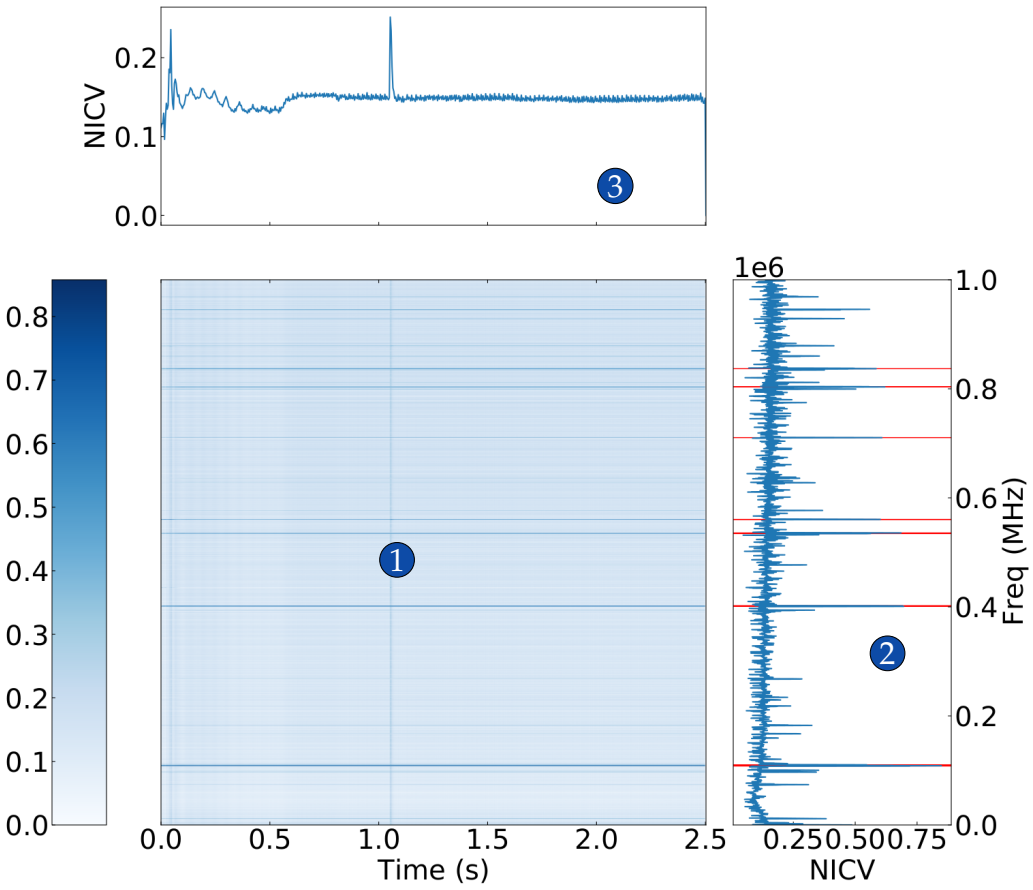


Figure 3.5 – NICV (1) and in red the 20 selected frequencies ($\epsilon = 20$) on the mean over the time (2) and the mean over the frequencies (3)

3.4.2 Classification framework

Model input

In Section 3.3.3, we described how we generate the spectrograms from the EM activity recorded by the oscilloscope. We measure 500 000 points to get a 2.5s recording with a sampling rate of 2MHz, which has about 8MB per trace. The Graphical Processing Unit (GPU) RAM necessary to run a Keras model is equal to the size of the parameters of its biggest layer multiplied by the batch size. To run the MLP described in Table 3.2 for example, we would need $500 \times input\_size \times batch\_size$. As explained above, it was necessary to generate tens of thousands of spectrograms to train our neural network models. We choose $(M, O) = (8192, 4096)$ as parameters for the STFT. To reduce

the dimension of the analyzed data and reduce the noise, we apply the feature extraction process described in Subsection 1.3.4 to conserve only ϵ different bandwidths. We tested $\epsilon \in \{4 \times i\}_{0<i<8}$ to determine for each experiment the number of conserved bandwidth ϵ_{opt} that reaches the best accuracy. If we used the whole spectrum with a batch size of 32, that would require about 13GB of RAM. It is worth mentioning that it could work in more powerful environment, but we were limited in our setting to 11GB effective GPU RAM.

Eventually, our dataset is split into three parts:

1. A test dataset which will never be used during the training phase of the model,
2. A validation dataset to assess the efficiency of the model on unseen data, and
3. A training dataset.

We kept, by default, 20% of the dataset for testing and used the 80% remaining for training and validation. These whole datasets were too big to be fitted into the memory of the graphical process unit, so we had to create batch generators that accessed the hard drive and fed small subsets of the data in real-time.

Training procedure

As performance assessment we used the accuracy of the classifier, which is defined by the average percentage of correct label predictions (previously discussed in Chapter 1.4.4). The neural network models have been trained over 50 epochs, where we stored the model according to the highest validation accuracy. In our setup using one RTX 2080 Ti GPU, CNN took around 50s per epoch, which gives $50 \times 50s = 2500s = 41$ min of training time, MLP performed 1 epoch within 9s, that gives $50 \times 9s = 2500s = 7$min. Testing one sample takes roughly 0.75s for MLP and 0.27s for CNN. We trained the networks on the training dataset. We found out that the accuracy increased at the beginning epochs and decreased afterwards, so we stopped the training after 50 epochs and saved the only best model. After the training phase, we calculate the accuracy of the model on the test dataset.

NB and SVM are much less resource demanding than neural network models, and can be computed using standard CPU computation systems. On our system with an Intel(R) Xeon(R) Silver 4214 @2.20GHz with 24 cores and 128GB RAM, NB took 0.14s to train, and SVM 18.90s. The testing of one sample took for NB around 1μs, and for SVM between 1ms and 6ms depending on the number of features considered.

Results were obtained using the Keras backend of TensorFlow running on one RTX 2080 Ti GPU for MLP and CNN and Scikit-learn [PVG+11] library (version 0.23.2) for NB, SVM and LDA.

3.5 Results and discussion

3.5.1 Experimental results

A synthesis of the results we obtained can be found in Table 3.4. The main column indicates the name of the scenario. In the first column, we state the number of outputs (*i.e.* classes) of the network. Finally, the other columns show the accuracy with the optimal number of bandwidths, as well as the precision and recall of the two neural network models, and the two machine learning algorithms on the test dataset. Details about the dataset construction of each scenario can be found in Table A.1 in the Appendix.

		MLP			CNN			LDA + NB			LDA + SVM		
Scenarios	#	**AC** [ϵ_{opt}]	**RC**	**PR**	**AC** [ϵ_{opt}]	**RC**	**PR**	**AC** [ϵ_{opt}]	**RC**	**PR**	**AC** [ϵ_{opt}]	**RC**	**PR**
Type	4	99.75 [28]	99.83	99.85	**99.82** [28]	99.89	99.88	98.01 [24]	98.84	98.35	98.08 [24]	98.71	98.76
Family	6	99.57 [28]	93.13	93.11	**99.61** [28]	98.61	98.60	97.19 [28]	90.78	90.99	97.27 [28]	91.12	91.14
Virtualization	2	95.60 [20]	95.76	94.99	**95.83** [24]	96.19	95.14	91.29 [6]	91.07	90.49	91.25 [6]	90.69	90.62
Packer	2	93.39 [28]	93.36	93.06	**94.96** [20]	94.94	94.70	83.62 [16]	83.13	83.08	83.58 [16]	83.08	83.04
Obfuscation	7	73.79 [28]	72.77	72.79	**82.70** [24]	82.08	82.08	64.29 [10]	63.08	63.01	64.47 [10]	63.22	63.00
Executable	35	73.56 [24]	74.66	76.75	**82.28** [24]	83.08	83.11	70.92 [28]	72.29	71.94	71.84 [20]	72.47	72.32
Novelty (family)	5	88.41 [16]	92.35	91.01	**98.85** [24]	98.59	98.59	98.25 [28]	98.69	98.53	98.61 [28]	98.90	98.82

Table 3.4 – Accuracy (AC), recall (RC) and the precision (PR) obtained with MLP, CNN, LDA + NB and LDA + SVM applied to several scenarios. ϵ_{opt} gives the value ϵ, the number of extracted bandwidth (Eq. 3.3.2), obtaining the highest accuracy. Bold numbers indicate the highest accuracy on the testing set per scenario. The column "#" gives the number of classes per scenario.

Type classification

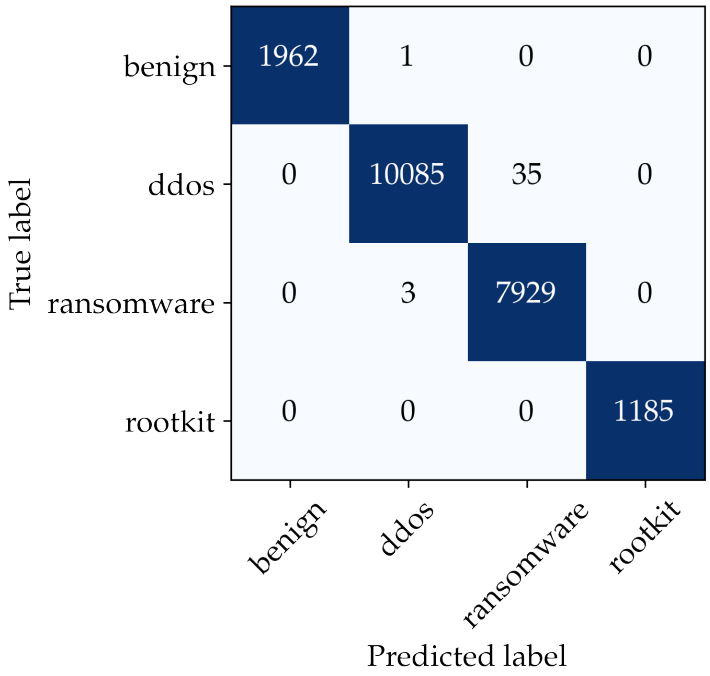


Figure 3.6 – CNN type classification (Accuracy 99.82%). Including ddos: *mirai*, obfuscated *mirai*, *bashlite*, obfuscated *bashlite*. Ransomware: *gonnacry* using Blowfish, *gonnacry* using AES, *gonnacry* using DES. Packed and without packing. rootkit: *maK_it* and *spy*. benign: random, video, music, picture, camera activities using random user environments.

We used traces recorded during the activity of 30 malware samples, plus traces of benign activities (random, video, music, picture, camera activities), both recorded in a random user environment to avoid biases. As explained in Section 3.2, the malware binaries are variations of five families: *gonnacry, spy, maK_It, mirai* and *bashlite*, including seven different obfuscation techniques. In this scenario, we aim to retrieve the type of malware (if any) infecting the device at the time of the recording. This gives us a 4-class classification problem: ransomware, rootkit, DDoS, and benign. All models are very efficient for this problem ($>$ 98% accuracy), and clearly obfuscation does not hinder type classification. We can observe that CNN (99.82%) is slightly more accurate than MLP, NB, and SVM. The confusion matrix is illustrated in Figure 3.6, which illustrates the predicted classes (predicted label) from the network per executed binary (true label). The darker the color, the higher the proportion of correctly predicted la-

bels. We can observe no confusion for the benign and rootkit class to any other class, and a minor confusion between DDos vs ransomware.

Family classification

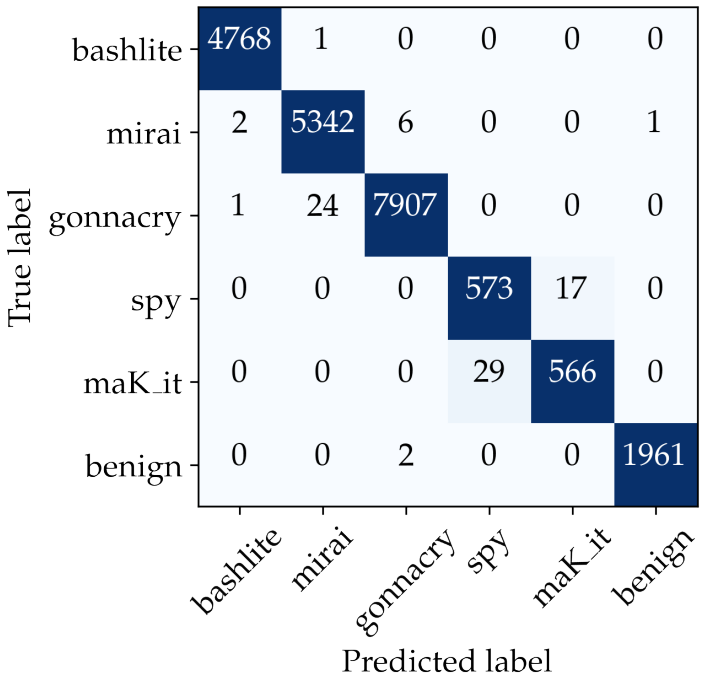


Figure 3.7 – CNN family classification (Accuracy 99.61%). Including *bashlite*: original & obfuscated. *mirai*: original & obfuscated. *Gonnacry*: *gonnacry* using blowfish, *gonnacry* using aes, *gonnacry* using des. Packed & without packing. *maK_it*: original rootkit. *spy*: original spy rootkit. benign: random, video, music, picture, camera activities using random user env.

In this scenario, we classify into the malware family plus benign class, which gives six different classes: *bashlite*, *mirai*, *gonnacry*, *spy*, *maK_it*, and benign. CNN gives the highest accuracy with 99.61%, but also MLP and ML provide results $> 97\%$. The confusion matrix is illustrated in Figure 3.7, which shows that all classes are mostly correctly classified with a small confusion on both sides between *spy* and *maK_it*. Again, we see that obfuscation does not hinder the classification.

Novelty classification

Previous experiments showed that it was possible to correctly classify known malware into its corresponding types and families. While this is certainly a first step, in real life malware analysis, it is very common to encounter unknown variations of a threat. To emulate this scenario, we split the dataset of malicious binaries according to the applied variations. Some of the variations of each of the five malware families (*gonnacry, bashlite, mirai, spy, and maK_It*) were not used in the training dataset and were reserved only for testing (detailed in Table A.1 in Appendix). In addition, we did not include in the test dataset any of the variations we used for training. We reserved the processed spectrograms representing the activities of 18 binaries for training purposes, and the spectrograms representing the activities of the remaining binaries for testing purposes. Also, *maK_It* was only present in the training, and *spy* only in the test dataset. As we can observe, even though the models are predicting unseen (obfuscated) variants, all models perform with an accuracy of $> 92\%$, with CNN at 99.38%. Accordingly, even *unseen* variations in the training phase do not hinder our methodology.

Virtualization and packer identification

In next two scenarios, we test if the binary is protected with virtualization or packing, which results to two (two-class) detection problems. For each of them, we used the traces of the original malware (*mirai*, *bashlite* and *gonnacry*) as well as the traces of the corresponding protected variation. We see that virtualization is slightly easier to detect than packing, with CNN performing with the highest accuracy of 95.83% and 94.96% respectively.

Obfuscation classification

In this scenario, we are interested in classifying dataset into the 7 obfuscation techniques: Opaque predicates, bogus control flow, control-flow flattening using O-LLVM or Tigress, instruction substitution, virtualization, or packing. Both of the network models were able to learn to differentiate obfuscation techniques independently of the five underlying malware families. CNN is more efficient achieving 82.70% (vs a random guess of 14.29%). Again, MLP was slightly worse and ML techniques show a gap of around 10%. The confusion matrix is illustrated in Figure 3.8, which shows that for each obfuscation technique CNN predicted the correct label (the darkest color/the

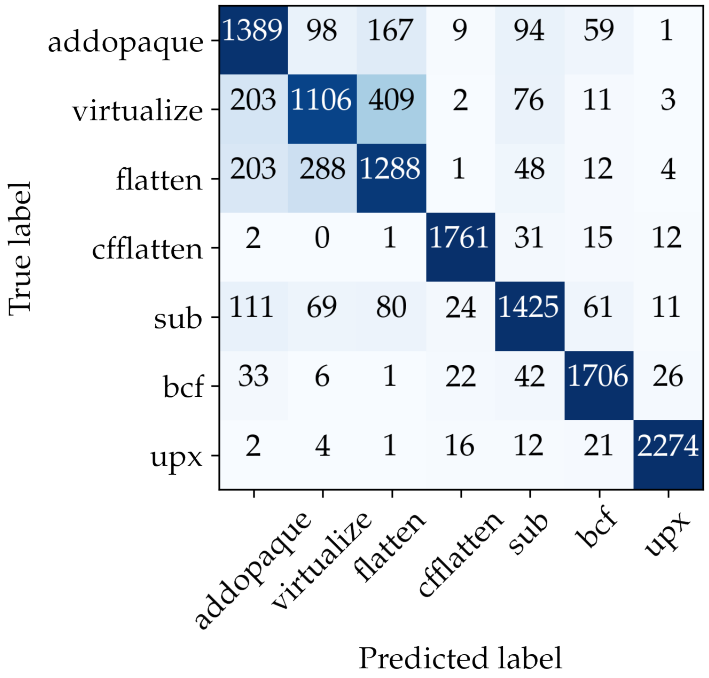


Figure 3.8 – CNN obfuscation classification (Accuracy 82.70%). Including *addopaque*: opaque predicates, *virtualize*: virtualization, *flatten*: control flow flattening using Tigress, *cfflatten*: control flow flattening using O-LLVM, *sub*: instruction substitution, *bcf*: bogus control flow, *upx*: UPX packing.

highest number on the diagonal). Some confusion can be observed between *addopaque*, *virtualize*, and *flatten*, which are executed using Tigress, and indeed they share similar options [9].

This result shows that our methodology is not only able to distinguish between malicious activities, but even focus solely on behavioral features independent of the underlying binary execution.

Executable classification

This scenario is a straightforward executable identification, where the model is trying to profile exactly the binary that generated the spectrogram. This translates into a classification problem of 35 classes (including distinct benign activities), identifying the family and variant with possible obfuscation of the malware. For the number of

9. http://tigress.cs.arizona.edu/transformPage/docs/flatten/index.html

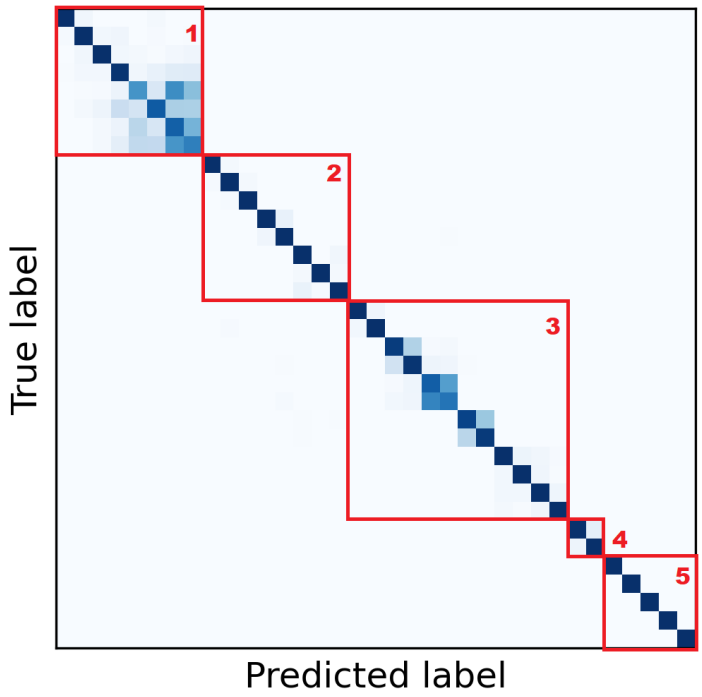


Figure 3.9 – Confusion matrix of a CNN classification into 35 binaries from left to right (with and without obfuscation).
(1) *bashlite_cfflatten, bashlite_upx, bashlite_bcf, bashlite, bashlite_addopaque, bashlite_sub, bashlite_flatten, bashlite_virtualize*;
(2) *mirai_sub, mirai_bcf, mirai_cfflatten, mirai, mirai_upx, mirai_addopaque, mirai_flatten, mirai_virtualize*;
(3) *gonnacry_des, gonnacry_des_upx, gonnacry, gonnacry_aes, gonnacry_aes_upx, gonnacry_upx, gonnacry_flatten, gonnacry_virtualize, gonnacry_addopaque, gonnacry_bcf, gonnacry_sub, gonnacry_cfflatten*;
(4) *spy, maK_It*;
(5) *benign:* encode video, play audio, take picture, record camera, random.

classes and the closeness of the underlying executions, all models get very good results, where CNN is the most effective, with a rate of 82.28% compared to a random guess of only 2.86%.

The confusion matrix of the CNN binary classification is given in Figure 3.9. As we can see, if confusions between classes happen, they happen between binaries that belong to the same family (squared in red in the figure). In addition, we observe that in most cases, the "darkest" color appears on the diagonal, which means that the highest score (output of the CNN) occurs for the true class label. So, in most cases, obfuscation

does not hinder exact binary profiling, the prediction of the exact binary is effective, and software obfuscation techniques do not hinder hardware analysis.

However, we still observe groups of binaries that are harder to distinguish and one misclassification. More precisely, one can observe that *bashlite_cfflatten*, *bashlite_upx*, *bashlite_bcf*, and *bashlite* have (nearly) no confusion with other binaries, which means that the obfuscation does not mask the behavior of the binary and the obfuscation technique itself is visible and distinguishable; *bashlite_addopaque* is misclassified as *bashlite_flatten* which is inline with our previous explanation of the similarities of the underlying techniques, and there is a confusion between *bashlite_flatten* and *bashlite_virtualize*.

