ULTRA: Ultimate Rootkit Detection over the Air

Duy-Phuc Pham, Damien Marion, Annelie Heuser

ULTRA: Ultimate Rootkit Detection over the Air

Duy-Phuc Pham*
duy-phuc.pham@irisa.fr
Univ Rennes, CNRS, Inria, IRISA
Rennes, France

Damien Marion
damien.marion@irisa.fr
Univ Rennes, CNRS, Inria, IRISA
Rennes, France

Annelie Heuser
annelie.heuser@irisa.fr
Univ Rennes, CNRS, Inria, IRISA
Rennes, France

ABSTRACT

Rootkits are the most challenging malware threats against server and desktop systems. They are created by highly skilled actors and are deployed in advanced persistent threat attacks. Lately and even in the future, rootkits will become a real threat to billions of IoT devices. Existing malware detection techniques based on static or dynamic analysis face major shortcomings, which become more apparent when it is necessary to detect threats on IoT devices.

In this paper, we propose the ULTRA framework, which can detect rootkits effectively and efficiently by operating outside of the "box" (literary device) with no resource requirement on the target device. ULTRA baits the rootkit to provoke activity, measures electromagnetic emanation with a software-defined radio, preprocesses signals, then detects and classifies rootkit behavior using machine/deep learning techniques. As use cases, we target two IoT devices with MIPS and ARM architectures. The proposed approach achieved promising results with high accuracy for detecting both known and unknown rootkits during the offline learning phase. Our experimental study involves classification of rootkit families and distinct variants, obfuscated rootkits, probe dislocation, benign noise (kernel) activities, and comparison with software-based solutions.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

rootkit detection; SDR; machine learning; deep learning; Electromagnetic; IoT devices

ACM Reference Format:

Duy-Phuc Pham, Damien Marion, and Annelie Heuser. 2022. ULTRA: Ultimate Rootkit Detection over the Air. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3545948.3545962

1 INTRODUCTION

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 [3] 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 [21]. Recently, NSA and FBI [5] reported Drovorub rootkit developed by state-sponsored APT28 to infect Linux systems to hide itself and files, directories, and network activities.

By 2025, we are expecting to have over 64 billion IoT devices [52] 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 [4, 51]. 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 [17], obfuscation techniques [50], and the fact that most IoT devices have limited resources or accessibility.

In this work, 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 device, making it particularly suitable in the context of IoT. Our approach consists of using hardware and software baits while only measuring electromagnetic (EM) emanation over the air with a (low-cost) software-defined radio. This setup prevents known malware evasion techniques as there is 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 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 probe (dis)location, evaluation of noisy environments, 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. In summary, our primary contributions are as follows:

*Current affiliation: duyphuc.pham@trellix.com, Trellix, CA, USA.

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545962

(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.
(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/anon-ultra/ultra.

2 RELATED WORK

On the effectiveness of Linux rootkit detection tools, Junnila J. [35] carried out an empirical evaluation of 5 prominent anti-rootkit tools: OSSEC [10], AIDE [38], rkhunter [8], chkrootkit [45] and Linux Kernel Runtime Guard (LKRG) [63] against 15 rootkits. Surprisingly, the results showed that only 37.3% of the detection tests provided any indication of infected systems. 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 [9]. 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 [26] 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.* [62] have presented a hypervisor-based system called HookSafe that monitors kernel hooks and prevents them from being hijacked by kernel rootkits. Shadow-box v2 [28] 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. However, a hypervisor-based solution is dependent on the operating system, architecture, and hardware support, which is not trivial to deploy per embedded device due to its limited power and resources. Rootkits living at the same protection level (hypervisor) and lower (*e.g.,* [12, 20, 31]) 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.* [9] 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 [60], Singh *et. al.* [57], and LKRDet [33] detect Linux rootkits by looking for HPC deviations during the execution of the kernel through virtualization to determine the presence of a rootkit. However, [12] 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 [18, 64] claim and experimentally support that using the micro-architecture information from HPCs cannot distinguish between benign and malware.

WattsUpDoc [16] was one of the earliest efforts in malware detection through hardware side-channel that demonstrated the measurement of power usage on medical embedded devices. Recently in [50, 56], the authors proposed to detect and classify malware by observing EM signals. This type of work only detects busy malware under its behavioral activity, however fails to detect stealthy rootkits after their installation. [37] described a network time analysis approach for monitoring performance changes caused by hardware virtualization, with the goal of detecting the hardware virtualization rootkit. [11, 39] identify rootkits by using power-based malware detection on general-purpose computers and [19, 39, 61] use machine learning (ML) and deep learning (DL) to perform a behavioral detection method based on CPU power consumption. Gibraltar[6] and Copilot [49] 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.

Most of the work 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 rootkit detection and classification. Finally, this work is the first and only to conduct research that levering

Table 1: Comparison with related works on kernel-level or user-level rootkit (user RK) detection using different side-channel analysis techniques: HPC, DMA, Power consumption (Power) and EM.

	Article	WnP	Classification	Baits	ML	DL	Sample size	Open source	Benign set	User RK	Detection latency	Targeted device(s)/Architecture
HPC	Numchecker [60]	-	-	✓	-	-	8	-	-	-	262.3ms	32-bit Linux PC
HPC	[57]	-	-	-	✓	-	5	-	-	-	45s	Windows 7 Intel (VMWare)
HPC	LKRDet[33]	-	-	✓	✓	-	4	✓	-	-	2.91s	ARM Cortex-A53 (TEE)
DMA	Copilot [49]	-	-	-	-	-	12	-	-	-	30s	PCI-compatible Intel PC Linux
DMA	Gibraltar [6]	-	-	-	-	-	23	-	✓	-	20s	PCI-compatible Intel PC Linux
Power	[39]	-	-	-	✓	✓	5	-	-	✓	>5m	PC Windows 10 & Ubuntu 14
Power	[11]	-	-	-	✓	-	5	-	-	-	>1m	Dell OptiPlex 755 Windows 7
EM	ULTRA	✓	✓	✓	✓	✓	9	✓	✓	✓	1.3s	ARM Raspberry Pi & MIPS Ci20

EM to perform real-time and real-world rootkit detection and classification using a software-defined radio (Table 1). ULTRA is the only wave (a probe) and play (WnP) solution, that one can simply wave over the device to instantly see what rootkit is infected, with a specification that facilitates the discovery of rootkits in a system in real-time without the need for device alteration or software requirements.

3 BACKGROUND

3.1 Rootkits

Rootkits exist to provide long-term covert access to a system, allowing it to be managed and monitored remotely while remaining undetected. In general, rootkits are classified into two forms: user mode and kernel mode rootkits [13, 29] according to the level (ring) of privileges obtained. Some rootkits are designed to perform both modes of operation and thus work at both levels. Furthermore, there exist rootkits living beyond the kernel level, such as hypervisor rootkits *(e.g.,* BluePill [53] on x86, and rHV [12] on ARM), system management mode rootkits [20], chipset rootkits (*e.g.,* Thunderstrike [31]). A rootkit is typically designed to conceal running modules and processes, mask the existence of files, directories, or users, hide network activities, and keystroke captures.

3.1.1 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, or preload loader to intercept calls to library functions and manipulate their execution. 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.

3.1.2 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. Therefore, kernel-level rootkits are often seen from strategic groups that have sufficient technical qualifications, such as APT groups that care for information theft or carry out destructive actions regardless of cost, or financial motivation [3].

3.2 Software defined radios (SDR)

Recent work to detect malware using EM [36, 50, 56] 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. It is a minimal hardware component with data processing and reconfigurability performed mostly in software, controlling the center frequency, bandwidth, 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[2], USRP and GNURadio [1, 59]). By leveraging EM signals from SDR, it is more suitable for malware detection than oscilloscopes since the EM measurement duration for malware typically requires a longer time window, compared to the general crypto side-channel attack. SDR is quickly becoming a promising candidate for EM side-channel research due to its capability to scan through a wide range of frequencies to locate potential EM leakages. According to recent research, SDR is an efficient solution for performing side channel attacks (on AES-128 [22], SHA1 [54]). Even attacks over short distance such as Screaming channels [15], which succeeded in breaking AES by exploiting wireless communication, are possible.