For *mirai* and its variants we see a much smaller effect of the obfuscation techniques on the classification outcome than for *bashlite* and *gonnacry*. Meaning that the obfuscation technique is clearly identifiable and does not mask the behavior of mirai itself.

For gonnacry we have several distinct groups:

— *gonnacry-des-upx*, *gonnacry-des*: only a very minor confusion can be visible between the packed and unpacked version using des. Interestingly, there is no confusion using des with the version of aes and blowfish and their packed variants.

— *gonnacry*, *gonnacry-aes*: *gonnacry* and *gonnacry-aes* are slightly confused.

— *gonnacry-des* and *gonnacry-des-upx* are not confused with any other binary;

— *gonnacry* and *gonnacry-aes* have a slight confusion, which means that in some cases the encryption with blowfish and aes are not clearly distinguishable;

— this effect is even more present when the binaries are packed, i.e. for *gonnacry-upx* and *gonnacry-aes-upx*;

— again, similar to bashlite, we see a slight confusion between *gonnacry_flatten* and *gonnacry_virtualize*.

— *gonnacry_addopaque*, *gonnacry_bcf*, *gonnacry_sub*, *gonnacry_cfflatten*: we observe only nearly no confusion between these four obfuscation techniques.

We see nearly no confusion when predicting maK_it and spy. Finally, as before the benign binaries show no confusion with any other malicious binaries, and there is no confusion between each of the benign classes.

3.5.2 Discussion

Novel malware

The results we obtained show that our approach is successful in classifying various malware samples into their types, families, and exact binaries, as well as identifying and classifying obfuscation. The closeness between accuracy, recall, and precision of each experiment indicates robustness and no overfitting to any specific class. We are able to classify malware variations unseen during the training phase, which is particularly relevant in practical scenarios when considering the evolution of malware. Furthermore, we realized the measurements were done in a stable lab environment, but no exact triggering, nor any restrictions on the background processes of the target device had been made, which corresponds to a setup a malware analyst could exploit.

We computed robust models as shown by the closeness between the accuracy, the recall and the precision of each experiment. This means that our models do not overfit into any specific class. While it is out of the scope of the present work, it is worth mentioning that our approach is very effective for malware detection. As we can see in Figures 3.6 and 3.7, the precision and recall of the benign type are both equal to 1, which gives us a perfect f1-score of 1. All the results presented could certainly be improved, for example, by fine-tuning the hyper-parameters of the neural network models on a wider range of parameters, or by using more complex models, or more bandwidth from the spectrogram.

Obfuscation resiliency

In this work, we examined 7 obfuscation techniques, including packers and virtualization, which have been used by real-world malware as a growing trend to exploit cryptors and packers to hide the true intent of malware samples. While previous solutions such as signature-based packer detection can be evaded, our results show that we can distinguish between obfuscation techniques solely based on EM traces, which gives us the opportunity to analyze the evolution of IoT malware since new obfuscation techniques will be reformed to thwart detection.

ML algorithms

According to our findings, the accuracy of SVM and NB is comparable for more straightforward classification problems such as type and family, but the gap widens when considering more complex scenarios (e.g., executable, obfuscation). Therefore, in some contexts where the amount of training time and resources matters, these algorithms could be a reasonable choice in all scenarios.

Note that, all our results have been obtained by considering that the malware analyst measures only one trace per binary to predict the correct class. However, it could be possible that he has the resources to measure multiple times the same binary execution and to reduce noise, computes the mean over each of these execution traces. Results using this approach are given in Figure A.1 in the Appendix for SVM and NB, which shows a drastic improvement in many scenarios. For example, NB could reach $> 80\%$ for binary classification and 100% in type and novelty classification. Interestingly, we could not observe an improvement for MLP and CNN, which is discussed in more detail in the Appendix.

Furthermore, we will extend from classification models to detection models that combine with mean computing over multiple execution traces. This dramatically improves the findings, which will be described in detail in Chapter 4.

Malware evasion

From our results, one can observe that malware SCA evasion (*i.e.* prevention from our methodology) is not straightforward. In particular, we derived that our system is robust against various code transformations and obfuscations, including random junk insertion, packing, and virtualization, even when the transformation was previously unknown to the system. Another approach to evasion, instead of obfuscating malicious behavior, could be to hide exploitable information by lowering the signal-to-noise ratio. This could, for example, be achieved by forcing highly parallel/multi-core executions such as ConcSpectre malware [LXF+22], which remotely disturbs thread scheduling. However, as our methodology relies on EM emanation, which can be observed on a local and global scale, and on frequency transformation with filtering, it is unclear if hiding exploitable information is easily achievable at all. Even more, unexpected highly parallel/multi-core activities can be more easily detected as abnormal behavior, which is not in the interest of malware designers. We therefore see the topic

of malware evasion against physical side-channel information as a new open direction with non-straightforward solutions.

3.6 Conclusion and perspectives

We have demonstrated in this chapter that by using simple neural network models, it is possible to gain considerable information about the state of a monitored device, by observing solely its EM emanations. We were indeed able to not only detect, but also determine the type of real-world malware infecting a Raspberry Pi running a full Linux OS, with an accuracy of 99.89% on a test dataset including 20 000 traces from 30 different malware samples (and five different benign activities). We demonstrated that software obfuscation techniques do not hinder our classification approach, even if the obfuscation technique was not known to the analyst before. Even more, we showed that it is possible to detect a particular obfuscation and classify between them (or groups of obfuscation techniques), and classify the family with its exact variant/obfuscation labels.

Given our experimental results, malware analysts therefore profit from our robust methodology to gain a better understanding of the variant, type/family, forensic, and/or evolution of malware groups and campaigns, particularly in the context when software systems fail (due to malware evasion) or cannot be applied (due to restricted resources or update processes on the embedded device). While these results were obtained in a controlled setup, as it is usually the case with malware analysis,

Future work may concentrate on the extension of our approach to real-time user systems (*e.g.* Section 4). Another interesting direction could be the investigation of other architectures and devices, to assess in which measure the knowledge learned by a model on one device can be transferred to another one.

While the experiment results are certainly compelling, it is important to mention that all experiments took place under controlled and ideal settings. At no point in this chapter is it claimed that this technique, at least in its current form, could be used to detect whether a computer or IoT device has been infected with malware but only IoT malware classification. This problem will be further discussed in the next chapter. Furthermore, data from a far broader range of computing devices would need to be collected before analysts could evaluate whether this framework had any practical value outside the lab. There are different factors to be considered that any of the EM

signatures recorded on the Raspberry Pi test equipment would be relevant to a random Wi-Fi router off the shelf. Nonetheless, our work can be considered as a first step towards (detailed) behavioral analysis through electromagnetic emanation, opening a new research direction for future work.

This page intentionally left blank

4

ULTRA:Ultimate Rootkit Detection over the Air

The content of this chapter, which is based on a joint work with Damien Marion, and Annelie Heuser, resulted in one paper that was published in the 25th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID) 2022 [PMH22]

Contents

4.1 Introduction

In this chapter, we present the ULTRA framework, which can identify rootkits in real-world scenarios while being contactless, low-cost, and with no resource requirements on the specific embedded device, making it particularly suitable in the context of IoT. Our approach consists of using hardware and software baits while only measuring Electromagnetic Emanations (EM) over the air with a software-defined radio device. This setup prevents known malware evasion techniques as there is often no identifiable activity or resource on the monitored device. Further, baits enable us to detect minor behavioral changes in the system from stealthy rootkits, rather than rely on the signature of one-time active infection process. In our experiments, we move beyond detection to classification, tackling practical scenarios of undiscovered rootkit (variants) during the training phase, probe (dis)location, and evaluation of noisy environments. We show that our methodology is compliant to real-world analysis scenarios and can be labeled as wave (a probe) and play (WnP). For this, we particularly investigate the influence of added kernel activity, classification scenarios, and the influence of undiscovered rootkits variants during the training phase. Our evaluation is carried out on two devices with two architectures: Creator CI20 which uses MIPS and Raspberry Pi relying on ARM, demonstrating that ULTRA is not limited to a single technology. We compare our detection results to three software rootkit detection tools, where ULTRA outperforms them in terms of requirements, detection level, and latency.

4.1.1 Motivation

By 2025, we are expecting to have over 64 billion IoT devices [RHK20] and more will be produced as beyond 5G technologies mature. Simultaneously to the advances in IoT and embedded devices, the number and variety of cyberattacks have grown in recent years, making current security approaches outdated in a short time [AG18, RSSHCB21]. Many IoT manufacturers use Linux-based operating systems, making it easier to migrate rootkits to target embedded devices. Generic malware detection solutions rely on static or dynamic analysis that still have various shortcomings. In particular, major problems are related to the diversity of IoT architectures [CGFB18], obfuscation techniques (discussed in Chapter 3), and the fact that many IoT devices may have constrained resources, limited battery or accessibility.

Rootkits, those nefarious pieces of software that conceal deep within a system in order to grant hackers access, are one of the most challenging malware to defend against over the years. A recent study [pts21] shows that 44% of cybercriminal cases used rootkits to attack government agencies, while 56% of the investigated rootkits were used in advanced persistent threat (APT) attacks. They are typically utilized by highly skilled actors with extensive malware creation skills and financial resources to develop or acquire rootkits. One of the most sophisticated APT style attacks that employed a rootkit was Stuxnet, which targeted industrial control systems and included the first ever programmable logic controller (PLC) rootkit and a Windows rootkit to hide its malicious files as well as injected code into PLC [FMC11]. Recently, NSA and FBI [AoI21] reported Drovorub rootkit developed by state-sponsored APT28 to infect Linux systems to hide itself and files, directories, and network activities.

Even though rootkits can control the highest protection levels on the device's system, they do not have control over outside hardware-level events such as EM emanation. A protection system relying on hardware features cannot be taken down, even if the malware owns the maximum privileges on the machine. Therefore, with EM emanation, it becomes possible to detect stealthy malware such as kernel-level rootkits, which are able to avoid software-based analysis methods.

In Chapter 3, while the experiment results from the IoT malware classification are certainly encouraging, they took place under lab-settings without exact probes tuning and structured product-oriented. In this chapter, we demonstrate novel solutions to improve and extend it to a low-cost and portable detection solution.

4.1.2 Our contributions

In summary, our primary contributions are as follows:

1. **SDR-based practical rootkit detection solution.** Our approach is the first and only to detect rootkits in real-time solely by EM, using a low-cost contactless SDR device, with no triggering or resynchronization of side-channel measurements required.
2. **A novel methodology for detecting stealthy rootkits on IoT devices.** We present 2 forms of *baits* to trigger the behavior of stealthy rootkits. ULTRA detects the presence of modifications from the rootkit to the system, which is significantly more subtle than traditional rootkit infection detection.

3. **Realistic data collection with obfuscated variants on embedded devices.** Using 7 distinct rootkit families and 2 obfuscated variations, we collected an unbiased collection of 800 000 raw traces, including benign noise from both kernel and user space activity. Traces are collected from two separate IoT devices with different architectures: MIPS and ARM.
4. **Proposed scenarios compliant to rootkit detection in real-world settings, and ready for implementation.** We put together various scenarios, each reflecting a real-world rootkit detection and classification use case: rootkit novelty detection, obfuscated rootkit detection, keylogger novelty detection, evaluation of benign activities, noise, probe dislocation, or type invariance. These scenarios go way beyond (simple) detection scenarios considered in the state-of-the-art.
5. **Open-source.** The source code for the ULTRA framework, measurement datasets, rootkit detection and classification models, results of our experiments, and demo videos are all publicly available at `https://gitlab.com/ultra-RK/ultra/`.

4.1.3 Roadmap

The structure of this chapter is organized as follows. In Section 4.2 we describe the main part of our work which is the Ultimate Rootkit Detection over the Air (ULTRA) framework. In Section 4.3 we show how we conduct experiments for data acquisition and data processing. Section 4.4 discusses the results and concludes the chapter in 4.5.

4.2 ULTRA: Ultimate Rootkit Detection over the Air framework

In this section, we propose “ULTRA: ULTimate Rootkit classification and detection over the Air" that is able to analyze stealthy (non-active) rootkits by baiting and waving an EM probe over the device. The framework takes an EM trace monitored from a target device as an input and predicts the presence of a rootkit and reveals its labels. Figure 4.1 illustrates ULTRA’s workflow, which will be detailed within the following subsections. First, we define our threat model, definitions, and methodology, and then explain how the EM dataset is collected while a bait is executed on the target device with or without the presence of a rootkit. Thereafter, as the collected data is very large

and noisy, a preprocessing step is required to separate relevant informative signals. Finally, using this result, we train neural network models and machine learning algorithms for detection and classification in a variety of practical scenarios.

4.2.1 Threat model and methodology

Placing rootkit detection mechanism on the same level as the rootkit itself makes it apparent from both sides, thus detection can be evaded by advanced rootkits, *e.g.* any inspection at the kernel level can be subverted by kernel-level rootkits. In this section, we propose a framework that is solely placed outside of the target device, thus providing the least possibility of being evaded by rootkits. We analyze the prerequisites for developing such a dynamic rootkit detection system (i.e., one that cannot be detected and evaded by the rootkit). These requirements serve as guiding principles for the development and implementation of ULTRA. We start with a brief threat model for rootkits, present notations and definitions, before providing an overview of the ULTRA framework. We then extend our model to consider datasets that are part of realistic execution environments and baits to trigger rootkit.

Threat model

Stealthy rootkit

We focused our attention on rootkits, which have the highest privilege of "*root*" within the Linux system. Since they have unrestricted access to any protection rings ranging from user space to hypervisor level, we assume that any behavior from the target device is untrustworthy. Unlike "normal" malware, *stealthy* rootkits are not constantly active, but only operate during certain activities. We assume that the rootkit is able to avoid any traditional malware analysis technique such as signatures, VM inspection, hooking, etc.

To avoid being evaded by advanced rootkits, the proposed framework must not use any modifications similar to traditional malware analysis techniques such as signatures, VM inspection, hooking, etc.

Target environment

The target environment requires stability and high availability, thus interruption, downtime, or device reconfiguration should be kept to the least possible. This is a crucial requirement for military control systems, unmanned vehicles, autonomous vessels,

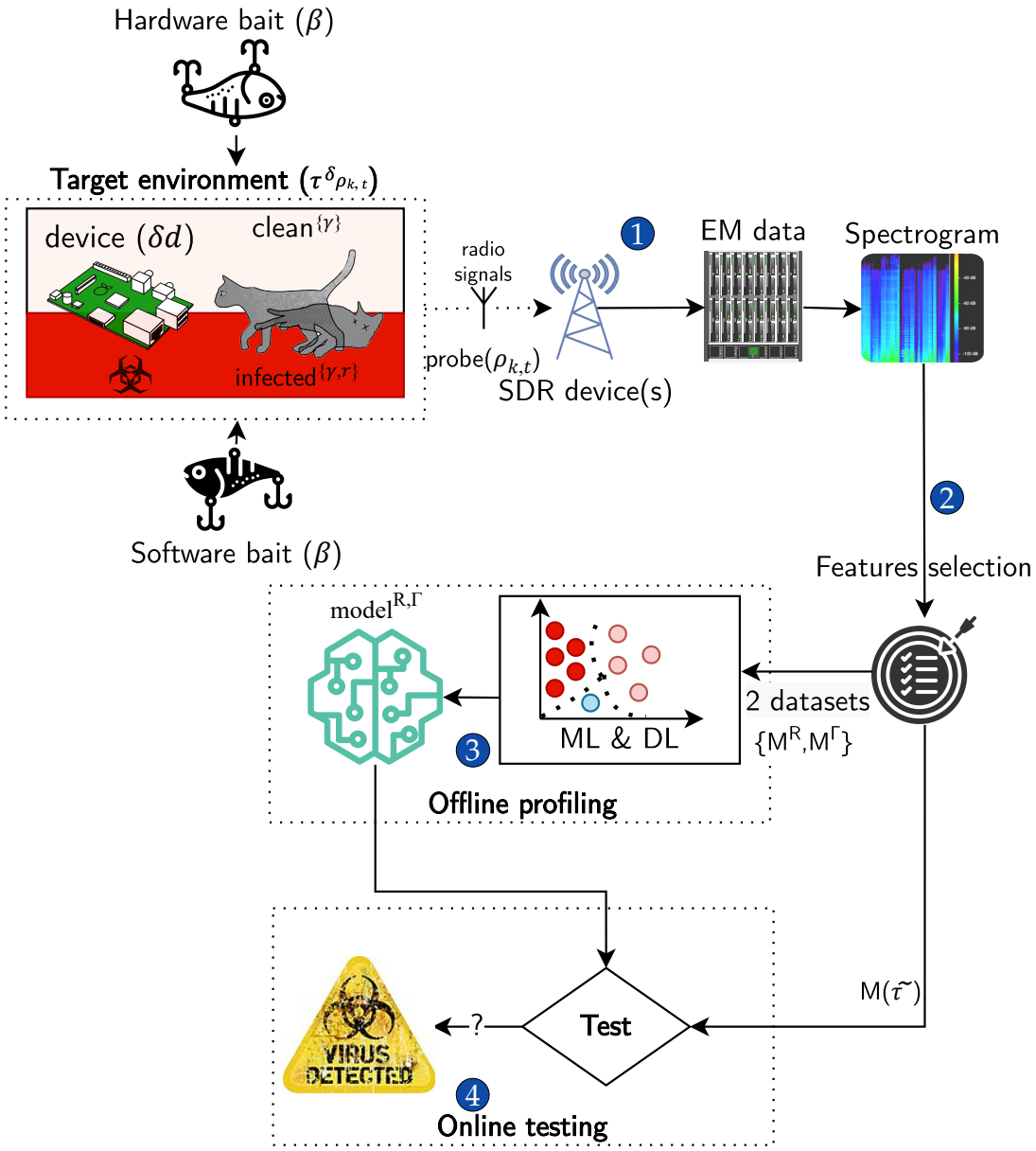


Figure 4.1 – Illustration of ULTRA framework. 1 Trace acquisition: from black box target devices receiving software and hardware baits to EM measurements; 2 Pre-processing: converting raw measurements into usable data; Malware detection and classification: from offline profiling models (trained on processed data 3) through online rootkit label prediction (testing 4).

etc. Nowadays, industrial systems have numerous active embedded devices and they are difficult to manage, interrupt and customize their analysis environment, so that we

suppose no acknowledge of any malicious service presented on the target devices.

Notations and definitions

Definition 3: Device

A device δ_d is defined as a computing unit with technology d running benign activity γ. The device can be either in a clean state $\delta^{\{\gamma\}}$ or infected $\delta^{\{\gamma,r\}}$ with a rootkit r.

For simplification, we omit indices when they are not relevant. For example, we only use δ when the infection and activity status of the device is unknown or not relevant, and explicitly use $\delta^{\{\gamma,r\}}$ or $\delta^{\{\gamma\}}$ to refer to a clean or infected device respectively.

Definition 4: Environment

An *environment* is defined by its device and probe. In particular, an environment $\tau^{\delta}_{\rho_{k,t}}$ consists of a device δ and a probe $\rho_{k,t}$ with type t and location k.

Our approach consists in using baits on a device to trigger and thus be able to profile and detect rootkit behavior. Baits are defined in the following, where we give a detailed description with examples in Subsection 4.2.3.

Definition 5: Bait

A *bait* β is a software or hardware stimulus on a device δ with the following requirements:

1. The bait can trigger partial or full behavior of rootkits without knowing *modus operandi* of the rootkit in advance
2. It supports a variable duration of execution activities that can be remotely controlled
3. It cannot be distinguished from common benign behavior (e.g. it relies on unprivileged execution)

To represent that a bait β is executed in an environment $\tau^{\delta}_{\rho_{k,t}}$ we write $\tau^{\delta}_{\rho_{k,t}}(\beta)$. Now our approach consists in measuring EM traces of an environment that is triggered with a bait: $\tau^{\delta}_{\rho_{k,t}}(\beta) \rightsquigarrow \tilde{\tau}^{\delta}_{\rho_{k,t}}(\beta)$. In a practical scenario, we observe q EM measurement traces denoted as $\tilde{\tau}^{\delta}_{\rho_{k,t}}(\beta)^q = \{\tilde{\tau}^{\delta}_{\rho_{k,t}}(\beta)^1, \ldots, \tilde{\tau}^{\delta}_{\rho_{k,t}}(\beta)^q\}$. Note that the benign activity γ on the

device δ is randomly chosen for each measurement $(1, \ldots, q)$. For instance, let the benign activity set be denoted as $\Gamma = \{\Gamma_0, \Gamma_1, \Gamma_2, \ldots, \Gamma_n\}$, where in our experiments Γ_0 is defined as no activity and Γ_i for $0 < i \leq n$ represents (noisy) benign activity, then for a device without infection $\tilde{\tau}^{\delta\{\vec{\gamma}\}}_{\rho_{k,t}}(\beta)^q = \{\tilde{\tau}^{\delta\{\gamma_1\}}_{\rho_{k,t}}(\beta)^1, \ldots, \tilde{\tau}^{\delta\{\gamma_q\}}_{\rho_{k,t}}(\beta)^q\}$, where $\vec{\gamma} = \{\gamma_1, \ldots, \gamma_q\}$ is a vector of randomly chosen activities from Γ. The notation for an infected device $\delta^{\{r,\gamma\}}$ follows straightforwardly.

Let the set of $p > 0$ possible baits be denoted as $\mathcal{B} = \{\beta_1, \beta_2, \ldots, \beta_p\}$. For any bait $\beta_i \in \mathcal{B}$, $1 \leq i \leq p$, we capture q observations, which results into a matrix $\mathcal{M}$ of size $p \times q$:

$$\mathcal{M} = \mathcal{M}(\tilde{\tau}^{\delta}_{\rho_{k,t}}(\mathcal{B})^q) = \begin{bmatrix} \tilde{\tau}^{\delta}_{\rho_{k,t}}(\beta_1)^1 & \ldots & \tilde{\tau}^{\delta}_{\rho_{k,t}}(\beta_p)^1 \\ \vdots & \ddots & \vdots \\ \tilde{\tau}^{\delta}_{\rho_{k,t}}(\beta_1)^q & \ldots & \tilde{\tau}^{\delta}_{\rho_{k,t}}(\beta_p)^q \end{bmatrix}.$$

ULTRA Framework classification and detection methodology

The framework consists of two phases, offline profiling and online testing.

Offline device profiling

In the offline phase an analyst profiles an environment that contains a device δ that is infected with a rootkit r (i.e. $\delta^{\{\gamma,r\}}$) and that is in a benign state (i.e. $\delta^{\{\gamma\}}$) both running benign activity γ. Let the rootkit set be denoted as $\mathcal{R} = \{r_1, r_2, \ldots, r_m\}$, and the benign set as $\Gamma = \{\Gamma_0, \Gamma_1, \Gamma_2, \ldots, \Gamma_n\}$. We measure two sets of matrices: traces of baits running on an infected system:

$$\mathcal{M}^{\mathcal{R}} = \{\mathcal{M}(\tilde{\tau}^{\delta\{r_1,\vec{\gamma}_1\}}_{\rho_{k,t}}(\mathcal{B})^q), \mathcal{M}(\tilde{\tau}^{\delta\{r_2,\vec{\gamma}_2\}}_{\rho_{k,t}}(\mathcal{B})^q), \ldots, \mathcal{M}(\tilde{\tau}^{\delta\{r_m,\vec{\gamma}_m\}}_{\rho_{k,t}}(\mathcal{B})^q)\},$$

and against clean system:

$$\mathcal{M}^{\Gamma} = \mathcal{M}(\tilde{\tau}^{\delta\{\vec{\gamma}_1\}}_{\rho_{k,t}}(\mathcal{B})^q)$$

Both states of the system are initialized under benign activities $\vec{\gamma}_i, 0 < i \leq m$, that are randomly chosen from Γ and independent of the rootkit r_i.

Using these two datasets, our approach consists in building feature extraction methods to reduce complexity and build machine and deep learning models that are able to detect if a device is infected with a rootkit or not, i.e.

$$\{\mathcal{M}^{\mathcal{R}}, \mathcal{M}^{\Gamma}\} \Rrightarrow_{training} \texttt{model}^{\mathcal{R},\Gamma}.$$

Online testing

In the online phase, the goal is to determine if a device in an environment is infected with a rootkit or in a benign state, where no information on γ is known. For this, we use the estimated model $\texttt{model}^{\mathcal{R},\Gamma}$ that has been built in an environment $\tau^{\delta}_{\rho_{k,t}}(\mathcal{B})$ in the offline profiling phase, and the corresponding machine/deep learning prediction algorithm to output either benign vs infected state, or classify into particular categories.

In a practical context, the prediction algorithm detects or classifies the status of the device using 1 measurement only in the testing phase. Naturally, in our experiments to perform statistical evaluation, we collect a significant number of traces to estimate accuracies and false positives and negatives. So, p' measurement traces $\mathcal{M}(\tilde{\tau}^{\delta}_{\rho_{k,t}}(\mathcal{B})^{p'})$ are measured by placing a probe $\rho_{k,t}$ over the target device δ while unknown activity is running. Note that, in the testing phase, the device is a blackbox since no information of activity is acknowledged (*i.e.* presence of rootkits r or device activities γ).

In our experimental part (see Section 4.4.2), we setup a variety of experimental studies to test the effectiveness and robustness of our models. We start by keeping $\rho_{k,t}$ and δ unchanged from the offline profiling phase, and consider two types of devices: δ_{rasp} for Raspberry Pi and δ_{CI20} for Creator CI20 which are further detailed in the next subsections. Next, we consider the dislocation and changing the type of the EM probe(s). In particular, we acquire traces with the same probe from two distinct locations: $\rho_{k=0,t=0}$ and $\rho_{k=1,t=0}$ and include a cheaper handcrafted EM probe $\rho_{k=2,t=1}$. Furthermore, we investigate novelty detection where the rootkit set $\mathcal{R}$ varies between the learning and testing phases. As well as a scenario where the noise level of the benign activity Γ between learning and testing changes.

4.2.2 Dataset

The collection process of datasets is a critical component in the development and evaluation of malware detection tools. At the time of writing, ULTRA framework supports but is not limited to rootkits of 32-bit ELF MIPS and ARM architectures, which have been validated on Ci20 and Raspberry Pi devices. Following, we will discuss the collection of rootkit and benign datasets.

Rootkit dataset

Even though rootkit malware samples are very rare in the wild and it is difficult to get a large enough sample size for Linux, we tried to be realistic in this study by acquiring 9 rootkit variants, from 7 up-to-date open source rootkit families listed in Table 4.3. The rootkit dataset, including popular rootkit strategies used by today's malware writers, covered various features of common Linux rootkits such as: self hiding, file, module, process, network port, socket hiding; keylogger; remote access backdoor and root privilege escalation (LPE). We therefore took under consideration both, user-level (*beurk* [UT17], *vlany* [mem19]), and kernel-level rootkits (*diamorphine* [m0n21], *m0ham3d* [m0h15], *adore-ng* [Han20], *spy* [Jan21], *maK_it* [McN15]).

Furthermore, since malware developers often use obfuscation techniques to bypass malware detection, we apply static code and string rewriting techniques to 2 rootkits: *m0ham3d* and *diamorphine* to test the robustness of our methodology, and to investigate obfuscation mechanisms against side-channel monitoring. Technical details of the applied obfuscation can be found in Appendix A.2. As a consequence, two new obfuscated variants are included in the dataset that can easily evade signature-base detection solution (see Table 4.11). The total dataset of 9 rootkits was compiled on both architectures, thus using 2 different Linux kernel headers. By code reviewing source code of collected rootkits, we draw an overview of their functionalities and features that will be described as follows:

- *spy* is a LKM rootkit which has functionalities to hook and record keys pressed in the keyboard events to debugfs by exploiting *register_keyboard_notifier*.
- *MaK_It* shares the same rootkit ability to *spy* as a keylogger by exploiting *register_keyboard_notifier*, with addition of kernel module self-hiding, local privilege escalation and reverse-shell backdoor.
- *adore-ng* is a LKM rootkit which has functionalities of self-hiding, process, network ports and files hiding, with a well-developed userspace toolkit *ava* for covert communication between kernel and user land. The rootkit patches the Linux Virtual File System (VFS) which is is a software layer in the Linux kernel that handles all accesses related to the standard Unix file system, rather than the common system call table hooking techniques, to serve several purposes: hide files, devices and network ports, anti logging, as well as local privilege escalation.
- *m0hamed* is a LKM rootkit that is based on syscall table hooking for *open*, *read* and

write system calls. The features of this rootkit are local privilege escalation, and file handler in */proc* to permits the attacker to hide specified ports.

- *Diamorphine* is a LKM rootkit which has functionalities of self-hiding, hide/unhide any process, local privilege escalation, hiding specific files and directories by hooking syscalls such as *getdents*, *getdents64*, *kill*.
- *Beurk* is a user-space rootkit based on *LD_PRELOAD* hooking technique which allows specified libraries to be loaded before other libraries (*e.g.* libc, so that by placing the *Beurk* library in *LD_PRELOAD*, it can hook libc functions). Beurk supports files and directories hiding, wipe logs in realtime, anti process (especially anti-rootkits) and login detection, bypassing: *lsof*, *ps*, *ldd*, *netstat* analysis, and a remote pseudoterminal backdoor. In total, *Beurk* hooks 19 libc functions.
- *vlany* is a user-space rootkit also based on *ld.so* patching technique and armed with evasion techniques such as anti-debug via *ptrace* hooking, anti-forensics that temporarily unload itself. This rootkit supports a vast amount of features: process, files and ports hiding, real-time log wiping, Linux containers hiding, persistent re-infection, dynamic linker modification (vlany randomizes the *LD_PRELOAD* at a random path), reverse-shell backdoor as well as PAM backdoor.

Benign dataset

To train and evaluate the models, we collected a large dataset from a fresh installation of the Linux system. It is critical to select an unbiased dataset to avoid errors in later binary detection models and to assure the quality of the framework during real-time testing. Different execution modes are considered, such as default firmware actions, random computations, hibernation, or stress activities on target devices. There is one kind of software which shares similarities with rootkits: kernel drivers, which are installed at kernel-level and execute "good" behaviors. Compared to Chapter 3 which only considered user and system processes, we further extend the benign dataset by collecting both Linux user space binaries and kernel space modules from MIPS and ARM architectures. Thus, we generated a vast number of benign activities such as computations, background processes with malware-free, or usual activities on embedded IoT devices (listed in Table 4.1). This collection varies from high to low CPU resource consumption, from very long to short duration of activities, and likely confuses detection models as being classified as false positives. It is worth mentioning that only one

Table 4.1 – Benign dataset(Γ): Linux executables and kernel modules

Activities	**Executables and kernel modules ($\Gamma_{i>0}$)**			
Linux Utilities	mknod	vdir	more	find
	zgrep	ls	cat	findmnt
	zmore	as	ed	rm
	touch	dmesg	sleep	cd
	less	grep	objdump	time
Network	wget	hostname	ss	ip
Compression	gunzip	bunzip2	bzip2	tar
	uncompress			
Data backup	dd			
Scripting	python			
Linux drivers	rpcsec_gss_kı	lru_cache	bluetooth	atm
Firewalls	x_tables	ip_vs	br_netfilter	
Filesystem	9pnet	9p	ecryptfs	nfsd
protocol	btrfs	udf	xfs	cifs
modules	overlay			

of the previous rootkit studies on Table 2.4 considered the impact of benign activities.

4.2.3 Baits to trigger rootkit hooks

In contrast to generic malware, rootkits are deeply concealed in the system, with no indication of active activity. They are passive and only activated when a specific “backdoor" behavior is triggered *e.g.* making a specific system call, interacting with a covert channel, using special file names, etc. Therefore, revealing rootkit behaviors on a targeted device requires the use of triggers or unique tactics. In this subsection, we detail our approach to uncovering rootkits using baits that satisfy Def. 4.2.1.

The bait execution can trigger the behavior of specified benign activity (*i.e.* system calls, keystroke input, etc.) regardless of knowing the exact rootkit family it is dealing

with, therefore any deviation occurring between bait executions inside a clean and an infected state will indicate the presence of the rootkit on the target device. Algorithm 1 illustrates the proposed bait mechanism.

Algorithm 1 Algorithm of bait β_i

Require: $c \geq 1$

$\beta(i, args, c)$: ▷ i: index, $args$: arguments, c: iterations
$C \leftarrow c$
while $C > 0$ **do**
  $\beta_i(args)$
  $C \leftarrow C - 1$
end while

We define 2 novel strategies to trigger rootkits: software and hardware baits as follows.

Software baits

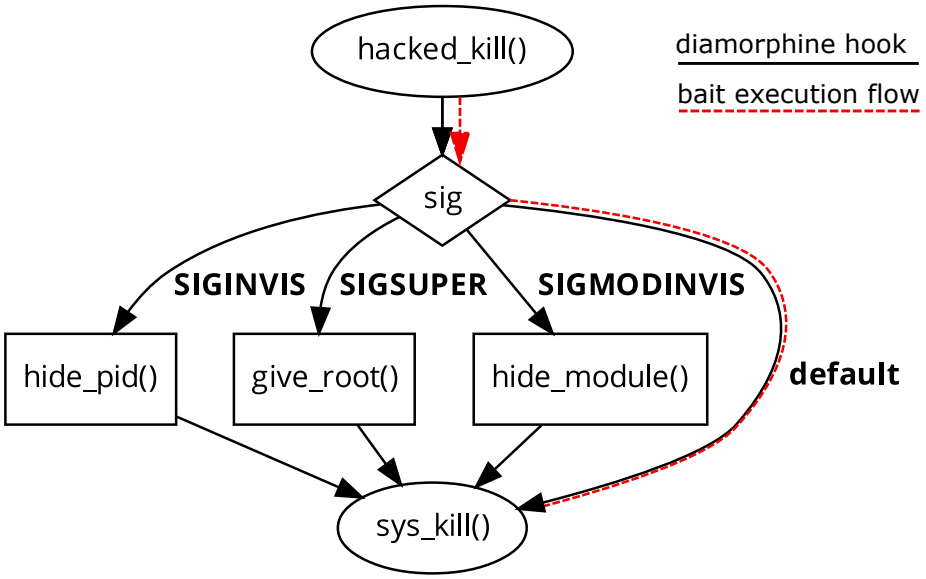


Figure 4.2 – Illustration of execution flow for *kill* bait running under *diamorphine* which infected Linux kernel with hooked system call *kill()*.

On software rootkit detection, Numchecker [WK13] used test programs which performed 5 system calls such as *sys_open*, *sys_close*, *sys_read*, *sys_getdents64*, and *sys_stat64* into the guest VM to determine the presence of rootkit based on deviations of HPCs. In our work, we carefully crafted software baits by reverse-engineering and code-reviewing a number of rootkits described in Section 4.2.2 to understand the common system calls were being used by rootkits. In terms of keylogger detection, a similar tactic on the software level has been conducted by Ortolani *et. al.* [OGC10] which simulates carefully crafted keystroke sequences, *i.e.* the bait, as input and observes the behavior of the keylogger in output to identify keyloggers among all the running processes. In our work, we conduct a representative set of 10 designed baits that can trigger 7 different rootkit families. The set takes into account not only syscall triggers but also initiated network activities and keyboard inputs (Table 4.3).

For example, *diamorphone* rootkit intercepts *kill()* to redirect syscall convention to *hacked_kill()*, which serves as a switch for 3 specific signal inputs (in Fig. 4.2). If the signal matches, it will turn the call to either process hiding, module hiding, or root escalation. In general, the designed bait does not acknowledge any rootkit *modus operandi* in advance. It simply calls the system call *kill()* with valid arguments, thus the switch will route its execution via the default branch of the original system call *kill()*. Note that, if the bait had routed into one of the 3 hjacked branches, the captured activity would have fully exposed malicious behaviors. However, routing to the default switch branch (illustrated as the bait execution flow in Fig. 4.2) yet created a small deviation (only 15 additional ARM instructions during our experimental setting) between the clean device and the infected device *i.e.* partially triggers behaviors of *diamorphine*. It is not yet straightforward that the observed EM signals can lead to a result of whether *diamorphine* is present on the device. It is important to point out that this minor deviation detection is significantly more subtle and accurate than typical rootkit detection at the infection phase.

Hardware baits

One could argue that software baits expose unprivileged execution at the OS user-level, which can be observed and evaded by the rootkit installed on lower levels of the system. We present the following hardware bait targeting rootkits: an external device that can be physically connected to the target device. The hardware prototype (Fig.4.3 in Appendix) is composed of a BluePill STM32 board connected to the target

device via USB, and can be controlled remotely via the SWD debugger protocol using an ST-Link v2 controller. It acts as a bare-metal hardware keyboard emulator, sending a sequence of output keystrokes to the device. In the case of keylogger rootkit detection, it meets 3 requirements from Def. 4.2.1: remote controllability, fully triggering keylogger behaviors, and no difference from a standard keyboard from the software-layer perspective. Furthermore, there is only inter-kernel communication between USB human interface device (HID) events and the HID layer from the descriptor after each emulated keystroke, with no interference from other user processes. As a result, this hardware keyboard emulator is an optimal solution for keylogger detection and anti-evasion.

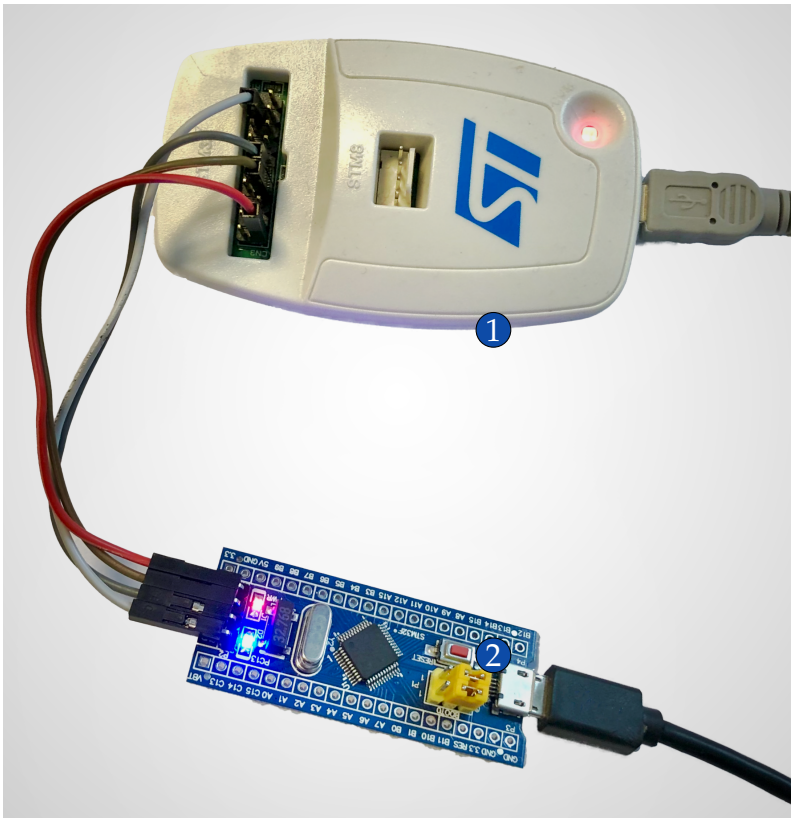


Figure 4.3 – Hardware keyboard emulator bait consists of one Blue Pill STM32 board (1) connected to a ST-link v2 (2) which under control of ULTRA host agent.

4.3 Practical use case of ULTRA

Table 4.2 – ULTRA's targeted devices specification, architectures (Arch.), and their targeted frequency leakage (Fc) and CPU in MHz.

Device δ	Arch.	CPU	RAM	OS	Fc
Raspberry Pi B+	ARM32	700	512MB	Linux 4.1.7	1222
Creator CI20	MIPS32	1200	1GB	Linux 3.18.3	792

4.3.1 Target devices

[CVD+20] showed the compilation information from a dataset of 93 000 malware samples collected over a period of 3.5 years on VirusTotal. Two-thirds of all samples are represented for by the two architectures, MIPS and ARM 32-bit. This can be explained by the widespread use of these CPU architectures in popular consumer embedded devices that are frequently exploited by IoT malware.

Our work focuses on rootkit detection on IoT devices, thus we have considered two widely embedded architectures for all experiments as specified in Table 4.2. They support broad activities for embedded and IoT scenarios including prototypes and developments, regarding their prominent architectures, size, power consumption and cost effective. Chapter 3 and previous works [WZH+18, FI17] show that cryptographic and anomaly activities can be distinguished by leveraging EM signals from the Raspberry Pi. However it is worth to mention that no study has been investigated the possibility of side-channel leakage on MIPS Creator CI20, so that there is no influence of prior knowledge on our experiment design. Thus, this work focuses on versatile rootkit detection challenges validating on different combinations of hardware and software rather than a narrowed solution to a specific device or architecture. We are not limiting the capabilities of the infrastructure with restricted bare-metal firmware, since both target devices are deployed with fully-functioning Linux on MIPS and ARM. Therefore any IoT applications can be performed together with internal noise such as background processes, services and interrupts.

To prevent the detection of any artificial artifacts from evasion techniques, and to maintain a realistic environment, all default background services are kept to their default (in practice, there are more than 100 running processes and services). Additionally, no adjustments, overclocking, or tuning of the processor clock rate are applied to the processor of target devices. By leveraging the combination of bare-metal analysis

and EM through SDR, it takes advantages that avoid the necessity of software tools such as sandbox, hooking and anti-evasion techniques. The proposed framework is an ideal candidate for target devices that require to be always-on and steady, where the installation of new protection software such as kernel integrity, firmware upgrade, or virtualization is not trivial.