3.3 Dimension reduction and features extraction techniques

The output of the SDR is typically a large measurement trace which must be reduced for further processing. The extraction of useful information from a trace can be a difficult procedure. 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, the normalized inter-class variance (NICV) [7], SOST/SOSD [25], Pearson correlation coefficient [30, 42], TVLA [55]. The authors of [50] demonstrate how

to extract features from spectrograms using NICV to discover frequency bandwidths that may reveal behavioral information about the binaries. We propose an improvement by coupling the NICV with an hill climbing algorithm, as described in Section 5.3.1, to optimize the bandwidth extraction.

Principle component analysis (PCA) is a popular dimension reduction method that projects each data point onto only the first few principal components in order to obtain lower-dimensional data while preserving as much data variation as possible. The first principal component is chosen in the direction that maximizes the variance of the projected data. Each component is orthogonal to the previous one, and that again maximizes the variance of the projected data. Kernel PCA is an extension that replaces linear procedures in PCA with a kernel. In our proposed framework, we will use kernel PCA to further reduce the dimensionality of data for machine learning algorithms in our binary classification scenarios.

4 ULTRA:ULTIMATE ROOTKIT DETECTION OVER THE AIR FRAMEWORK

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. Fig. 1 illustrates ULTRA's workflow, which will be detailed within the following subsections. First, we define our threat model and methodology, and then explain how the EM emanation dataset is collected while a bait is executed on the target device with or without the presence of a rootkit. Thereafter, 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.

4.1 Threat model and methodology

Placing rootkit detection mechanism on the same level as the rootkit itself makes it obvious 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.

4.1.1 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.

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, etc.

In practice, the target-device is a blackbox in which the analyst has no prior knowledge of the rootkit's presence.

4.1.2 Notations and definitions.

Definition 4.1 (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 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 4.2 (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 4.3.

Definition 4.3 (Bait). A *bait* β, which is a software or hardware stimulus on a device δ, has the following requirements: *(i)* The bait can trigger partial or full behavior of rootkits without knowing *modus operandi* of the rootkit in advance; *(ii)* It has a variable duration time of execution activities that can be remotely controlled; *(iii)* 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 $p \times q$:

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

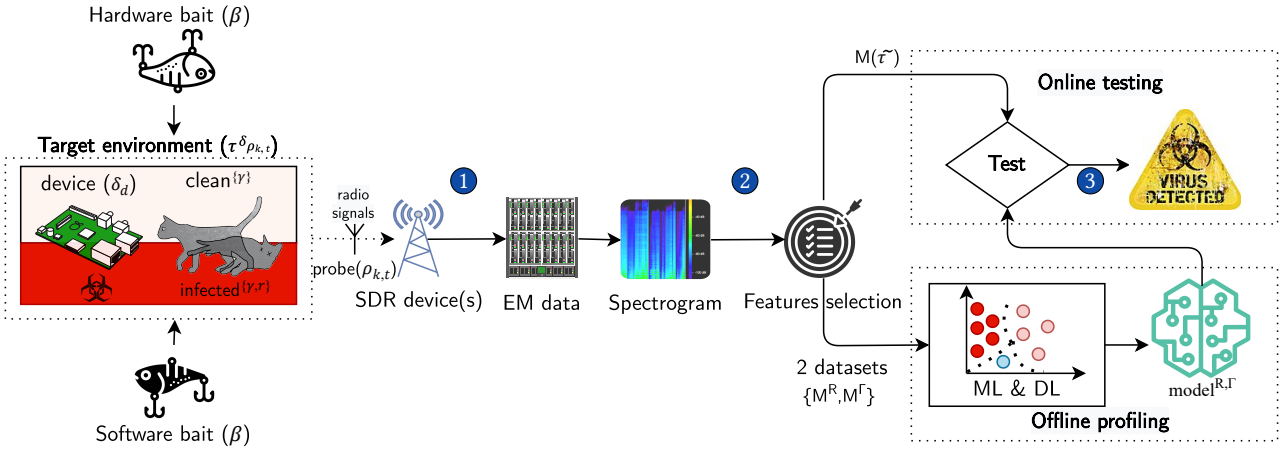


Figure 1: Illustration of ULTRA framework. (1) Trace acquisition: from black box target devices receiving software and hardware baits to EM measurements; (2) Pre-processing: converting raw measurements into usable data; (3) Malware detection and classification: from offline profiling models (trained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

4.1.3 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/deep learning) models that are able to detect if a device is infected with a rootkit or not, i.e.,

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

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 $\tau^{\delta}_{\rho_{k,t}}(\mathcal{B})$ 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 6.3.2), 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} (Raspberry Pi) and δ_{CI20} (Creator CI20) which are further detailed in the next subsections. Next, we consider the dislocation and changing the type of the EM probe. 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 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.