4.3.2 Data aquisition

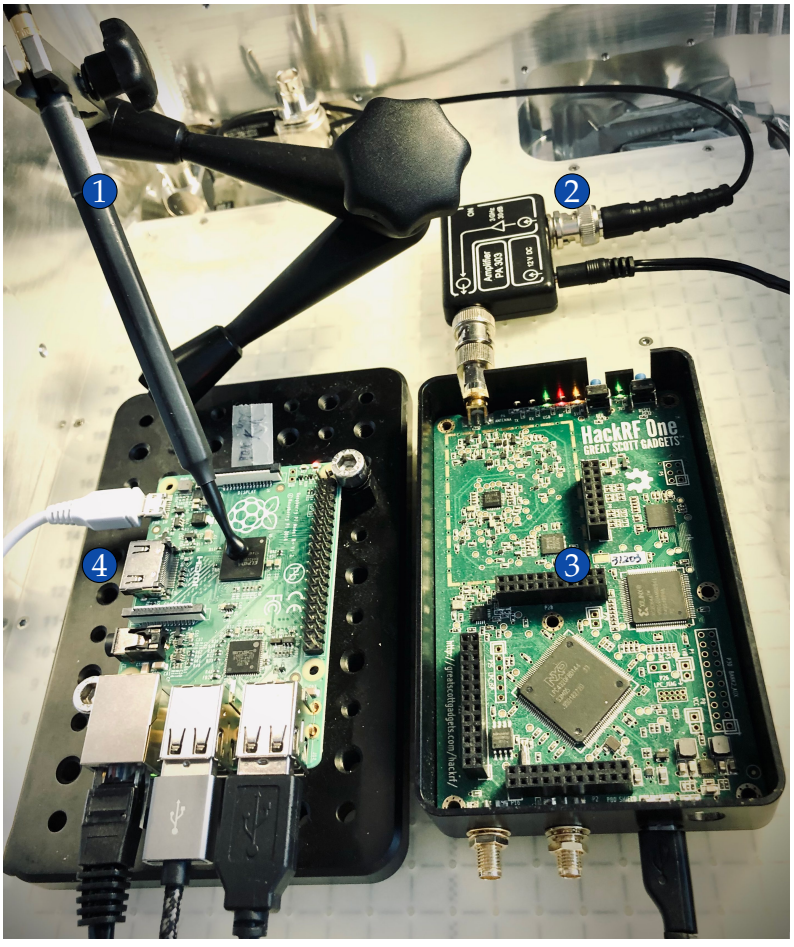


Figure 4.4 – ULTRA framework data acquisition consists of a H-Field probe (1), which connected to an amplifier (2) and HackRF (3), placed 1mm and 45 degree above the target device Raspberry Pi (4) processor.

The target devices are monitored in different setups ranging from low-cost to medium-cost (see bill of materials in Appendix table A.3). It consists of a HackRF SDR device

with a frequency range of 1MHz-6GHz, connected to an H-Field probe Langer RF-U 5-2, where the EM signal is amplified using a Langer PA-303 +30dB (Figure 4.4). A low-cost setup with an EM-compatibility probe, which is made of ferrite and conductor and placed 10mm farther from the processor (Figure 4.7) rather than 1mm, will later be discussed. In all acquisitions, the probes were placed contactless over the processors of target devices with a sampling rate of 2MHz bandwidth with neither modification nor decapsulation of the target devices, during days and nights in the open space of casual IT office buildings.

One challenge of side-channel analysis is to find a good setup of probe and points of interest [NDGJ21]. Our work does not claim to optimize the best combination of probes and its location, furthermore we will discuss the impact of probe (dis)locations in Section 4.4.

Another challenge is to empirically find the targeted frequency to be monitored by the SDR device. First, we hypothesize a centered frequency of the SDR device monitor leaking information in output traces that were captured from one bait under a clean *vs* an infected device. Thereafter, we shift an interval window of 2MHz starting from 200 MHz up to 3GHz and capture EM measurements. Each gap dataset is used to train a deep learning model of MLP. If the model achieves a high accuracy in testing, that corresponding centered frequency is assumed to leak information about the state of the device. We achieved the best results of center frequencies (Fc) as given in Table 4.2.

We collected $q = 5000$ traces each for $m = 9$ rootkits variants in regards to their input baits respectively (see Table 4.3) in both infected settings $\mathcal{M}^{\mathcal{R}}$ and 240 000 traces for benign traces $\mathcal{M}^{\Gamma}$. The same data acquisition process was conducted for both devices δ_{rasp} and δ_{ci20}. We recorded in total more than 800 000 raw traces, which took 6.6TB, thereafter we pre-processed the data as detailed in part 4.3.3.

Table 4.3 – Input baits that handled by system calls, network activities (*Net.*: we deployed 2 specific baits for this type), and keyboard emulator (*Key.*), targeting rootkit (RK) variants including obfuscated variants$^{(*)}$. List of RK activities: (H) Hide file/module/process; (N) Hide network port/socket; (L) Keylogger; (B) Remote access trojan; (R) LPE.

		Baits ($\mathcal{B}$)										**Activities**				
		System calls								*Net.*	*Key.*					
RK ($\mathcal{R}$)		getdents	readir	open	kill	read	write	stat	renameat	tcp	emu	H	N	L	B	R
kernel	diamorphine	✓			✓							✓				✓
	diamorphine$^{(*)}$	✓			✓							✓				✓
	m0ham3d	✓		✓		✓	✓					✓	✓			✓
	m0ham3d$^{(*)}$	✓		✓		✓	✓					✓	✓			✓
	adore-ng	✓	✓	✓						✓		✓	✓		✓	✓
	spy										✓			✓		
	maK_it										✓			✓		
user	beurk	✓	✓	✓				✓		✓		✓	✓		✓	
	vlany	✓	✓	✓				✓	✓	✓		✓	✓		✓	

4.3.3 Detection and classification framework

Due to the variable and length of the raw data nature outputted by the SDR, it is not suitable to be directly used in a machine learning or deep learning classification algorithm as input. We therefore preprocess the data as described below.

Data preprocessing

Our measured EM traces have varying lengths in the time domain due to a variety of factors, such as the time it takes each bait to complete, the network latency between the target device and the host computer, and the time it takes the HackRF to acquire data from the start to the end of the EM sampling without precise triggering. Furthermore, these EM traces have an enormous length, which makes them inappropriate to be utilized directly for training samples for machine learning and deep learning-based classifiers. To compensate for this unpredictability in the time domain, a fixed time segment is taken from the beginning of each EM trace. Experimentally, we observe that a segment of 0.5 s is sufficient for our dataset, but metrics can be further improved by selecting a larger time window. Like in previous works [SNA+20, NSA+17] and in Chapter 3, we then translate to the frequency domain while keeping the notion of time by using the short-time Fourier transform (STFT).

In our experiments, we tuned the STFT parameters: the window function splits the signal into chunks of length M, with an overlap O, where $(M, O) = (8192, 4096)$ gives the best results. Next, we only consider frequency bandwidths that contain interesting information. To define which bandwidths are interesting, we use NICV, that we coupled with an hill climbing algorithm following a *forward selection* [JKP94] based on the best (over time) NICV score of each bandwidth. The entire bandwidth selection procedure is described in Algorithm 2, the amount of bandwidths found by the algorithm is denoted as ϵ_{opt} and reported in all the results tables. To further reduce the data complexity to be usable by machine learning algorithms, we applied dimension reduction. We applied LDA for classification scenarios, while we determined that for detection scenarios (binary classification) kernel-PCA with *sigmoid* kernel, 15 components and default *sklearn* parameters resulted in higher effectiveness.

Detection and classification algorithms

Given the most informative bandwidth, our goal is to identify rootkit activity as effectively as possible while also classifying specific rootkit properties. One crucial constraint is our rather limited dataset (from a data analysis perspective), which contains a large amount of features even after the feature selection process (compared to data samples).

We fine-tune machine and deep learning algorithms to analyze which procedure is most suited given our datasets. Because of their simplicity, efficiency, and popularity in the community, we chose Naive Bayes (NB), Support Vector Machines (SVM), and Multi-layer Perceptions (MLP) as machine-learning and deep learning algorithms (detailed in Chapter 1). However, in order to perform the machine learning techniques such as NB and SVM efficiently, we further reduce features using Kernel PCA for binary classification and LDA when the number of classes is strictly greater than 2. The proposed MLP architecture is shown in Table 4.3.3. It was trained over 50 epochs with a batch size of 100, where we stored the model according to the highest validation accuracy in the offline phase. In our offline profiling setup we use one RTX 2080 Ti GPU; MLP performs 1 epoch below 1s during the validation tests.

Table 4.4 – Proposed MLP architecture of ULTRA framework

Layer	**Size**	**Filter**	**Activation**
Flatten	spectrogram_size	_	leaky relu
Dense	500	_	leaky relu
Dense	200	_	leaky relu
Dense	100	_	leaky relu
Dense	N	_	softmax (multi-class) or sigmoid (two-class)

4.4 Results and Discussion

4.4.1 Results

An individual dataset per device is measured for each of the 9 rootkits and their respective baits, as highlighted in Table 4.3. From these datasets, we provide a variety of detection or classification scenarios. A subset of the scenarios investigated are discussed in the following, where we highlight the results using *getdents* as bait because they trigger 7 out of 9 rootkits (see the first column of baits in Table 4.3). The remaining two rootkits, the two keyloggers *spy* and *maK_it*, are discussed in a scenario where hardware or software keyboard emulators are used as baits (Section 4.2.3).

Scenarios that use other baits (*e.g.* *network* tcp) can detect *adore-ng*, *beurk*, *vlany*, or "Hide network port/socket rootkit" activities in general with high accuracy can be found in the Appendix (Figures A.8a, A.8b and A.8c). We also investigate the classification between kernel-space and user-space rootkits, both with and without benign activities. The entire results are available in the Appendix Table A.4.

First, we conduct straightforward detection and move to classification by family and activity. Next, we show that our methodology works indeed in real-world scenarios where rootkits are unknown during training, obfuscated, or additional noise is present during the training or testing phase. In the discussion, we compare our results to open source host-based rootkit detection tools and demonstrate the effect of changing the probe location between training and testing phases.

For binary classification problems, we use balanced accuracy (BA), true positive rate (TPR), true negative rate (TNR) as metrics; for classification we use accuracy (AC), recall (RC) and the precision (PR). In practice, one technique to decrease the FNR and increase the TPR of detection is to monitor longer execution duration. In our experiments, we calculate the mean over several traces during testing, thus making a trade-off between longer raw data acquisition time (latency) and effectiveness. All reported results of binary classification are computed for $[1, 2, \ldots, 10]$ averaged traces of the testing set, where only the maximum is stated the tables to ease readability. To be complete, Figures A.2, A.3, A.4, A.5 and A.6 in Appendix illustrate results using $[1, 2, \ldots, 10]$ averaged traces for all binary scenarios. One can see that even if only one trace is available due to constraints on the response latency, high accuracy is already achievable.

Table 4.5 – Rootkit detection with the same environment between learning and testing; bait β = {*getdents*} and corresponding rootkit set $\mathcal{R}$ = {*adore*, *beurk*, *diamorphine*, *diamorphine-obed*, *m0hamed*, *m0hamed-obed*, *vlany*} with benign activity drawn randomly from Γ; tested devices: $\delta_{\text{rasp.}}$ and δ_{ci20}.

	MLP	**KPCA + NB**	**KPCA + SVM**
Device δ	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$
δ_{ci20}	$98.1_{[6]}$ $^{97.6}/_{98.6}$	$100.0_{[14]}$ $^{100.0}/_{100.0}$	$100.0_{[16]}$ $^{100.0}/_{100.0}$
$\delta_{\text{rasp.}}$	$91.8_{[16]}$ $^{91.0}/_{92.5}$	$97.8_{[11]}$ $^{96.7}/_{98.9}$	$98.0_{[15]}$ $^{100.0}/_{96.0}$

Rootkit detection

This first detection scenario shows the ability to detect known rootkits, meaning that the rootkit was seen during the offline learning phase. Thus, we have a constant environment $\tau^{\delta}_{\rho_{k,t}}(\beta)$ between the learning and testing phases. Table 4.5 shows the balanced accuracy, true positive (TPR) and true negative rate (TNR), as well as the optimal number of selected bandwidth (detailed in Subsection 4.3.3) for the bait $\beta = \{getdents\}$ and rootkit set $\mathcal{R}$ = {*adore*, *beurk*, *diamorphine*, *diamorphine-obed*, *m0hamed*, *m0hamed-obed*, *vlany*}. SVM with Kernel PCA performs best by reaching a TPR of 100% on both devices.

Rootkit classification

Table 4.6 – Classification by family and by activity obtained with MLP, LDA + NB and LDA + SVM. The column « # » gives the number of classes per scenario.

			MLP	**LDA + NB**	**LDA + SVM**
	Scenario	**#**	**AC** $_{[\epsilon_{\text{opt}}]}$ $^{\text{PR}}/_{\text{RC}}$	**AC** $_{[\epsilon_{\text{opt}}]}$ $^{\text{PR}}/_{\text{RC}}$	**AC** $_{[\epsilon_{\text{opt}}]}$ $^{\text{PR}}/_{\text{RC}}$
δ_{ci20}	family	19	$91.3_{[65]}$ $^{83.0}/_{83.0}$	$76.0_{[10]}$ $^{65.6}/_{65.4}$	$85.6_{[8]}$ $^{76.1}/_{76.3}$
	activity	46	$82.5_{[45]}$ $^{83.0}/_{82.5}$	$62.5_{[10]}$ $^{63.2}/_{62.4}$	$76.0_{[10]}$ $^{75.8}/_{76.0}$
$\delta_{\text{rasp.}}$	family	19	$82.1_{[50]}$ $^{79.1}/_{76.5}$	$54.7_{[10]}$ $^{53.9}/_{55.3}$	$66.2_{[10]}$ $^{66.9}/_{60.1}$
	activity	46	$75.0_{[40]}$ $^{75.4}/_{75.0}$	$50.6_{[10]}$ $^{51.5}/_{55.6}$	$59.2_{[9]}$ $^{59.4}/_{59.2}$

In this scenario, we go beyond detection (two-class binary classification) and classify the rootkits into multiple classes. Again, the settings between the training and

testing phase remain constant. First, we consider the classification according to their family (i.e. *diamorphine, m0ham3d, adore-ng, spy, maK_it, beurk, vlany*) which is independent of the bait or the obfuscation. Note that, clean activity is not considered as 1 family but each class corresponds to the executed benign activity, yielding a total of 19 classes. Table 4.6 shows the results for both devices, we observe that MLP performs the best, reaching 82.1% on $\delta_{rasp.}$ and 91.3% on δ_{ci20}, vs. a random guess of only 5.2%. Following that, we classify each rootkit individually and each cleanware separately, resulting in 46 classes. MLP once again outperforms with 75% on $\delta_{rasp.}$ and 82.5% on δ_{ci20}. A random guess would only result in 2.17%. These results suggest that using ULTRA, rootkits can not only be effectively detected, but also further information about the family, clean activity, and obfuscation can be revealed with remarkably high accuracy.

Rootkit novelty detection

Now, we raise the question of whether detection is still effective even when rootkits are unknown and not part of the offline learning phase. Or, generally speaking, can we detect *novel* rootkits? So, the setting is not constant between learning and testing, but with two sets of rootkits, $\mathcal{R}_{learning}$ and $\mathcal{R}_{testing}$ with $\mathcal{R}_{learning} \cap \mathcal{R}_{testing} = \emptyset$. Again we focus on the bait *getdents*, but we train one model per rootkit, i.e. $\mathcal{R}_{learning} = \{r_i\}$ with $r_i \in \{$*adore, beurk, diamorphine, m0hamed, vlany*$\}$ consists of one rootkit at a time. Additionally, we build a model where the learning (or testing) set corresponds to all rootkits except the one in the testing phase (or learning phase), i.e. $\mathcal{R}_{testing} = \{r_i\}$ and $\mathcal{R}_{learning} = \{\{$*adore, beurk, diamorphine, m0hamed, vlany* $\} \setminus r_i\}$ as well as $\mathcal{R}_{learning} = \{r_i\}$ and $\mathcal{R}_{testing} = \{\{$*adore, beurk, diamorphine, m0hamed, vlany* $\} \setminus r_i\}$ with $r_i \in \{$*adore, beurk, diamorphine, m0hamed, vlany*$\}$

Figure 4.5 illustrates the results, where each row starts with the name of a rootkit and refers to the accuracy obtained using the rootkit in the training. For comparison, we also illustrate the diagonal that corresponds to the case where $\mathcal{R}_{learning} = \mathcal{R}_{testing} = \{r_i\}$. The darker the blue color, the higher the accuracy. Results on δ_{ci20} are given on top, $\delta_{rasp.}$ are displayed on the bottom.

Again, we notice that δ_{ci20} gives higher detection rates than $\delta_{rasp.}$. In particular, for δ_{ci20} we see that for each scenario (except three) at least one algorithm is achieving a balanced accuracy of 100%. In total, SVM is performing the best and there is no large gap between the diagonal and the other entries, meaning that even though rootkits are unknown (new in the testing phase), the detection works effectively. Also, for SVM,

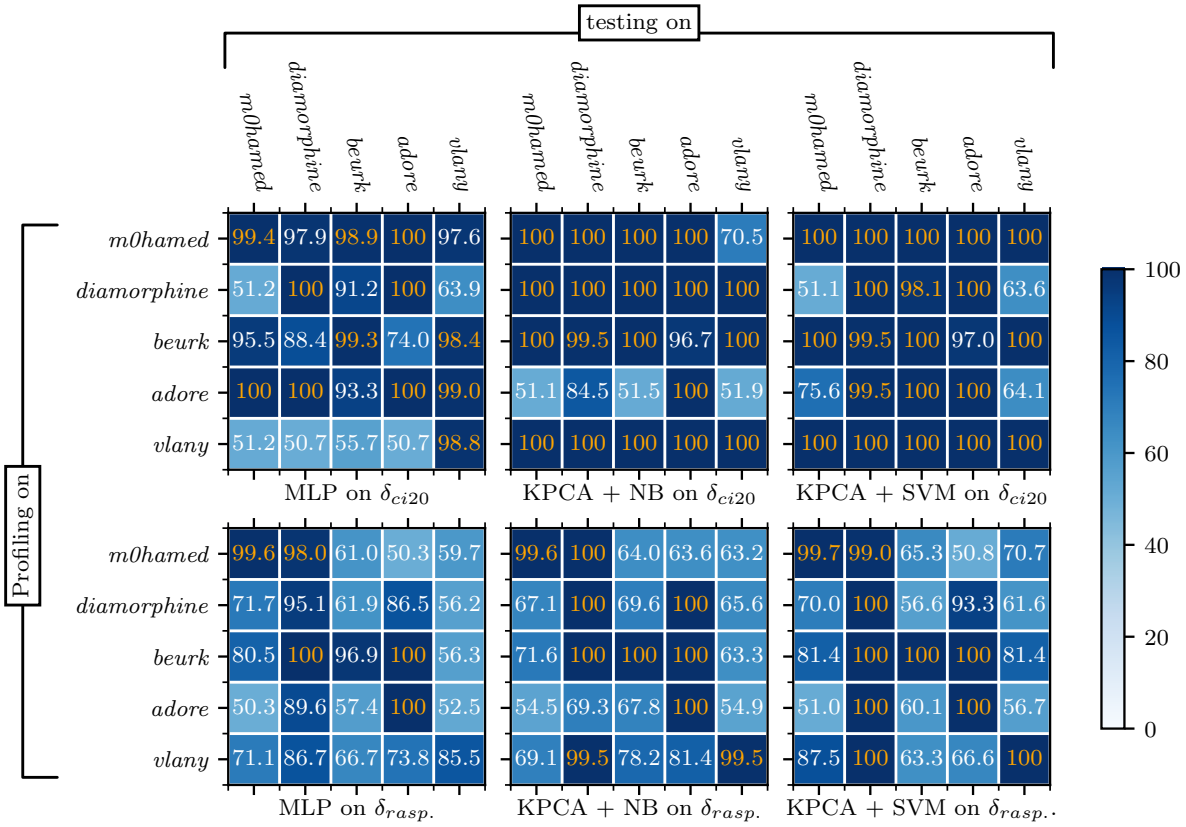


Figure 4.5 – Novelty rootkit detection. Each cell refers to an experiment, the row (resp. the column) informs on the rootkit(s) seen during the offline learning phase (resp. the online testing phase). Except on the diagonal, learning and testing sets are exclusive. Numbers are balanced accuracies, the darker the blue color the higher the balanced accuracy. All experiments share the same bait $\beta = \{getdents\}$, the same probe $p_{k,t}$, and benign activities are drawn randomly from Γ. δ_{ci20} on top; $\delta_{rasp.}$ on bottom.

we see no significant differences between rootkits, even if they work on different levels (user vs. kernel protection ring).

For $\delta_{rasp.}$ the detection is still effective in most cases even if the rootkit is novel in the testing phase, although there are more derivations between experiments. Still, for each rootkit, one can find at least one model that performs greater than 78.2% in all scenarios. We see that SVM performs slightly better than MLP and NB (on average) among the three methods. For example, using SVM we are able to detect *diamorphine* with 100% accuracy when training only on *adore*. Remarkably, looking at each column, ignoring the diagonal, we always have at least one model from the three algorithms that can detect unseen rootkit samples with an accuracy higher than 78% for $\delta_{rasp.}$ and 100% for δ_{ci20}.

Obfuscated Rootkit detection

To evaluate the effectiveness of ULTRA in the presence of obfuscated rootkits, we performed tests with the two rootkit samples *m0hamed* (m) and *diamorphine* (d) , as well as their obfuscated variants *m0hamed-obed* (m_o) and *m0hamed* (d_o). Table 4.7 summarizes the results for both devices. For each device, the first four lines refer to m and m_o while the last four lines refer to d and d_o. First, we detect the original or obfuscated rootkit while learning in the same setting. We denote this scenario by $r_i \rightarrow r_i$ with $r_i \in \{m, m_0, d, d_o\}$. Second, we train on the original or obfuscated and test the other one, i.e. $r_i \rightarrow r_j$ with $i \neq j$, r_i and r_j belonging to the same family. The major finding is that, in general, the static-code obfuscation mechanism has a very low impact on the detection rate, and ULTRA is able to detect the original and obfuscated variant. Whatever variant of the rootkit (with or without obfuscation), it is able to detect both variants with nearly no mistakes. In particular for δ_{ci20}, NB is achieving 100% in all scenarios, whereas for $\delta_{rasp.}$ SVM is performing best with 100% in all but one scenario. Remarkably, we observe that there is no drop in effectiveness when ULTRA is detecting obfuscated variants and reaching 100% effectiveness.

Keylogger novelty detection

Table 4.8 indicates how efficient ULTRA is in detecting keyloggers that are unknown to the system (not present in the learning phase). We use two baits to monitor keylogger activity: a hardware keyboard emulator denoted *hwkb* and a software bait

Table 4.7 – Detection scenarios on obfuscated rootkits. The bait β is *getdents*, $\mathcal{R} = \{$*diamorphine* (d), *diamorphine-obed* (d_o), *m0hamed* (m), *m0hamed-obed* $(m_o)\}$ with benign activity randomly drawn from Γ, tested on both devices $\delta_{\text{rasp.}}, \delta_{\text{ci20}}$. We use the notation $r_1 \rightarrow r_2$ to express that r_1 was used in the learning phase, and r_2 in testing. For $r_1 = r_2$ the environment stays constant, and for $r_1 \neq r_2$ we detect unseen binaries.

		MLP	**KPCA + NB**	**KPCA + SVM**
	Scenario	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$
δ_{ci20}	$m \rightarrow m$	$99.4_{[5]}$ $^{99.0}/_{99.8}$	$100.0_{[7]}$ $^{100.0}/_{100.0}$	$100.0_{[1]}$ $^{100.0}/_{100.0}$
	$m_o \rightarrow m_o$	$100.0_{[13]}$ $^{100.0}/_{100.0}$	$100.0_{[15]}$ $^{100.0}/_{100.0}$	$100.0_{[10]}$ $^{100.0}/_{100.0}$
	$m \rightarrow m_o$	$100.0_{[13]}$ $^{100.0}/_{100.0}$	$100.0_{[15]}$ $^{100.0}/_{100.0}$	$100.0_{[10]}$ $^{100.0}/_{100.0}$
	$m_o \rightarrow m$	$97.4_{[5]}$ $^{96.7}/_{98.2}$	$100.0_{[7]}$ $^{100.0}/_{100.0}$	$100.0_{[1]}$ $^{100.0}/_{100.0}$
	$d \rightarrow d$	$100.0_{[12]}$ $^{100.0}/_{100.0}$	$100.0_{[9]}$ $^{100.0}/_{100.0}$	$100.0_{[7]}$ $^{100.0}/_{100.0}$
	$d_o \rightarrow d_o$	$100.0_{[10]}$ $^{100.0}/_{100.0}$	$100.0_{[8]}$ $^{100.0}/_{100.0}$	$100.0_{[9]}$ $^{100.0}/_{100.0}$
	$d \rightarrow d_o$	$53.7_{[10]}$ $^{7.4}/_{100.0}$	$100.0_{[8]}$ $^{100.0}/_{100.0}$	$58.3_{[9]}$ $^{16.6}/_{100.0}$
	$d_o \rightarrow d$	$52.3_{[12]}$ $^{4.9}/_{99.6}$	$100.0_{[9]}$ $^{100.0}/_{100.0}$	$53.8_{[7]}$ $^{8.6}/_{99.1}$
$\delta_{\text{rasp.}}$	$m \rightarrow m$	$99.6_{[34]}$ $^{99.2}/_{100.0}$	$99.6_{[17]}$ $^{99.3}/_{100.0}$	$99.7_{[36]}$ $^{99.4}/_{100.0}$
	$m_o \rightarrow m_o$	$100.0_{[25]}$ $^{100.0}/_{100.0}$	$100.0_{[14]}$ $^{100.0}/_{100.0}$	$100.0_{[30]}$ $^{100.0}/_{100.0}$
	$m \rightarrow m_o$	$98.2_{[25]}$ $^{96.3}/_{100.0}$	$90.3_{[14]}$ $^{80.6}/_{100.0}$	$100.0_{[30]}$ $^{100.0}/_{100.0}$
	$m_o \rightarrow m$	$100.0_{[34]}$ $^{100.0}/_{100.0}$	$100.0_{[17]}$ $^{100.0}/_{100.0}$	$100.0_{[36]}$ $^{100.0}/_{100.0}$
	$d \rightarrow d$	$95.1_{[4]}$ $^{97.2}/_{93.1}$	$100.0_{[21]}$ $^{100.0}/_{100.0}$	$100.0_{[21]}$ $^{100.0}/_{100.0}$
	$d_o \rightarrow d_o$	$100.0_{[22]}$ $^{100.0}/_{100.0}$	$100.0_{[11]}$ $^{100.0}/_{100.0}$	$100.0_{[16]}$ $^{100.0}/_{100.0}$
	$d \rightarrow d_o$	$100.0_{[22]}$ $^{100.0}/_{100.0}$	$97.5_{[11]}$ $^{95.0}/_{100.0}$	$100.0_{[16]}$ $^{100.0}/_{100.0}$
	$d_o \rightarrow d$	$83.0_{[4]}$ $^{66.2}/_{99.8}$	$100.0_{[21]}$ $^{100.0}/_{100.0}$	$100.0_{[21]}$ $^{100.0}/_{100.0}$

swkb. Regarding the targets, in this scenario, we can detect them more accurately on $\delta_{rasp.}$, reaching 100% using SVM which again is remarkable.

Benign kernel-level module evaluation

We investigate the impact of additional kernel-level modules on the accuracy of detection and classification. For this we build a LKM dataset that is based on drivers

Table 4.8 – Detection scenarios of keyloggers unseen during the offline profiling phase. The baits β are software or hardware *keyboard emulator* respectively denoted as *swkb* and *hwkb*. The tested devices are $\delta_{\text{rasp.}}, \delta_{\text{ci20}}$, and benign activity is drawn randomly from Γ. When the learning has been done with *spy* and the testing with *maK_it*, we write $s \twoheadrightarrow m$ while the reverse scenario is denoted by $m \twoheadrightarrow s$.

			MLP	**KPCA + NB**	**KPCA + SVM**
		Scenario	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$
δ_{ci20}	$s \twoheadrightarrow m$	*swkb*	$66.8_{[5]}$ $^{34.8}/_{98.8}$	$94.9_{[9]}$ $^{89.8}/_{100.0}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$
		hwkb	$50.6_{[8]}$ $^{92.8}/_{8.4}$	$50.0_{[8]}$ $^{0.0}/_{100.0}$	$51.0_{[16]}$ $^{2.9}/_{99.2}$
	$m \twoheadrightarrow s$	*swkb*	$57.1_{[15]}$ $^{14.8}/_{99.5}$	$100.0_{[13]}$ $^{100.0}/_{100.0}$	$100.0_{[7]}$ $^{100.0}/_{100.0}$
		hwkb	$46.9_{[15]}$ $^{43.5}/_{50.2}$	$50.0_{[27]}$ $^{0.0}/_{100.0}$	$48.2_{[10]}$ $^{0.1}/_{96.3}$
$\delta_{\text{rasp.}}$	$s \twoheadrightarrow m$	*swkb*	$100.0_{[23]}$ $^{100.0}/_{100.0}$	$100.0_{[8]}$ $^{100.0}/_{100.0}$	$100.0_{[8]}$ $^{100.0}/_{100.0}$
		hwkb	$100.0_{[14]}$ $^{100.0}/_{100.0}$	$100.0_{[10]}$ $^{100.0}/_{100.0}$	$100.0_{[13]}$ $^{100.0}/_{100.0}$
	$m \twoheadrightarrow s$	*swkb*	$100.0_{[11]}$ $^{100.0}/_{100.0}$	$100.0_{[9]}$ $^{100.0}/_{100.0}$	$100.0_{[4]}$ $^{100.0}/_{100.0}$
		hwkb	$99.5_{[15]}$ $^{99.1}/_{100.0}$	$85.3_{[9]}$ $^{70.6}/_{100.0}$	$100.0_{[12]}$ $^{100.0}/_{100.0}$

Table 4.9 – Detection of malware with additive benign kernel activities during the learning phase only (S_0) or during the testing phase only (S_1). The bait β is *getdents*, tested on the device δ_{ci20}, the malware $\mathcal{R} = \{\text{m0hamed}\}$ and the benign activity is randomly chosen from Γ.

	MLP	**KPCA + NB**	**KPCA + SVM**
Scenario	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$
S_0	$100.0_{[4]}$ $^{100.0}/_{100.0}$	$100.0_{[13]}$ $^{100.0}/_{100.0}$	$100.0_{[21]}$ $^{100.0}/_{100.0}$
S_1	$55.4_{[5]}$ $^{99.0}/_{11.7}$	$100.0_{[7]}$ $^{100.0}/_{100.0}$	$55.7_{[1]}$ $^{100.0}/_{11.4}$

having extra benign activities that may deceive the models (e.g. *netfilter*, *VFS* hooks, *ecryptfs* etc.). Table 4.9 shows that with benign LKM added in the learning phase, all models achieve 100%. However, the introduction of benign LKM during the testing decreases the accuracy of the SVM and MLP models, but has no impact on the NB model, which can still detect at 100% with no false positive. In conclusion, even if there is additional unexpected activity from the kernel-space, NB is able to detect without any errors.

Noise evaluation

Table 4.10 – Detection scenarios with rootkits seen during the learning phase but with different background benign activity levels: the « quiet » (Q) level with $\Gamma = \Gamma_0$, meaning no background benign activity, and the « noisy » (N) level $\Gamma = \{\Gamma_0, \ldots, \Gamma_n\}$. On the left (resp. on the right) of the arrow ($\rightarrow$) in the column « Scenario » we give the noise level used during the offline profiling (resp. the online testing). The bait β is *write*, tested on both devices $\delta_{\text{rasp.}}, \delta_{\text{ci20}}$, the malware $\mathcal{R} = \{\text{m0hamed-obed}\}$.

		MLP	**KPCA + NB**	**KPCA + SVM**
Scenario		**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$
δ_{ci20}	$N \rightarrow Q$	$54.1_{[6]}$ $^{96.1}/_{12.2}$	$98.7_{[7]}$ $^{100.0}/_{97.4}$	$100.0_{[12]}$ $^{100.0}/_{100.0}$
	$Q \rightarrow N$	$58.1_{[3]}$ $^{100.0}/_{16.3}$	$52.9_{[7]}$ $^{100.0}/_{5.9}$	$50.0_{[3]}$ $^{100.0}/_{0.0}$
	$N \rightarrow N$	$99.1_{[6]}$ $^{98.2}/_{100.0}$	$100.0_{[7]}$ $^{100.0}/_{100.0}$	$100.0_{[12]}$ $^{100.0}/_{100.0}$
$\delta_{\text{rasp.}}$	$N \rightarrow Q$	$100.0_{[11]}$ $^{100.0}/_{100.0}$	$100.0_{[10]}$ $^{100.0}/_{100.0}$	$50.0_{[11]}$ $^{100.0}/_{0.0}$
	$Q \rightarrow N$	$50.0_{[1]}$ $^{100.0}/_{0.0}$	$50.0_{[1]}$ $^{100.0}/_{0.0}$	$50.0_{[1]}$ $^{100.0}/_{0.0}$
	$N \rightarrow N$	$100.0_{[11]}$ $^{100.0}/_{100.0}$	$100.0_{[10]}$ $^{100.0}/_{100.0}$	$100.0_{[11]}$ $^{100.0}/_{100.0}$

In this scenario, we evaluate the influence of benign activity in the background and assess robustness. We measured the SDR measurement traces of « noisy » (N) (noisy benign background activities) and « quiet » (Q) (no additional benign activity besides the OS background processes). Table 4.10 demonstrates the necessity of noise during the training phase. In fact, models build with no extra background benign activity ($Q \rightarrow N$) get high TPR and low TNR. That is, those models are not able to distinguish rootkits from benign and classify all activity as malicious. However, models built with noisy traces (N) reach 100% (for at least one model) even when used in a quiet testing environment (Q). This result shows how important it is to include usual noise activities in the training phase in practice.

4.4.2 Discussion

Performance evaluation

We compare our detection results to other up-to-date opensource tools: rkhunter v1.4.6 (2018), chkrootkit v0.54 (2021), and LKRG v0.91 (2021). The 3 tools are host-

based techniques compiled under ARM architecture and all experiments are conducted on the same platform as δ_{Raspi}, except ULTRA was executed on a remote host agent: Intel Xeon W-2104@4x 3.2GHz CPU 64GB RAM with Quadro P2200 GPU. It is worth mentioning that LKRG is specifically required kernel-rebuilt to enable protection flags such as kernel symbols extraction and seccomp. Since the detection component is based on open source and cross-compilation capable GNURadio, sklearn and Tensorflow, the ULTRA detector can be deployed on portable devices (e.g. Raspberry Pi, NVIDIA Jetson, etc.) along with SDR receivers.

Table 4.11 shows the detection results and execution latency of all scans, where all results are averaged for 10 executions. ULTRA takes 1.5s for MLP on the CPU, and 1.3s using the GPU. Rkhunter and chkrootkit are unable to detect obfuscated variants of *diamorphine* and *m0ham3d* LKRG aims to detect kernel-level rootkits, so that 4/9 are not detected. Noticeably, while detecting triggered behaviors of *adore-ng* the device OS crashed in kernel panic and thus brings the device in an unusable mode in a real-world setting. ULTRA detected rootkits living at both kernel and user space mentioned in Table 4.3, and regardless of the stage before or after rootkit infection, while integrity detection solutions must be installed before the moment the rootkit infected the device. Further, ULTRA detects rootkits by actively using baits without fully triggering malicious behaviors of the rootkit (except in the keylogger detection scenario), in contrast to LKRG which requires the occurrence of malicious behaviors explicitly.

Invariant to probe position

Figure 4.6 shows a detection scenario setup of 2 different probe locations on the device δ_{ci20}. Probe $\rho_{k=0,t=0}$ is used for offline training and probe $\rho_{k=1,t=0}$ with the same type $t = 0$, but a different location was used for testing exclusively. Furthermore, we conduct an experimental setup of ULTRA with a handcrafted EM compatible probe $\rho_{k=2,t=1}$ which consists of ferrites and conductor (see Fig.4.7). The bait *open* was used to detect the presence of rootkit *beurk*. Table 4.12 shows the results. We observe that the location has a low impact on the accuracy. In fact, a model trained with $\rho_{k=0,t=0}$ can detect rootkit signatures acquired with $\rho_{k=1,t=0}$ still with 100% whatever the classification algorithm. However, when changing position and probe ($\rho_{k=0,t=0}$ in training, and $\rho_{k=1,t=2}$ in testing), MLP is performing the best with only 60%.

Concluding, this result shows that the ULTRA framework works even with the relocation of the probe position, but care should be taken with the type of the probe. It

Table 4.11 – Performance evaluation of rootkit (RK) and their obfuscated variants$^{(*)}$ detection results, and execution latency. List of indicators: (✓) RK detected; (-) Not detected; (†) Malicious behavior trigger required; (⚠) Kernel panicked; Executed on (‡) CPU ; (§) GPU.

RK	**AV solutions**			
	rkhunter	*chkrootkit*	*LKRG*	*ULTRA*
diamorphine	✓	-	✓†	✓
diamorphine$^{(*)}$	-	-	✓†	✓
m0ham3d	✓	-	✓†	✓
m0ham3d$^{(*)}$	-	-	✓†	✓
adore-ng	-	-	✓†⚠	✓
spy	-	-	-	✓
maK_it	-	-	-	✓
beurk	-	-	-	✓
vlany	-	-	-	✓
Latency *(sec)*	1326.6‡	44.3‡	2.6‡	1.3§-1.5‡

Table 4.12 – Detection scenarios with three distinct probes locations $k \in \{0, 1, 2\}$ and two different types $t \in \{0, 1\}$. The bait β is *open*, tested on the devices δ_{ci20}, the malware $\mathcal{R} = \{beurk\}$ and the benign activity is randomly drawn from Γ. Notation used in Scenario: $\{k, t\}$ in learning $\rightarrow \{k, t\}$ in testing.

	MLP	**KPCA + NB**	**KPCA + SVM**
Scenario	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$	**BA** $_{[\epsilon_{\text{opt}}]}$ $^{\text{TPR}}/_{\text{TNR}}$
$\{0,0\}\rightarrow\{0,0\}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$
$\{0,0\}\rightarrow\{1,0\}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$
$\{0,0\}\rightarrow\{2,1\}$	$60.6_{[2]}$ $^{21.4}/_{99.9}$	$50.0_{[2]}$ $^{0.0}/_{100.0}$	$50.0_{[2]}$ $^{0.0}/_{100.0}$
$\{1,0\}\rightarrow\{1,0\}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$	$100.0_{[3]}$ $^{100.0}/_{100.0}$	$100.0_{[2]}$ $^{100.0}/_{100.0}$
$\{2,1\}\rightarrow\{2,1\}$	$100.0_{[1]}$ $^{100.0}/_{100.0}$	$100.0_{[4]}$ $^{100.0}/_{100.0}$	$100.0_{[4]}$ $^{100.0}/_{100.0}$

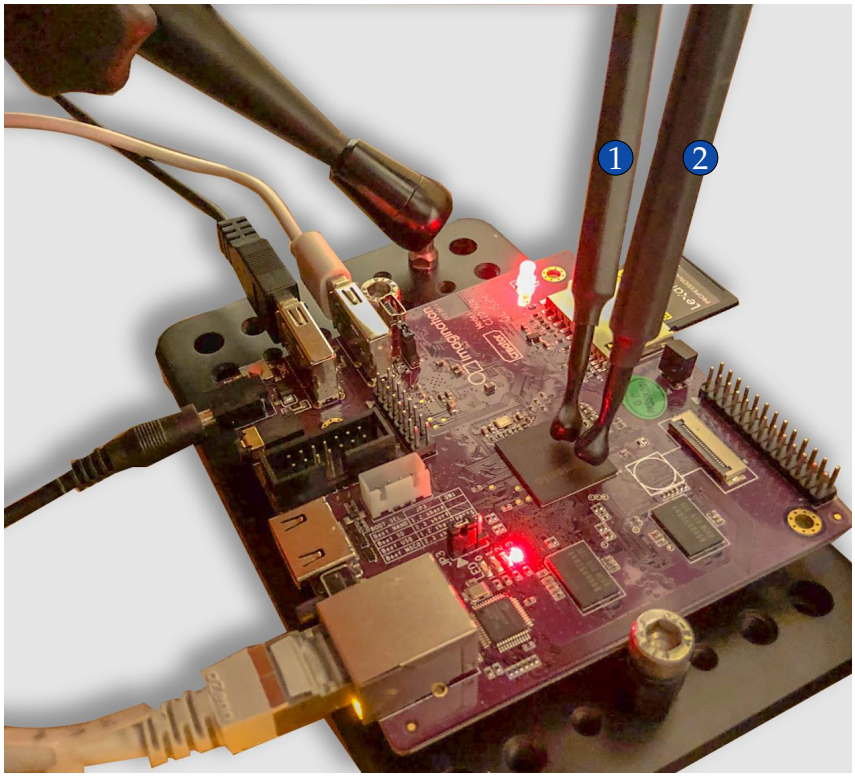


Figure 4.6 – Invariant to probe position. Framework setup with 2 probes 1 2 of the same type placing contactless at 2 different locations placed 10mm above the processor.

indicates that our framework has a significant advantage and is powerful when detecting rootkits in on-site scenarios with portable equipment, or in incident response and digital forensics on a large scale.

Threats to validation

Adversaries to manipulate the ML and DL models

One actor may be able to manipulate the detection model by reverse engineering and creating a rootkit that can evade the models, however this is out of scope for this work.

Noise generation to decrease accuracy

Frieslaar *et. al.* [FI18] proposed a countermeasure to a SDR side-channel attack on the Raspberry Pi by generating EM noise that consists of executing arithmetic instructions in an infinite loop. Noise-SDR [CF22] presents a novel technique by exploiting

Figure 4.7 – ULTRA framework is installed with an handcrafted EM-compatible probe (1) placed 10mm above Ci20's processor (2) to detect *beurk* rootkit.

DRAM accesses to shape arbitrary signals out of EM noise from unprivileged software. Therefore, a rootkit may try to tamper by generating noise during the execution. However, it still raises a challenge to evade for rootkit authors: finding a solution to generate noise without making any deviation between infected and benign state, *i.e.* will be detected by the anomaly detectors, and the limitation of the signal's bandwidth (*e.g.* Noise-SDR can only generate signals that are limited by the target device leakage and sampling rate).

The rootkit may undo modifications

This concern has been discussed in [WK13], if an advanced rootkit is aware of the occurrence of a bait, it can hide its modifications before the bait is presented and reactivate them afterwards. As per the bait design requirement, the execution of a bait must present no difference from the execution of benign programs. For example, the execution of *getdents* bait is equivalent to one simple binary which lists the content

of the current directory. Additionally, the detector can randomize the intervals and iterations inside the baits to avoid the rootkit's prediction of the checking period.

4.5 Conclusion

Stealthy nonactive rootkits constitute a real challenge for host-based malware analysis systems. In this chapter, we introduced the ULTRA framework that operates outside the box by relying solely on the EM activity of IoT devices and evokes rootkit behavior by baiting.