4.2.1 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 (see Table 4). 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* [58], *vlany* [44]), and kernel-level

rootkits (*diamorphine* [41], *m0ham3d* [40], *adore-ng* [27], *spy* [32], *maK_it* [43]).

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 7. As a consequence, two new obfuscated variants are included in the dataset that can easily evade signature-base detection solution (see Table 11). The total dataset of 9 rootkits was compiled on both architectures, thus using 2 different Linux kernel headers.

Table 2: 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_	lru_cache	bluetooth	atm
Firewalls	x_tables	ip_vs	br_netfilter	
Filesystem protocol modules	9pnet	9p	ecryptfs	nfsd
	btrfs	udf	xfs	cifs
	overlay			

4.2.2 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. Additionally, by collecting both Linux user space binaries and kernel space modules from MIPS and ARM architectures, 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 2). 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 of the previous studies on Table 1 considered the impact of benign activities.

4.3 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. 4.3.

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. Appendix 7 contains technical information about the bait mechanism. We present 2 novel strategies to trigger rootkits: software and hardware baits.

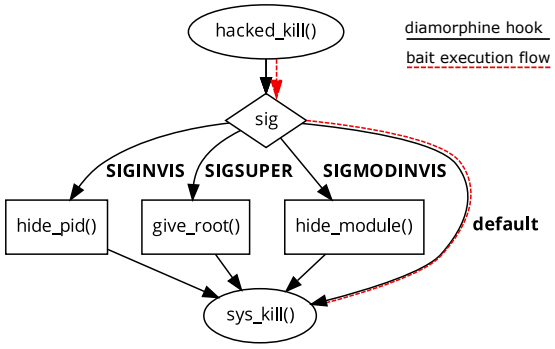


Figure 2: Illustration of execution flow for *kill* bait running under *diamorphine* which infected Linux kernel with hooked system call *kill().*

4.3.1 Software baits. Numchecker [60] used test programs that performed 5 system calls into the guest VM to determine the presence of a rootkit based on deviations of HPCs. In terms of keylogger detection, a similar tactic on the software level has been conducted by Ortolani *et. al.* [48] 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 this paper, 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).

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. 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. 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

Table 3: Target 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

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.

4.3.2 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.6 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. 4.3: remote controllability, fully triggering keylogger behaviors, and no difference from a standard keyboard. 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.

5 PRACTICAL USE CASE OF ULTRA

5.1 Target devices

Since this work focuses on rootkit detection on IoT devices due to their restricted resources, we have considered two widely embedded architectures for all experiments as specified in Table 3. They support broad activities for embedded and IoT scenarios regarding their prominent architectures, size, power consumption, and cost-effectiveness. Previous works [23, 61] show that cryptographic and anomaly activities can be distinguished by leveraging EM signals from the Raspberry Pi. However it is worth mentioning that no study has investigated the possibility of side-channel leakage on MIPS Creator CI20, so that there is no influence of prior knowledge on its experiment design. Thus, this work focuses on a 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.

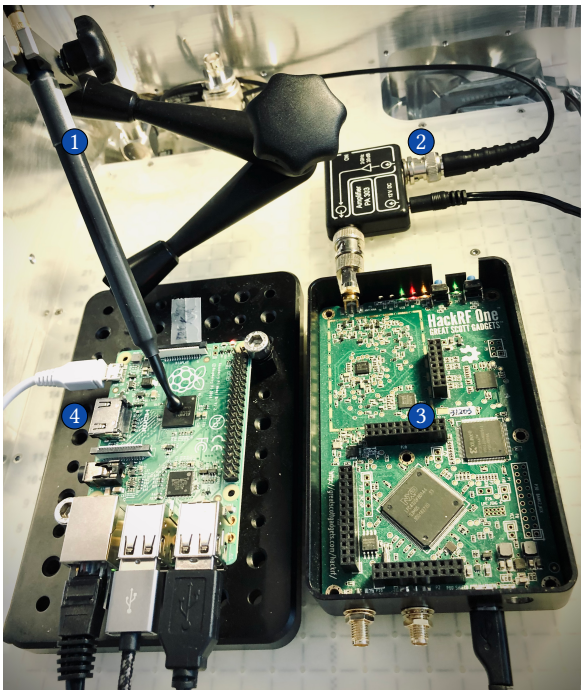


Figure 3: ULTRA framework data acquisition consists of a H-Field probe (1), which connected to an amplifier (2) and HackRF (3), placed 1mm and 45 degree above the target device Raspberry Pi (4) processor.

5.2 Data aquisition

The target devices are monitored in different setups ranging from low-cost to medium-cost (see bill of materials in Appendix table 15). 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 (Fig. 3). A low-cost setup with an EM-compatibility probe, which is made of ferrite and conductor and placed 10mm farther from the processor (Fig. 5) 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 probes and points of interest [47]. Our work does not claim to optimize the best combination of probes and their location, furthermore we will discuss the impact of probe (dis)locations in Section 6. Another challenge is to empirically find the targeted frequency to be monitored by the SDR device. First, we hypothesize the 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 high accuracy in testing, that corresponding centered frequency is assumed to leak information about the state

of the device. We achieved the best results for center frequencies (Fc) as given in Table 3.

We collected $q = 5000$ traces for each $m = 9$ rootkit variants in regard to their input baits, respectively (see Table 4) in both infected settings $\mathcal{M}^{\mathcal{R}}$ and 240 000 traces for benign traces $\mathcal{M}^{\Gamma}$. The same data acquisition process was carried out for both devices δ_{rasp} and δ_{ci20}. We recorded in total more than 800 000 raw traces, which took 6.6TB. Subsequently, we preprocessed the data as detailed in part 5.3.1.

5.3 Detection and classification framework

5.3.1 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 [46, 56], 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 [7] that we coupled with an hill climbing algorithm following a *forward selection* [34] 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.

5.3.2 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. For multi-class classification, we use the algorithms provided by [50] that include Naive Bayes (NB), Support Vector Machines (SVM), and Multi-layer Perceptions (MLP). In this work, we also focus on detection which is a two-class classification. For this, we exchange the activation function of the last layer of the MLP to comply with a two-class classification, i.e.,, we use *sigmoid* instead of *softmax*. The MLP architectures is shown in Appendix 14. Networks are 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.

6 RESULTS AND DISCUSSION

6.1 Scenarios and metrics

An individual dataset per device is measured for each of the 9 rootkits and their respective baits, as highlighted in Table 4. 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). 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.

Scenarios that use other baits (*e.g., network* tcp) can detect *adoreng*, *beurk*, *vlany*, or "Hide network port/socket rootkit" activities in general with high accuracy, are not detailed due to lack of space, but can be found in the Appendix (Figures 14a, 14b and 14c). We also investigate the classification between kernel-space and user-space rootkits, both with and without benign activities, results are available in the Appendix Table 16.

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 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 8, 9, 10, 11 and 12 in Appendix illustrate results using $[1, 2, \ldots, 10]$ averaged traces for all binary scenarios. One can see that if only one trace is available due to constraints on the response latency, high accuracy is already achievable.

6.2 Results

6.2.1 Rootkit detection. The 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 5 shows the balanced accuracy, true positive (TPR) and true negative rate (TNR), as well as the optimal number of selected bandwidth (see Subsection 5.3.1) 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.

Table 4: Input baits, that handled by system calls, network activities (*Net.*) 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	✓	✓	✓				✓	✓	✓		✓	✓		✓	

Table 5: 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 = $\delta_{\text{rasp.}}$, δ_{ci20}.