We investigated a large number of experiments and scenarios to show that our methodology is robust in real-world scenarios and can be applied like a wave-and-play solution. We achieve detection rates of 100% for known and unknown variants, while the device performs random and varied benign activities (plus active OS operations). Further, we show that probe dislocation results in no loss of effectiveness, making ULTRA highly practicable and employable. Furthermore, the classification of rootkit families achieved up to 91.3% accuracy (vs a random guess of 5.2%), and even more we are able to classify exact activity with up to 82.5% (vs a random guess of 2.17%). The comparison of our solution to open-source host-based solutions shows superiority of ULTRA on all levels: effectiveness, latency, resource requirements, stability on the target device.

This work opens up new research areas pursuing the classic cat-and-mouse game: improving detection and classification rates or evading and making our approach harder. An appealing enhancement could be to increase the capability of the framework by integrating even more baits, thus providing the potential to detect APT rootkits.

Besides, ULTRA may provide cues to manufacturers to build a standalone solution that uses electromagnetic waves to detect malware and similar threats for other platforms such as PLC, Linux servers, etc. in the future. Further empirical research could be conducted to investigate the power of single-board computers so that ULTRA can be deployed in a fully portable manner.

Conclusion and Perspectives

In this chapter, we summarize the presented contributions of this manuscript, and to highlight short-term and long-term perspectives for future works as well as questions that remain open.

Answers to research questions

We addressed the following research problems and our answers along this thesis:

RQ1 *How can we build and setup an IoT malware analysis and detection on embedded device?* Since automated malware analysis techniques are matured on other platforms, but they have shortcomings that cannot be applied straightforward for IoT systems. It is important to create an advanced IoT malware analysis framework and overcome malware evasion tactics. In this thesis, we proposed AHMA and ULTRA framework that automatically performs EM captures from blackbox devices executing malware. The EM traces can be further investigated and classified to detect and classify malware.

RQ2 *If a malware analyst has a dataset of unlabeled binaries. Would it be possible to classify the dataset into labeled types, families, variants of malware or rootkits, obfuscation techniques used etc.?*

In Chapter 3, a large dataset of malware has been studied to understand generic IoT malware families, variants and their *modus operandi*. Further, we analyzed 100 000 traces recorded from 35 malicious samples including different types, fam-

ilies and obfuscation techniques used. The classification results from the dataset in different scenarios with accuracy ranging from 82.28% to 99.82%. In Chapter 4, we analyzed 800 000 traces recorded from 9 rootkit variants. We conducted several scenarios to classify into different families, activities and rootkit protection level with high accuracy.

RQ3 *Is it feasible to utilize EM for stealthy rootkit detection on embedded devices?*

Our results in Chapter 4 showed that it is possible to differentiate the status of embedded devices under stealthy malware such as rootkits with detection accuracy for many scenarios at 100% rates. Although the detection rate of 100% was obtained with a limited sample size due to the rarity of Linux IoT rootkits in the wild, it is a solid sign that our technique is promising and novel.

A real-world IoT malware test bed using side-channel.

In this manuscript, we first surveyed the area of malware and rootkit detection, specifically using side-channel, and then we presented some works aimed in improving rootkit detection both from a theoretical and practical point of view. In Chapter 1 and Chapter 3, we studied the IoT malware classification using Electromagnetic Emanations (EM).

Compared to the state of the art, our work provides a novel method of identifying IoT malware using side channel information, specifically EM, instead of installing software on the targeted device and monitoring its operations or finding static features. We classify malware utilizing in-the-wild malware samples rather than proof-of-concept samples. We conducted several scenarios including benign datasets and obfuscated malware to demonstrate the resilience of the models while most previous research often use anomaly detection with a limited sample size.

We were able to not only detect, but also determine the type of real-world malware infecting a Raspberry Pi running a full Linux OS, with an encouraging results, Specifically, it can detect the type of malware with greater than 98% accuracy, where CNN surpasses the others. It has also been demonstrated that it is successful in detecting novel malware with 99.38% accuracy. Obfuscation techniques are likewise classified with 82.70 percent accuracy, indicating that CNN outperforms the rest once again.

Given our experimental results, malware analysts can use our comprehensive technique to acquire a better understanding of the variations, types, families, and/or evo-

lution of malware actors and campaigns, especially when software detection systems fail due to malware evasion, or cannot be deployed due to restricted resources on the embedded device.

Real-time detection of IoT rootkits using side-channel

For host-based malware analysis systems, stealthy rootkits pose a significant challenge for detection. Side-channel based techniques such as EM collecting traces from oscilloscopes are ineffective when stealthy rootkits are not yet disclosing harmful activity.

We proposed the ULTRA framework in this thesis, which utilizes electromagnetic waves to achieve detection rates of up to 100% for known and novel rootkit variants, while the embedded device performs usual tasks and various benign processes as in practical industrial application.

This approach has the potential to become a stand-alone and portable solution for detecting not only stealthy rootkits, but also other advanced malware and APT campaigns on any platform.

Perspectives

IoT malware is expanding since various malicious resources are easy to access on open source platforms, where malware actors can reuse and extend with new features and introduce new vulnerability exploits. With the rapid proliferation of advanced malware in APT campaigns, as well as the growth in the number of IoT devices deployed in organizations and government, we anticipate that the importance of IoT malware detection solutions and automated IoT malware analysis framework must be considered.

The major use of AHMA is to provide an environment for analysts to experiment with IoT malware. Would such a strategy be effective in terms of IoT security: scalability and affordability are likely to be significant challenges, as we cannot place an additional bulky device next to each and every IoT device. Even if the strategy is non-intrusive and clearly beneficial to analysts, we still require a method that can be used more handy and in real-time to safeguard IoT devices from malware threats.

ULTRA proposed some solutions to overcome shortcomings of AHMA, and future work may concentrate on the extension of our approach to real-time generic malware classification systems. Based on results accomplished of ULTRA from Chapter 4, this can be considered as a first step towards real-time and low-cost IoT behavioral analysis through electromagnetic emanation, opening a new research direction for future work.

Malware identification and prevention can be considered as similar to a cat-and-mouse game: As new malware analysis techniques are explored, malware authors invent new strategies to avoid detection. Our results have the potential to contribute to the development for further research into malware evasion of side-channel signals monitor.

Reproducibility

Our IoT malware classification framework AHMA can be reproducible. The artifacts associated with the AHMA are found to be documented, consistent, complete, exercisable, and include appropriate evidence of verification and validation by Annual Computer Security Applications Conference (ACSAC) Artifacts 2021 committee [1].

Furthermore, we open access for ULTRA artifacts [2] including source code, captured dataset and trained models.

Our IoT malware classification framework AHMA and ULTRA can be reproducible. The artifacts associated with the AHMA are proved to be documented, consistent, complete, exercisable, and include appropriate evidence of verification and validation by Annual Computer Security Applications Conference (ACSAC) Artifacts 2021 committee [3]. ULTRA's artifacts [4] including source code, dataset and trained models are published and open-access.

1. https://www.acsac.org/2021/program/artifacts/
2. https://gitlab.com/ultra-RK/ultra/
3. https://www.acsac.org/2021/program/artifacts/
4. https://gitlab.com/anon-ultra/ultra

Media coverage

The publishing of our work in Chapter 3 and [PMMH21] has attracted a great attention from the media, including sources such as Schneier [5], HackAday [6], The Hacker News [7], and 01net [8]. It would increase public awareness of detecting malware and rootkits using electromagnetism emanations for cybersecurity researchers in both academia and industry. Our work is a first step towards malware behavioral analysis through electromagnetic emanation, opening a new research direction for future work.

5. https://www.schneier.com/blog/archives/2022/01/using-em-waves-to-detect-malware.html
6. https://hackaday.com/2022/01/19/identifying-malware-by-sniffing-its-em-signature/
7. https://thehackernews.com/2022/01/detecting-evasive-malware-on-iot.html
8. https://www.01net.com/actualites/on-peut-detecter-des-malwares-avec-precision-grace-aux-ondes-electromagnetiques-2053625.html

This page intentionally left blank

Bibliography

[AAB+17] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*, pages 1093–1110, 2017.

[AARR02] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The EM side—channel (s). In *International workshop on cryptographic hardware and embedded systems*, pages 29–45. Springer, 2002.

[ACH15] Ioannis Andrea, Chrysostomos Chrysostomou, and George Hadjichristofi. Internet of Things: Security vulnerabilities and challenges. In *2015 IEEE symposium on computers and communication (ISCC)*, pages 180–187. IEEE, 2015.

[ADCC18] Amin Azmoodeh, Ali Dehghantanha, Mauro Conti, and Kim-Kwang Raymond Choo. Detecting crypto-ransomware in IoT networks based on energy consumption footprint. *Journal of Ambient Intelligence and Humanized Computing*, 9(4):1141–1152, August 2018. Accessed on 2019-03-20.

[AG18] Vipindev Adat and Brij B Gupta. Security in Internet of Things: issues, challenges, taxonomy, and architecture. *Telecommunication Systems*, 67(3):423–441, 2018.

[AoI21] National Security Agency and Federal Bureau of Investigation. Russian GRU 85th GTsSS Deploys Previously Undisclosed Drovorub Malware. https://media.defense.gov/2020/Aug/13/2002476465/-1/-1/0/CSA_DROVORUB_RUSSIAN_GRU_MALWARE_AUG_2020.PDF, 2021. Accessed: 2022-01-01.

[AQN+11] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7(4):247–258, November 2011. Accessed on 2020-04-02.

[ARR] ARRL. Ulrich Rohde, N1UL, Recognized for Pioneering Work on SDR. https://www.arrl.org/news/ulrich-rohde-n1ul-recognized-for-pioneering-work-on-sdr.

[BAT19] Mohammad Bagher Bahador, Mahdi Abadi, and Asghar Tajoddin. HLMD: a signature-based approach to hardware-level behavioral malware detection and classification. *The Journal of Supercomputing*, March 2019. Accessed on 2019-03-21.

[Bau99] Arthur O Bauer. Some aspects of military line communications as deployed by the german armed forces prior to 1945. In *The History of Military Communications, Proc. 5th Annual Colloquium*, 1999.

[BCH08] Rory Bray, Daniel Cid, and Andrew Hay. *OSSEC host-based intrusion detection guide*. Syngress, 2008.

[BCK+10] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS*, 2010.

[BDGN14] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. NICV: Normalized Inter-Class Variance for Detection of Side-Channel Leakage. In *International Symposium on Electromagnetic Compatibility (EMC '14 / Tokyo)*. IEEE, May 12-16 2014. eprint version: https://eprint.iacr.org/2013/717.pdf.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[Ber05] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[BGI11] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting Kernel-Level Rootkits Using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5):670–684, 2011.

[BH12] Michael Boelen and John Horne. The rootkit hunter project. *Online. http://rkhunter.sourceforge.net*, 2012. Accessed on 2021-06-23.

[BJN+18] Robert Bridges, Jarilyn Hernández Jiménez, Jeffrey Nichols, Katerina Goseva-Popstojanova, and Stacy Prowell. Towards malware detection via cpu power consumption: Data collection design and analytics. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1680–1684. IEEE, 2018.

[BLRS10] Sergey Bratus, Michael E Locasto, Ashwin Ramaswamy, and Sean W Smith. VM-based security overkill: a lament for applied systems security research. In *Proceedings of the 2010 New Security Paradigms Workshop*, pages 51–60, 2010.

[BLS13] Donabelle Baysa, Richard M Low, and Mark Stamp. Structural entropy and metamorphic malware. *Journal of computer virology and hacking techniques*, 9(4):179–192, 2013.

[Blu12] Bill Blunden. *The Rootkit arsenal: Escape and evasion in the dark corners of the system*. Jones & Bartlett Publishers, 2012.

[BS08] Yuriy Bulygin and David Samyde. Chipset based approach to detect virtualization malware. *BlackHat Briefings USA*, 2008.

[BSRB15] Dmitri Bekerman, Bracha Shapira, Lior Rokach, and Ariel Bar. Unknown malware detection using network traffic classification. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 134–142. IEEE, 2015.

[Bun04] Andreas Bunten. Unix and linux based rootkits techniques and countermeasures. In *16th Annual First Conference on Computer Security Incident Handling, Budapest*, 2004.

[BVN16] Robert Buhren, Julian Vetter, and Jan Nordholz. The threat of virtualization: Hypervisor-based rootkits on the ARM architecture. In *International Conference on Information and Communications Security*, pages 376–391. Springer, 2016.

[CAM+08] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[CDD+14] Christophe Clavier, Jean-Luc Danger, Guillaume Duc, M. Abdelaziz Elaabid, Benoît Gérard, Sylvain Guilley, Annelie Heuser, Michael Kasper, Yang Li, Victor Lomné, Daisuke Nakatsu, Kazuo Ohta, Kazuo Sakiyama, Laurent Sauvage, Werner Schindler, Marc Stöttinger, Nicolas Veyrat-Charvillon, Matthieu Walle, and Antoine Wurcker. Practical improvements of side-channel attacks on AES: feedback from the 2nd DPA contest. *J. Cryptogr. Eng.*, 4(4):259–274, 2014.

[CF22] Giovanni Camurati and Aurelien Francillon. Noise-SDR: Arbitrary modulation of electromagnetic noise from unprivileged software and its impact on emission security. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2022.

[CGFB18] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux Malware. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 161–175, May 2018. ISSN: 2375-1207.

[CIYD+15] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. *arXiv preprint arXiv:1512.08546*, 2015.

[CKM21] Nikhil Chawla, Harshit Kumar, and Saibal Mukhopadhyay. Machine Learning in Wavelet Domain for Electromagnetic Emission Based Malware Analysis. *IEEE Transactions on Information Forensics and Security*, 16:3426–3441, 2021.

[CMM+] Christian Collberg, Sam Martin, Jonathan Myers, Bill Zimmerman, Petr Krajca, Gabriel Kerneis, Saumya Debray, and Babak Yadegari. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/index.html. Accessed on 2020-05-14.

[CPM+18] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 163–177, Toronto Canada, October 2018. ACM.

[CRR+13] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Wenyuan Xu, and Kevin Fu. WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices. In *2013 USENIX Workshop on Health Information Technologies (HealthTech 13)*, Washington, D.C., August 2013. USENIX Association.

[CS221] CS231N. Convolutional neural networks for visual recognition. https://cs231n.github.io/convolutional-networks/, 2021. Accessed: 2022-01-01.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[CTY13] Julia Yu-Chin Cheng, Tzung-Shian Tsai, and Chu-Sing Yang. An information retrieval approach for malware classification based on Windows API calls. In *2013 International conference on machine learning and cybernetics*, volume 4, pages 1678–1683. IEEE, 2013.

[CVD+20] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. The tangled genealogy of IoT malware. In *Annual Computer Security Applications Conference*, pages 1–16, 2020.

[DA18] Jonas Depoix and Philipp Altmeyer. Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning. page 11, 2018.

[DLL+20] Fei Ding, Hongda Li, Feng Luo, Hongxin Hu, Long Cheng, Hai Xiao, and Rong Ge. DeepPower: Non-intrusive and Deep Learning-based Detection of IoT Malware Using Power Side Channels. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 33–46, 2020.

[DMS+13] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559–570, 2013.

[Dom21] Christopher Domas. The Movfuscator. https://github.com/xoreaxeaxeax/movfuscator, 2015 (accessed 30-August-2021).

[DWA+19] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2019.

[Dwy19] Andrew Carl Dwyer. *Malware ecologies: a politics of cybersecurity*. PhD thesis, University of Oxford, 2019.

[EH11] Mojtaba Eskandari and Sattar Hashemi. Metamorphic malware detection using control flow graph mining. *Int. J. Comput. Sci. Network Secur*, 11(12):1–6, 2011.

[EN20] Chris Eagle and Kara Nance. *The Ghidra Book: The Definitive Guide*. no starch press, 2020.

[Ent18] Symantec Enterprise. Internet Security Threat Report 2018. *Mountain View, CA, USA*, 2018.

[ESZ13] Shawn Embleton, Sherri Sparks, and Cliff C Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[Fer07] Peter Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 55, 2007.

[FI17] Ibraheem Frieslaar and Barry Irwin. Recovering AES-128 encryption keys from a Raspberry Pi. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, pages 228–235, 2017.

[FI18] I Frieslaar and B Irwin. Developing an electromagnetic noise generator to protect a Raspberry Pi from side channel analysis. *SAIEE Africa Research Journal*, 109(2):85–101, 2018.

[FMC11] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[GAD21] GREAT SCOTT GADGETS. GREAT SCOTT GADGETS HackRF One. https://greatscottgadgets.com/hackrf/one/, 2021. Accessed: 2022-01-01.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[Gér19] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[GLP06] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. Stochastic Methods. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2006.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *International workshop on cryptographic hardware and embedded systems*, pages 251–261. Springer, 2001.

[GNU21] GNUradio. GNU Radio. https://www.gnuradio.org/about/, 2021. Accessed: 2022-01-01.

[GPPT15] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing Keys from PCs using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. Technical Report 170, 2015. Accessed on 2020-01-13.

[GPSE15] Mordechai Guri, Yuri Poliak, Bracha Shapira, and Yuval Elovici. JoKER: Trusted detection of kernel rootkits in android devices via JTAG interface. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 65–73. IEEE, 2015.

[GPT15] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. *Journal of Cryptographic Engineering*, 5(2):95–112, 2015.

[Han20] Seunghun Han. Adore-NG v2.5. https://github.com/kkamagui/adore-ng, 2020. Accessed: 2022-02-10.

[HHM+14] Naofumi Homma, Yu-ichi Hayashi, Noriyuki Miura, Daisuke Fujimoto, Daichi Tanaka, Makoto Nagata, and Takafumi Aoki. Em attack is non-invasive? - design methodology and validity verification of em attack sensor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 1–16, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[HL07] David Harley and Andrew Lee. The root of all evil?-rootkits revealed, 2007.

[HLI13] KyoungSoo Han, Jae Hyun Lim, and Eul Gyu Im. Malware analysis method using visualization of binary files. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pages 317–321. 2013.

[HP18] Seunghun Han and JH Park. Shadow-box v2: The practical and omnipotent sandbox for arm. *2018, slideshow at Blackhat Asia*, 2018.

[HR15] Trammell Hudson and Larry Rudolph. Thunderstrike: EFI firmware bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–10, 2015.

[HZ12] Annelie Heuser and Michael Zohner. Intelligent machine homicide - breaking cryptographic devices using support vector machines. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, volume 7275 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2012.

[Jan21] Arun Prakash Jana. spy v1.8. https://github.com/jarun/spy, 2021. Accessed: 2022-02-10.

[JCO20] Blake Janes, Heather Crawford, and TJ OConnor. Never Ending Story: Authentication and Access Control Design Flaws in Shared IoT Devices. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 104–109, 2020.

[JKP94] George H. John, Ron Kohavi, and Karl Pfleger. Irrelevant Features and the Subset Selection Problem. In William W. Cohen and Haym Hirsh, editors, *Machine Learning, Proceedings of the Eleventh International Conference, Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994*, pages 121–129. Morgan Kaufmann, 1994.

[JLC20] Xingbin Jiang, Michele Lora, and Sudipta Chattopadhyay. Efficient and Trusted Detection of Rootkit in IoT Devices via Offline Profiling and Online Monitoring. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, pages 433–438, 2020.

[Joh85] P Johnson. New research lab leads to unique radio receiver. *E-Systems Team*, 5(4):6–7, 1985.

[JRWM15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[Jun20] Juho Junnila. (Thesis) Effectiveness of Linux Rootkit Detection Tools. 2020.

[JWHT14] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[KJJR11] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[KK17] Itzik Kotler and Amit Klein. THE ADVENTURES OF AV AND THE LEAKY SANDBOX. *Black Hat USA Briefings*, 2017.

[KKB+21] Eunbyeol Ko, Jinsung Kim, Younghoon Ban, Haehyun Cho, and Jeong Hyun Yi. ACAMA: Deep Learning-Based Detection and Classification of Android Malware Using API-Based Features. *Security and Communication Networks*, 2021, 2021.

[KKV11] Stefan Katzenbeisser, Johannes Kinder, and Helmut Veith. *Malware Detection*, pages 752–755. Springer US, Boston, MA, 2011.

[KM06] J Zico Kolter and Marcus A Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(Dec):2721–2744, 2006.

[Koc96] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[KRM+18] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil DB Bruce, Yang Wang, and Farkhund Iqbal. Malware classification with deep convolutional neural networks. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018.

[KSN+19a] H. A. Khan, N. Sehatbakhsh, L. N. Nguyen, R. L. Callan, A. Yeredor, M. Prvulovic, and A. Zajic. IDEA: Intrusion Detection through

Electromagnetic-Signal Analysis for Critical Embedded and Cyber-Physical Systems. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2019.

[KSN+19b] Haider A. Khan, Nader Sehatbakhsh, Luong N. Nguyen, Milos Prvulovic, and Alenka G. Zajic. Malware detection in embedded systems using neural network model for electromagnetic side-channel signals. *J. Hardware and Systems Security*, 3(4):305–318, 2019.

[KVH+20] Raphaël Khoury, Benjamin Vignau, Sylvain Hallé, Abdelwahab Hamou-Lhadj, and Asma Razgallah. An analysis of the use of CVEs by IoT malware. In *International Symposium on Foundations and Practice of Security*, pages 47–62. Springer, 2020.

[KY13] Deguang Kong and Guanhua Yan. Discriminant Malware Distance Learning on Structural Information for Automated Malware Classification. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, page 1357–1365, New York, NY, USA, 2013. Association for Computing Machinery.

[KZLR12] Iain Kyte, Pavol Zavarsky, Dale Lindskog, and Ron Ruhl. Enhanced side-channel analysis method to detect hardware virtualization based rootkits. In *World Congress on Internet Security (WorldCIS-2012)*, pages 192–201, 2012.

[KZWE16] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[Lab21] Nozomi Networks Labs. How IOT botnets evade detection and analysis. https://www.nozominetworks.com/blog/how-iot-botnets-evade-detection-and-analysis/, Mar 2022 (accessed 30-July-2021).

[LAS15] Jared Lee, Thomas H Austin, and Mark Stamp. Compression-based analysis of metamorphic malware. *International Journal of Security and Networks*, 10(2):124–136, 2015.

[LBMNS18] Quan Le, Oisín Boydell, Brian Mac Namee, and Mark Scanlon. Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, 26:S118–S126, 2018.

[LGO04] John Levine, Julian Grizzard, and Henry Owen. A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table. In *Second IEEE International Information Assurance Workshop, 2004. Proceedings.*, pages 107–125. IEEE, 2004.

[Lin11] Martina Lindorfer. Thesis: Detecting Environment-Sensitive Malware. http://martina.lindorfer.in/files/papers/disarm_thesis.pdf, 2011.

[LKMC11] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection*, pages 338–357. Springer, 2011.

[LMG+18] Patrick Luckett, J Todd McDonald, William B Glisson, Ryan Benton, Joel Dawson, and Blair A Doyle. Identifying stealth malware using CPU power consumption and learning algorithms. *Journal of Computer Security*, 26(5):589–613, 2018.

[LWYZ17] Liu Liu, Bao-sheng Wang, Bo Yu, and Qiu-xi Zhong. Automatic malware classification and new malware detection using machine learning. *Frontiers of Information Technology & Electronic Engineering*, 18(9):1336–1347, 2017.

[LXF+22] Yang Liu, Zisen Xu, Ming Fan, Yu Hao, Kai Chen, Hao Chen, Yan Cai, Zijiang Yang, and Ting Liu. ConcSpectre: Be Aware of Forthcoming Malware Hidden in Concurrent Programs. *IEEE Transactions on Reliability*, 2022.

[m0h15] m0hamed. lkm-rootkit: A rootkit implemented as a linux kernel module. https://github.com/m0hamed/lkm-rootkit, 2015. Accessed: 2022-02-10.

[m0n21] m0nad. Diamorphine: a LKM rootkit. https://github.com/m0nad/Diamorphine, 2021. Accessed: 2022-02-10.

[MBBB16] Rauf Mahmudlu, Valentina Banciu, Lejla Batina, and Ileana Buhan. LDA-based clustering as a side-channel distinguisher. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 62–75. Springer, 2016.

[MBM+18] Yair Meidan, Michael Bohadana, Yael Mathov, Yisroel Mirsky, Asaf Shabtai, Dominik Breitenbacher, and Yuval Elovici. N-BaIoT—Network-Based Detection of IoT Botnet Attacks Using Deep Autoencoders. *IEEE Pervasive Computing*, 17(3):12–22, July 2018. Conference Name: IEEE Pervasive Computing.

[MCHR22] J Todd McDonald, Rebecca C Clark, Lee M Hively, and Samuel H Russ. Phase space power analysis for PC-based rootkit detection. In *Proceedings of the 2022 ACM Southeast Conference*, pages 82–90, 2022.

[McN15] Ciarán McNally. maK_it-Linux-Rootkit. https://web.archive.org/web/20190119045332/https://r00tkit.me/, 2015. Accessed: 2022-02-10.

[MD11] M Narasimha Murty and V Susheela Devi. *Pattern recognition: An algorithmic approach*. Springer Science & Business Media, 2011.

[mem19] mempodippy. vlany: a Linux LD_PRELOAD rootkit. https://github.com/mempodippy/vlany, 2019. Accessed: 2022-02-10.

[MHN+13] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Citeseer, 2013.

[Mit93] Joseph Mitola. Software radios: Survey, critical evaluation and future directions. *IEEE Aerospace and Electronic Systems Magazine*, 8(4):25–36, 1993.

[Mit97] Tom M Mitchell. Machine learning, 1997.

[MKK07a] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[MKK07b] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MSB22] Sanjay Madan, Sanjeev Sofat, and Divya Bansal. Tools and Techniques for Collection and Analysis of Internet-of-Things Malware: A Systematic State-of-Art Review. 2022. Accessed on 2022-02-08.

[MSJ01] Nelson Murilo and Klaus Steding-Jessen. Métodos para detecção local de rootkits e módulos de kernel maliciosos em sistemas UNIX. In *Anais do III Simpósio sobre Segurança em Informática (SSI'2001)*, pages 133–139, 2001.

[NAC82] NSA NACSIM. 5000: Tempest Fundamentals. *National Security Agency*, 1982.

[NDGJ21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM. *IACR Cryptol. ePrint Arch.*, 2021:79, 2021.

[NKJM11] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware Images: Visualization and Automatic Classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, New York, NY, USA, 2011. Association for Computing Machinery.

[NSA+17] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic. EDDIE: EM-based detection of deviations in program execution. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 333–346, 2017.

[ODG+15] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 651–661, Burlingame, CA, USA, February 2015. IEEE. Accessed on 2019-07-02.

[OGC10] Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo. Bait your hook: a novel detection technique for keyloggers. In *International workshop on recent advances in intrusion detection*, pages 198–217. Springer, 2010.

[OMR] Markus F.X.J. Oberhumer, László Molnár, and John F. Reiser. UPX the Ultimate Packer for eXecutables. https://upx.github.io/. Accessed on 2020-05-14.

[OSM11] P. O'Kane, S. Sezer, and K. McLaughlin. Obfuscation: The Hidden Malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[otCTO21] Mayor's Office of the Chief Technology Officer. IoT Strategy The New York City Internet of Things Strategy. http://nyc.gov/cto/, 2021. Accessed: 2022-02-01.

[Pag02] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *Cryptology ePrint Archive*, 2002.

[PHG17] Stjepan Picek, Annelie Heuser, and Sylvain Guilley. Template attack versus Bayes classifier. *J. Cryptogr. Eng.*, 7(4):343–351, 2017.

[PJFMA04] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX security symposium*, pages 179–194. San Diego, USA, 2004.

[PMH21] Duy-Phuc Pham, Damien Marion, and Annelie Heuser. Poster: Obfuscation Revealed-Using Electromagnetic Emanation to Identify and Classify Malware. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 710–712. IEEE, 2021.

[PMH22] Duy-Phuc Pham, Damien Marion, and Annelie Heuser. ULTRA: Ultimate Rootkit Detection over the Air. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2022.

[PMMH21] Duy-Phuc Pham, Damien Marion, Mathieu Mastio, and Annelie Heuser. Obfuscation Revealed: Leveraging Electromagnetic Signals for Obfuscated Malware Classification. In *Annual Computer Security Applications Conference (ACSAC)*, 2021.

[pts21] ptsecurity. Rootkits: evolution and detection methods. https://www.ptsecurity.com/ww-en/analytics/rootkits-evolution-and-detection-methods/, 2021. Accessed: 2022-01-10.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[PVM19] Duy-Phuc Pham, Duc-Ly Vu, and Fabio Massacci. Mac-A-Mal: macOS malware analysis framework resistant to anti evasion techniques. *Journal of Computer Virology and Hacking Techniques*, 15(4):249–257, 2019.

[PZCW16] Milos Prvulovic, Alenka Zajić, Robert L Callan, and Christopher J Wang. A method for finding frequency-modulated and amplitude-modulated electromagnetic emanations in computer systems. *IEEE Transactions on Electromagnetic Compatibility*, 59(1):34–42, 2016.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *International Conference on Research in Smart Cards*, pages 200–210. Springer, 2001.

[RGF+19] R. A. Riley, J. T. Graham, R. M. Fuller, R. O. Baldwin, and A. Fisher. A New Way to Detect Cyberattacks: Extracting Changes in Register Values From Radio-Frequency Side Channels. *IEEE Signal Processing Magazine*, 36(2):49–58, March 2019.

[RHK20] Khaled Riad, Teng Huang, and Lishan Ke. A dynamic and hierarchical access control for IoT in multi-authority cloud storage. *Journal of Network and Computer Applications*, 160:102633, 2020.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[RKK07] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo,

and René Peralta, editors, *Information Security*, pages 1–18, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Rou16] Thibaut Rouffineau. Research: Consumers are terrible at updating their connected devices. https://ubuntu.com/blog/research-consumers-are-terrible-at-updating-their-connected-devices, Dec 2016.

[RSSHCB21] Valerian Rey, Pedro Miguel Sánchez Sánchez, Alberto Huertas Celdrán, and Gérôme Bovet. Federated Learning for Malware Detection in IoT Devices. 204:108693, 2021. Accessed on 2022-01-14.

[Rut06] Joanna Rutkowska. Introducing blue pill. *The official blog of the invisiblethings. org*, 22:23, 2006.

[Rut09] Joanna Rutkowska. The sky is falling? http://theinvisiblethings.blogspot.com/2009/03/sky-is-falling.html, Mar 2009.

[RZC+18] Edward Raff, Richard Zak, Russell Cox, Jared Sylvester, Paul Yacci, Rebecca Ward, Anna Tracy, Mark McLean, and Charles Nicholas. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, 14(1):1–20, 2018.

[SEE+17] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 483–493, New York, NY, USA, 2017. ACM. Accessed on 2019-03-26.

[SEZS01] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, pages 38–49, May 2001. ISSN: 1081-6011.

[Sha16] Udi Shamir. Analyzing a New Variant of BlackEnergy 3, Likely Insider-Based Execution. *SentinelOne, Mountain View, California*, 288, 2016.

[SLKS19a] Asanka Sayakkara, Nhien-An Le-Khac, and Mark Scanlon. A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics. *Digital Investigation*, 29:43–54, 2019.

[SLKS19b] Asanka Sayakkara, Nhien-An Le-Khac, and Mark Scanlon. Leveraging Electromagnetic Side-Channel Analysis for the Investigation of IoT Devices. *Digital Investigation*, 29:S94–S103, July 2019. Accessed on 2020-06-23.

[SM16] Tobias Schneider and Amir Moradi. Leakage assessment methodology - Extended version. *J. Cryptogr. Eng.*, 6(2):85–99, 2016.

[SN01] Matthew NO Sadiku and Sudarshan Nelatury. *Elements of electromagnetics*, volume 428. Oxford university press New York, 2001.

[SNA+20] N. Sehatbakhsh, A. Nazari, M. Alam, F. Werner, Y. Zhu, A. Zajic, and M. Prvulovic. REMOTE: Robust External Malware Detection Framework by Using Electromagnetic Signals. *IEEE Transactions on Computers*, 69(3):312–326, 2020.

[SNZP16] Nader Sehatbakhsh, Alireza Nazari, Alenka Zajic, and Milos Prvulovic. Spectral profiling: Observer-effect-free profiling by monitoring EM emanations. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–11. IEEE, 2016.

[Son22] SonicWall. 2022 Sonicwall cyber threat report. https://www.sonicwall.com/2022-cyber-threat-report/, 2022.

[SPP+18] Hossein Sayadi, Nisarg Patel, Sai Manoj P.D., Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Ensemble Learning for Effective Run-Time Hardware-Based Malware Detection: A Comprehensive Analysis and Classification. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018.

[Sut13] Scott Sutherland. 10 evil user tricks for bypassing anti-virus. https://blog.netspi.com/10-evil-user-tricks-for-bypassing-anti-virus/, 2013. Accessed on 2013.

[TIBV10] Ronghua Tian, Rafiqul Islam, Lynn Batten, and Steve Versteeg. Differentiating malware from cleanware using behavioural analysis. In *2010 5th international conference on malicious and unwanted software*, pages 23–30. Ieee, 2010.

[TK10] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers*, 27(1):10–25, 2010.

[Tou16] Fred Touchette. The evolution of malware. *Network Security*, 2016(1):11–14, 2016.

[UT17] Unix-Thrust. Unix-Thrust/Beurk: Beurk Experimental unix rootkit. https://github.com/unix-thrust/beurk, 2017. Accessed: 2022-02-10.

[Val08] Danilo Valerio. Open source software-defined radio: A survey on gnuradio and its applications. *Forschungszentrum Telekommunikation Wien, Vienna, Technical Report FTW-TR-2008-002*, 2008.

[VE85] Wim Van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4(4):269–286, 1985.

[WF12] Carsten Willems and Felix C. Freiling. Reverse Code Engineering – State of the Art and Countermeasures. 54(2):53–63, 20-03-2012.

[WFL+15] Markus Wagner, Fabian Fischer, Robert Luh, Andrea Haberson, Alexander Rind, Daniel A Keim, and Wolfgang Aigner. A Survey of Visualization Systems for Malware Analysis. 2015.

[WJCN09] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554, 2009.

[WK13] Xueyang Wang and Ramesh Karri. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–7. IEEE, 2013.

[WZH+18] Xiao Wang, Quan Zhou, Jacob Harer, Gavin Brown, Shangran Qiu, Zhi Dou, John Wang, Alan Hinton, Carlos Aguayo Gonzalez, and Peter Chin. Deep learning-based classification and anomaly detection of side-channel signals. In *Cyber Sensing 2018*, volume 10630, page 1063006. International Society for Optics and Photonics, 2018.

[YIT+16] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. SandPrint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 165–187. Springer, 2016.

[YSI17] Pavel Yosifovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. Microsoft Press, 2017.

[Zab18] Adam Zabrocki. Linux Kernel Runtime Guard (LKRG) under the Hood. In *CONFidence Conference*, 2018.

[ZF05] YongBin Zhou and DengGuo Feng. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. Technical Report 388, 2005. Accessed on 2022-02-24.

[ZGJ+18] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 457–468, 2018.

[ZP14] Alenka Zajić and Milos Prvulovic. Experimental demonstration of electromagnetic information leakage from modern processor-memory systems. *IEEE Transactions on Electromagnetic Compatibility*, 56(4):885–893, 2014.

[ZPKA18] Alenka Zajic, Milos Prvulovic, Haider Adnan Khan, and Monjur Alam. Detailed tracking of program control flow using analog side-channel signals: a promise for IoT malware detection and a threat for many cryptographic implementations. In Peter Chin and Igor V. Ternovskiy, editors,

Cyber Sensing 2018, page 5, Orlando, United States, May 2018. SPIE. Accessed on 2019-03-20.

This page intentionally left blank

A Appendix

Malware obfuscation classification technical details

Experimental results on the meaning of test traces

Figure A.1 shows accuracy of NB and SVM when calculating the mean over t execution measurements from the same binary in the test dataset. We see an improvement in all scenarios. Therefore, when the number of executions/ measurements per unknown binary is not a restricting factor for the malware analyst, then computing the mean over t traces will result in a more accurate prediction. This meaning process is usually in the side-channel domain, as in [CDD+14]. Interestingly, we could not observe a straightforward improvement when applying this technique to MLP and CNN classifications. One reason could be that the random user environment changes for each execution, thus even though the binary stays unaltered, the measurement trace changes. Now when calculating the mean, the patterns of features that may help MLP and CNN to make correct predictions may be mixed or changed. Contrary. NB and even SVM may not be able to model these patterns due to their more simplistic nature and computing the mean can be seen as noise reduction instead.

Table A.1 – AHMA: Malware tag map
The first column lists all malware and benign samples, followed by the number of recorded traces. Then each column refers to a scenario and gives for each sample the group it belongs to if it has been used. [*] (resp., [+]) means the sample has been used only during the training phase (resp. the testing phase), by default samples are used during both phases (80% for training, 20% for testing).

Bin. names	#	Types tags	Family tags	Virt. tags	Packer tags	Obf. tags	Exec. tags	Novelty tags
random34	6000	benign	benign				random34	benign
mirai.arm7	6000	ddos	mirai	orig	not_packed		mirai	mirai [*]
mirai_addopaque	3000	ddos	mirai			addopaque	mirai_addopaque	mirai [*]
mirai_virtualize	3000	ddos	mirai	virtualized		virtualize	mirai_virtualize	mirai [+]
mirai_flatten	3000	ddos	mirai			flatten	mirai_flatten	mirai [+]
mirai-bcf	3000	ddos	mirai			bcf	mirai-bcf	mirai [*]
mirai-cfflatten	3000	ddos	mirai			cfflatten	mirai-cfflatten	mirai [+]
mirai-sub	3000	ddos	mirai			sub	mirai-sub	mirai [+]
upx-mirai	3000	ddos	mirai		packed	upx	mirai-upx	mirai [*]
gonnacry	6000	ransomware	gonnacry	orig	not_packed		gonnacry	gonnacry [*]
upx-gonnacry	3000	ransomware	gonnacry		packed	upx	gonnacry-upx	gonnacry [*]
aes-upx-gonacry	3000	ransomware	gonnacry		packed	upx	gonnacry-aes-upx	gonnacry [+]
aes-gonacry	3000	ransomware	gonnacry		not_packed		gonnacry-aes	gonnacry [+]
des-gonnacry	3000	ransomware	gonnacry		not_packed		gonnacry-des	
des-upx-gonnacry	3000	ransomware	gonnacry		packed		gonnacry-des-upx	
gonnacry_Virtualize2	3000	ransomware	gonnacry	virtualized		virtualize	gonnacry_virtualize2	gonnacry [*]
gonnacry_flatten	3000	ransomware	gonnacry			flatten	gonnacry_flatten	gonnacry [*]
gonnacry_bcf	3000	ransomware	gonnacry			bcf	gonnacry_bcf	gonnacry [*]
gonnacry_sub	3000	ransomware	gonnacry			sub	gonnacry_sub	gonnacry [*]
gonnacry_cfflatten	3000	ransomware	gonnacry			cfflatten	gonnacry_cfflatten	gonnacry [+]
gonnacry_addopaque	3000	ransomware	gonnacry			addopaque	gonnacry_addopaque	gonnacry [*]
maK_it4.19.57-v7+.ko	3000	rootkit	maK_it				rootkit_maK_it	rootkit [*]
spy-4.19.57-v7+.ko	3000	rootkit	spy				rootkit_spy	rootkit [+]
bashlite	3000	ddos	bashlite	orig	not_packed		bashlite	bashlite [*]
bashlite_bcf	3000	ddos	bashlite			bcf	bashlite_bcf	bashlite [*]
bashlite_flatten	3000	ddos	bashlite			flatten	bashlite_flatten	bashlite [+]
bashlite_upx	3000	ddos	bashlite		packed	upx	bashlite_upx	bashlite [*]
bashlite_addopaque	3000	ddos	bashlite			addopaque	bashlite_addopaque	bashlite [*]
bashlite_cfflatten	3000	ddos	bashlite			cfflatten	bashlite_cfflatten	bashlite [*]
bashlite_sub	3000	ddos	bashlite			sub	bashlite_sub	bashlite [*]
bashlite_virtualize	3000	ddos	bashlite	virtualized		virtualize	bashlite_virtualize	bashlite [+]
playaudio	1000	benign	benign				playaudio	benign
recordcamera	1000	benign	benign				recordcamera	benign
takepicture	1000	benign	benign				takepicture	benign
encodevideo	1000	benign	benign				encodevideo	benign

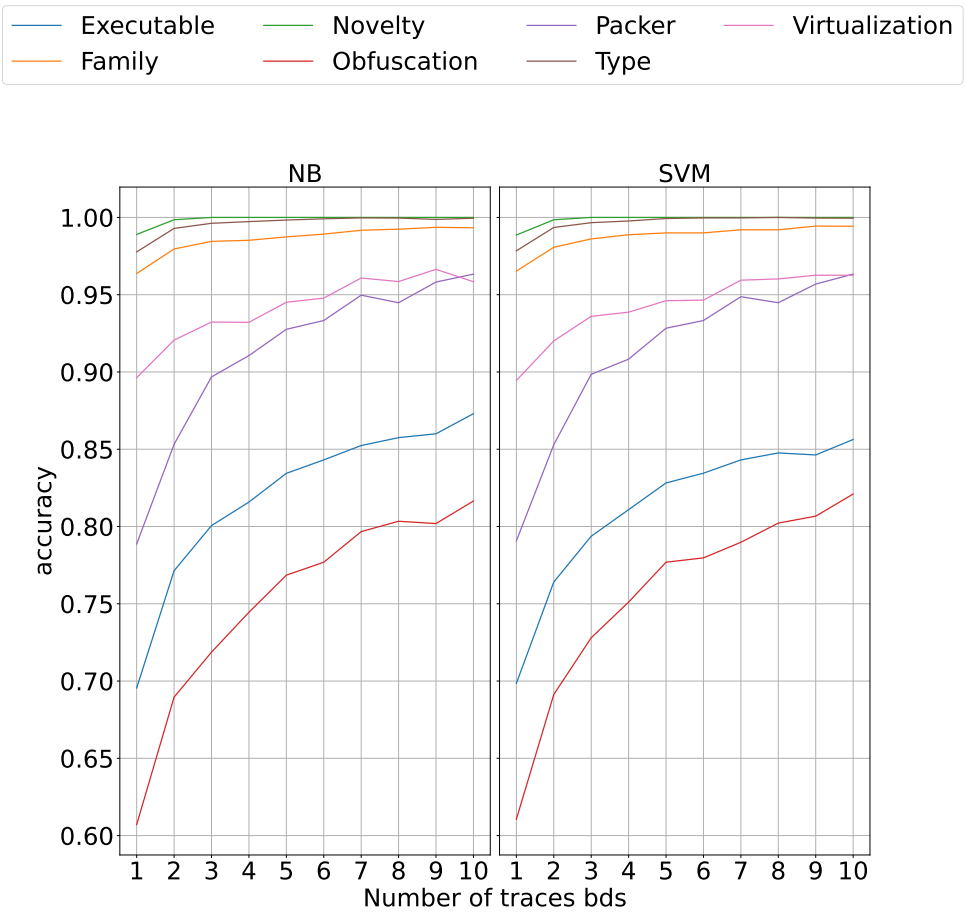


Figure A.1 – Accuracy when computing the mean over $t = [1, 2, 3, \ldots, 10]$ samples per binary in the test dataset. One can observe a drastic performance improvement for SVM and NB for the scenarios where the accuracy was lower to begin with. For type and novelty classification the accuracy reaches 100%.