δ_{ci20}									$\delta_{\text{rasp.}}$								
MLP			**KPCA + NB**			**KPCA + SVM**			**MLP**			**KPCA + SVM**			**KPCA + SVM**		
BA $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**
$98.1_{[6]}$	97.6	98.6	$100_{[14]}$	100	100	$100_{[16]}$	100	100	$91.8_{[16]}$	91.0	92.5	$97.8_{[11]}$	96.7	98.9	$98.0_{[15]}$	100	96.0

Table 6: Classification by family and by activity obtained with MLP, LDA + NB and LDA + SVM. The column "#" gives the number of classes per scenario.

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

6.2.2 Rootkit classification. 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 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.

6.2.3 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.

Fig. 4 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 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, we see no significant differences

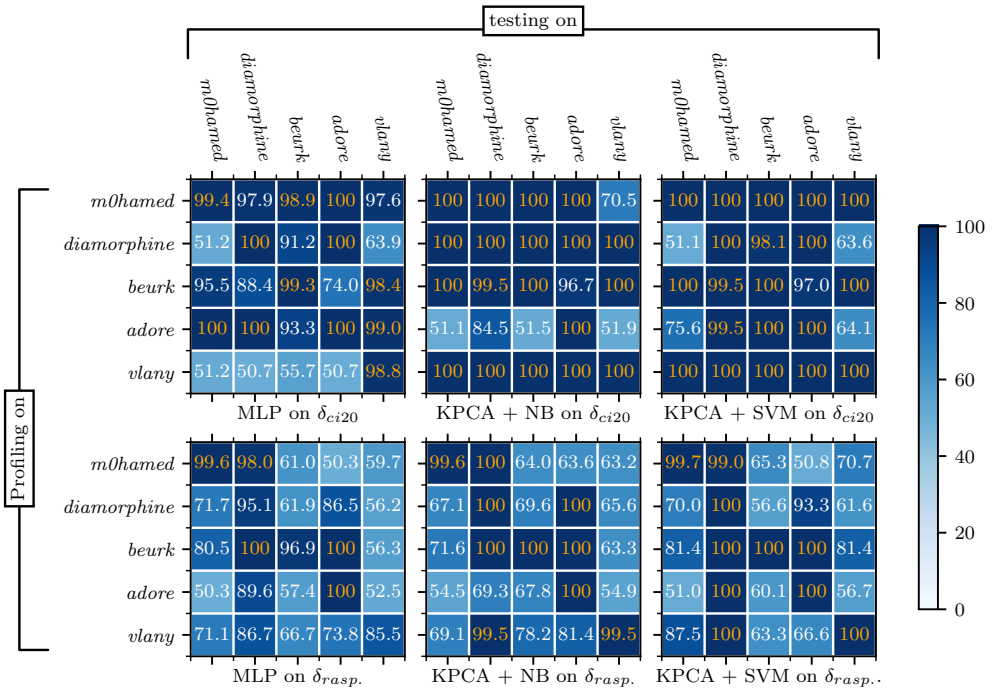


Figure 4: Novelty rootkit detection. Each cell refers to an experiment, the row (resp. the column) informs on the rootkit(s) seen during the offline learning phase (resp. the online testing phase). Except on the diagonal, learning and testing sets are exclusive. Numbers are balanced accuracies, the darker the blue color the higher the balanced accuracy. All experiments share the same bait β = {*getdents*}, the same probe $p_{k,t}$, and benign activities are drawn randomly from Γ. δ_{ci20} on top; $\delta_{rasp.}$ on bottom.

between rootkits, even if they work on different levels (user vs. kernel).

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}.

6.2.4 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 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 $r_i \neq r_j$ with 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.

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

6.2.6 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 having extra benign activities that may deceive the models (e.g., netfilter, VFS hooks, ecryptfs etc.). Table 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

Table 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 $\delta_{\text{rasp.}}$, δ_{ci20}. We use the notation $r_1 \nrightarrow r_2$ 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.