ULTRA: Rootkit detection framework additional materials

Script example of network bait

Code A.1 – Network TCP bait script

```
for i in {1..30};
do (cat /proc/net/tcp > /dev/null)
done
```

Tuned iteration configuration values (c)

Table A.2 – Tuned iteration configuration values (c) for bait corresponding with the targeted devices.

	Devices δ	
Baits β_i	Raspberry	Ci20
getdents	5000	3000
readir	5000	5000
open	6000	6000
kill	400000	400000
read	225000	200000
write	350000	300000
stat	70000	70000
renameat	50000	50000
tcp	30	30
emu	5	5

Table A.3 – ULTRA's bill of materials

Equipment	Rate/Unit	Count	Amount (Euro)
HackRF One SDR	309	1	309
Adapter SMA Male BNC Female RG316	5	1	5
Amplifier Langer PA-303 BNC	375	1	375
*Probe Langer RF-U 5-2**	250	1	250
Total			939

* *This can be omitted in the case of using a hand-crafted probe.*

Patch snippet for static string obfuscation

Code A.2 – Patch diff between original and obfuscated rootkit to evade static signature

```
--- m0hamed/rootkit.c
+++ m0hamed-obed/cm9vdGtpdAo.c
-void hide_module(void) {
+void aGlkZV9tb2R1bGUK(void) {
       /*snippet*/
       -asmlinkage int hacked_getdents(unsigned int fd, struct
          linux_dirent *dirp, unsigned int count)
       +asmlinkage int aGFja2VkX2dldGRlbnRzCg(unsigned int fd, struct
          linux_dirent *dirp, unsigned int count)
       /*snippet*/
       - syscall_table[__NR_getdents] = hacked_getdents;
       + syscall_table[__NR_getdents] = aGFja2VkX2dldGRlbnRzCg;
       /*snippet*/
       - hide_module();
       + aGlkZV9tb2R1bGUK();
       /*snippet*/
```

Table A.4 – Classification scenario distinguishing kernel-space and user-space rootkits in (S_0), then in (S_1) we add benign samples.

		MLP	KPCA + NB	KPCA + SVM
Scenario		**AC** $_{[\epsilon_{opt}]}$ **PR**/**RC**	**AC** $_{[\epsilon_{opt}]}$ **PR**/**RC**	**AC** $_{[\epsilon_{opt}]}$ **PR**/**RC**
δ_{cl20}	S_0	$98.5_{[24]}$ /	$82.8_{[60]}$ $^{83.0}/_{82.8}$	$97.6_{[95]}$ $^{97.6}/_{97.6}$
	S_1	$98.1_{[28]}$ /	$75.9_{[95]}$ $^{76.0}/_{75.9}$	$96.5_{[25]}$ $^{96.5}/_{96.5}$
$\delta_{rasp.}$	S_0	$90.5_{[28]}$ /	$74.6_{[15]}$ $^{74.9}/_{74.6}$	$89.3_{[100]}$ $^{89.3}/_{89.3}$
	S_1	$87.5_{[28]}$ /	$62.8_{[40]}$ $^{64.6}/_{62.8}$	$85.1_{[25]}$ $^{85.3}/_{85.1}$

Enhancement through meaning testing traces

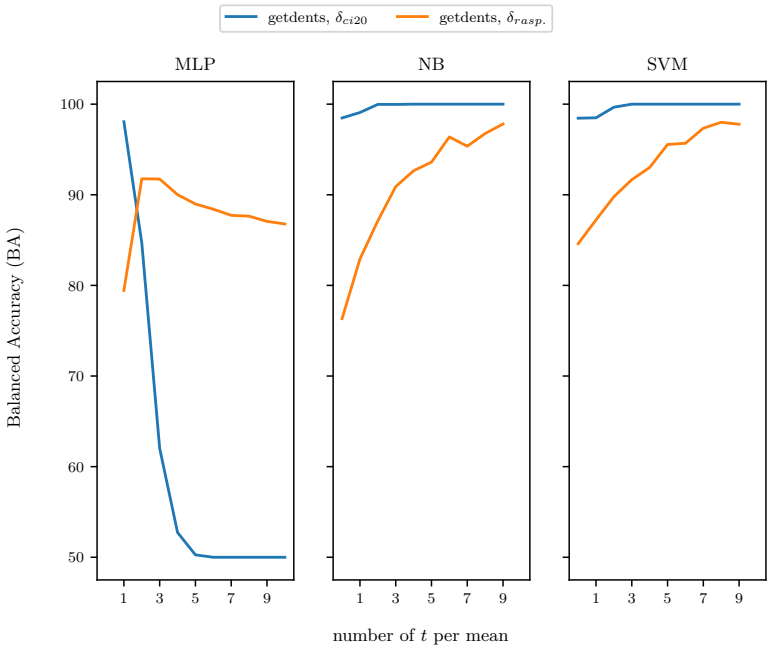


Figure A.2 – Balanced accuracy (BA) of the Table 4.5 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

Optimal bandwidth selection

Additional results

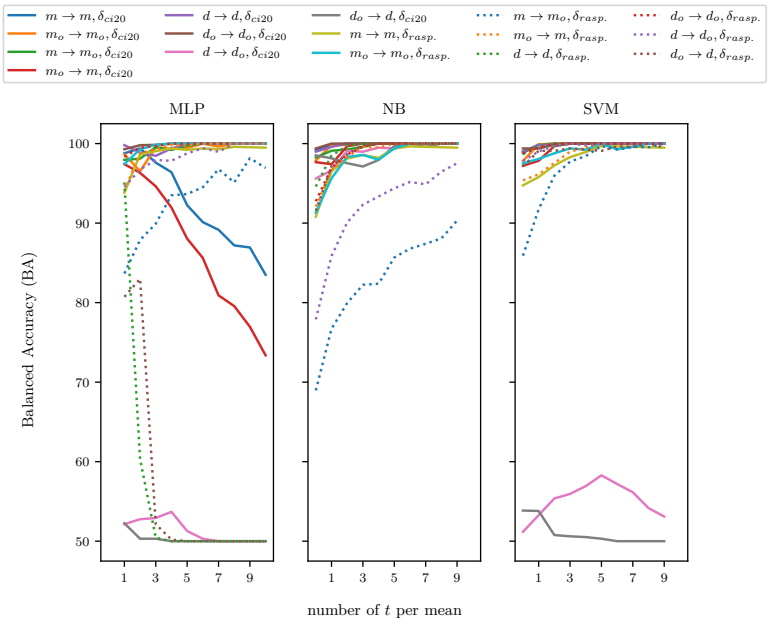


Figure A.3 – Balanced accuracy (BA) of the Table 4.7 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

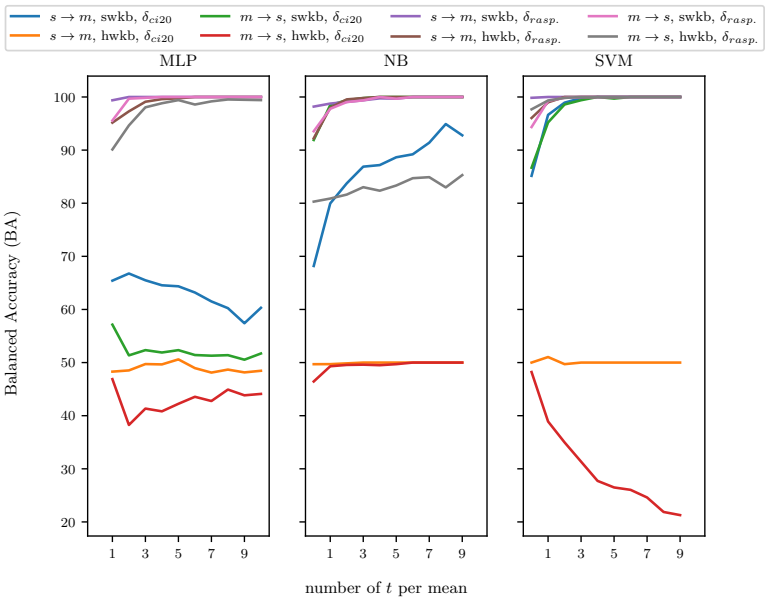


Figure A.4 – Balanced accuracy (BA) of the Table 4.8 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

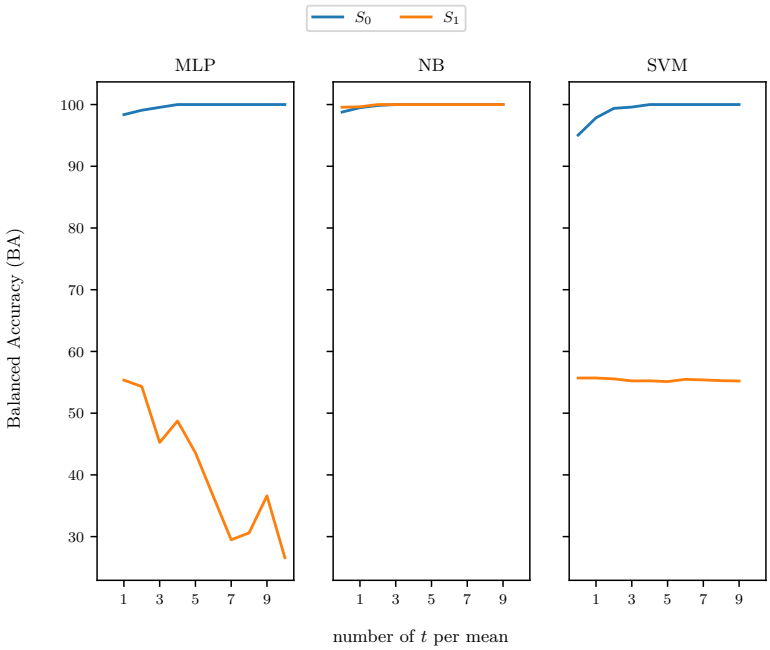


Figure A.5 – Balanced accuracy (BA) of the Table 4.9 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

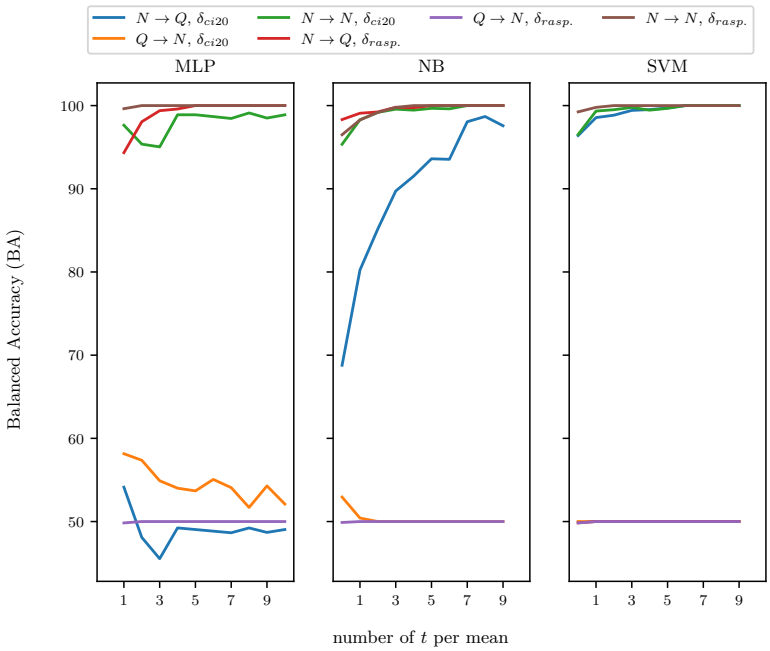


Figure A.6 – Balanced accuracy (BA) of the Table 4.10 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

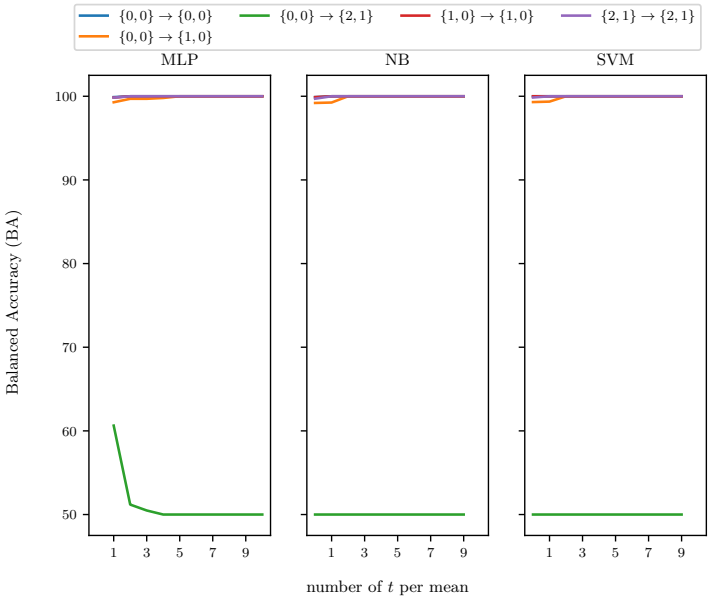


Figure A.7 – Balanced accuracy (BA) of the Table 4.12 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

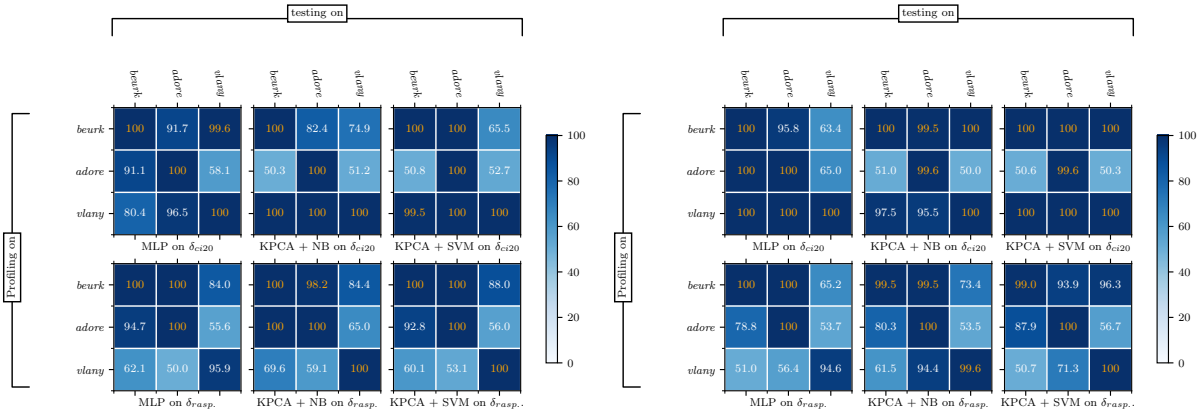
Algorithm 2 Bandwidth extraction procedure

nicv = NICV (learning_set) $\in \mathbb{R}^{F \times D}$ ▷ the learning set is composed of labed spectrogram of dimension $F \times D$
max_nicv $=$ max$_D$(nicv) $\in \mathbb{R}^{F}$
sorted_bandwidth = argsort (tmp)

last_added = 0 ▷ use as stopping criterion
current_acc = 0
current_bandwidth = sorted_bandwidth [0] ▷ start with the bandwidth with the highest NICV

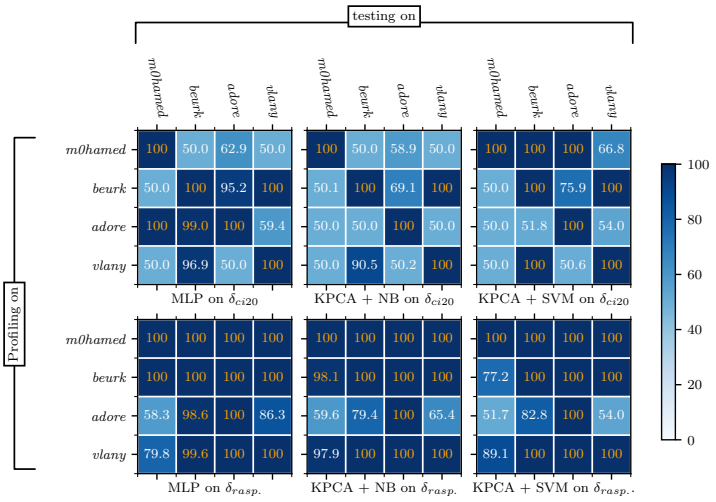
while last_added < len (sorted_bandwidth) **do**
    compute model of the m on current_bandwidth of the learning_set ▷ ML or DL learning phase
    tmp_res = eval (m, current_bandwidth of the validating_set) ▷ evaluate the model accuracy on the validating set
    if tmp_res > current_res **then**
        remove first element of sorted_bandwidth ▷ the last added bandwidth will be concerved in the optimal list
        last_added = 0 ▷ to test all remaining bandwidth with the current optimal list
        current_res = tmp_res
    else
        put the first element of sorted_bandwidth to its end
        last_added += 1
    end if
    add to the current_bandwidth the first sorted_bandwidth
end while

if len (sorted_bandwidth) > 0 **then**
    remove last added element of current_bandwidth ▷ the last added is not part of the optimal selection
end if
return current_bandwidth



(a) $\beta = \{tcp\}$

(b) $\beta = \{readir\}$



(c) $\beta = \{open\}$

Figure A.8 – Novelty rootkit detection. Same description as the Figure 4.5, but with differents baits β.

This page intentionally left blank

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Exploitation des signaux de canaux auxiliaires pour la classification des malwares IoT et la détection des rootkits.

Mot clés : Dispositifs IoT, systèmes embarqués, classification des malwares, détection des rootkits, emmanation électromagnétique, radio logicielle

Résumé : L'Internet des objets est constitué de périphériques dont le nombre et la complexité augmentent de manière exponentielle. Ils utilisent une variété de logiciels et de micrologiciels personnalisés sans tenir compte des problèmes de sécurité, ce qui en fait une cible attrayante pour les cybercriminels. La détection des malware reposant sur des caractéristiques statiques et dynamiques se heurte encore à diverses difficultés, telles que les techniques d'empaquetage ou d'obfuscation, ou la possibilité d'échapper à la surveillance sandbox. Contrairement aux systèmes informatiques et aux serveurs, les systèmes cyber-physiques embarqués peuvent manquer de ressources ou d'accessibilité pour les outils anti-malware. Nous présenterons des méthodes qui ne nécessitent pas d'altération du dispositif et qui peuvent être déployées de manière indépendante sans aucune surcharge en exploitant l'émanation électromagnétique par radio. Les contributions de cette thèse de doctorat sont séparées en deux parties différentes. Premièrement, nous présentons une nouvelle approche qu'un analyste malware peut utiliser pour recueillir des informations exactes sur le type et l'identité des malwares, même en présence de techniques d'obfuscation qui peuvent entraver l'analyse. De plus, nous présentons le système ULTRA à faible coût, qui est la première solution de type "wave-and-play", où il suffit d'agiter une sonde sur le dispositif pour voir instantanément quel rootkit est infecté. ULTRA a une spécification qui facilite la découverte des rootkits dans un système en temps réel sans qu'il soit nécessaire de modifier le dispositif ou d'exiger des logiciels en surveillant deux types distincts d'appâts qui sont capables d'exposer le comportement des rootkits furtifs.

Title: Leveraging side-channel signals for IoT malware classification and rootkit detection

Keywords: Iot devices, embedded systems, malware classification, rootkit detection, Electromagnetic emmanation, software-defined radio

Abstract: The Internet of Things is constituted of devices that are exponentially growing in number and in complexity. They utilize a variety of customized software and firmware without consideration of security concerns, making them an appealing target for cybercriminals. Malware detection relying on static and dynamic features still have various difficulties such as packer or obfuscation techniques, or sandbox monitoring can be evaded. Unlike computer systems and servers, embedded cyber physical system may lack resources or accessibility for anti-malware tools. We will present methods that do not require device alteration while they can be deployed independently without any overhead by leveraging electromagnetic emanation over the air. The contributions of this PhD thesis are separated into two different parts. First, we present a novel approach that a malware analyst can use to gather exact information about the type and identity of malware, even in the presence of obfuscation techniques that may hinder analysis. Further we present low-cost ULTRA framework which is the first wave-and-play solution, where one can simply wave a probe over the device to instantly see what rootkit is infected. ULTRA has a specification that facilitates the discovery of rootkits in a system in real-time without the need for device alteration or software requirements by monitoring two distinct types of baits that are capable of exposing the behavior of stealthy rootkits.