	δ_{ci20}									$\delta_{\text{rasp.}}$								
	MLP			**KPCA + NB**			**KPCA + SVM**			**MLP**			**KPCA + NB**			**KPCA + SVM**		
Scenario	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**
$m \nrightarrow m$	$99.4_{[5]}$	99.0	99.8	$100_{[7]}$	100	100	$100_{[1]}$	100	100	$99.6_{[34]}$	99.2	100	$99.6_{[17]}$	99.3	100	$99.7_{[36]}$	99.4	100
$m_o \nrightarrow m_o$	$100_{[13]}$	100	100	$100_{[15]}$	100	100	$100_{[10]}$	100	100	$100_{[25]}$	100	100	$100_{[14]}$	100	100	$100_{[30]}$	100	100
$m \nrightarrow m_o$	$100_{[13]}$	100	100	$100_{[15]}$	100	100	$100_{[10]}$	100	100	$98.2_{[25]}$	96.3	100	$90.3_{[14]}$	80.6	100	$100_{[30]}$	100	100
$m_o \nrightarrow m$	$97.4_{[5]}$	96.7	98.2	$100_{[7]}$	100	100	$100_{[1]}$	100	100	$100_{[34]}$	100	100	$100_{[17]}$	100	100	$100_{[36]}$	100	100
$d \nrightarrow d$	$100_{[12]}$	100	100	$100_{[9]}$	100	100	$100_{[7]}$	100	100	$95.1_{[4]}$	97.2	93.1	$100_{[21]}$	100	100	$100_{[21]}$	100	100
$d_o \nrightarrow d_o$	$100_{[10]}$	100	100	$100_{[8]}$	100	100	$100_{[9]}$	100	100	$100_{[22]}$	100	100	$100_{[11]}$	100	100	$100_{[16]}$	100	100
$d \nrightarrow d_o$	$53.7_{[10]}$	7.4	100	$100_{[8]}$	100	100	$58.3_{[9]}$	16.6	100	$100_{[22]}$	100	100	$97.5_{[11]}$	95.0	100	$100_{[16]}$	100	100
$d_o \nrightarrow d$	$52.3_{[12]}$	4.9	99.6	$100_{[9]}$	100	100	$53.8_{[7]}$	8.6	99.1	$83.0_{[4]}$	66.2	99.8	$100_{[21]}$	100	100	$100_{[21]}$	100	100

Table 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 $\delta_{\text{rasp.}}$, δ_{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 $s \nrightarrow m$ while the reverse scenario is denoted by $m \nrightarrow s$.

		δ_{ci20}									$\delta_{\text{rasp.}}$								
		MLP			**KPCA + NB**			**KPCA + SVM**			**MLP**			**KPCA + NB**			**KPCA + SVM**		
Scenario		**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**
$m \nrightarrow s$	*swkb*	$66.8_{[5]}$	34.8	98.8	$94.9_{[9]}$	89.8	100	$100_{[2]}$	100	100	$100_{[23]}$	100	100	$100_{[8]}$	100	100	$100_{[8]}$	100	100
	hwkb	$50.6_{[8]}$	92.8	8.4	$50.0_{[8]}$	0.0	100	$51.0_{[16]}$	2.9	99.2	$100_{[14]}$	100	100	$100_{[10]}$	100	100	$100_{[13]}$	100	100
$s \nrightarrow m$	*swkb*	$57.1_{[15]}$	14.8	99.5	$100_{[13]}$	100	100	$100_{[7]}$	100	100	$100_{[11]}$	100	100	$100_{[9]}$	100	100	$100_{[4]}$	100	100
	hwkb	$46.9_{[15]}$	43.5	50.2	$50.0_{[27]}$	0.0	100	$48.2_{[10]}$	0.1	96.3	$99.5_{[15]}$	99.1	100	$85.3_{[9]}$	70.6	100	$100_{[12]}$	100	100

Table 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 $\mathcal{R}$ = {m0hamed} and the benign activity is randomly chosen from Γ.

	MLP			**KPCA + NB**			**KPCA + SVM**		
Sc.*	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**
S_0	$100_{[4]}$	100	100	$100_{[13]}$	100	100	$100_{[21]}$	100	100
S_1	$55.4_{[5]}$	99.0	11.7	$100_{[7]}$	100	100	$55.7_{[1]}$	100	11.4

***Sc.** stands for **Scenario**.

false positive. In conclusion, even if there is additional unexpected activity from the LKM, NB is able to detect without any errors.

6.2.7 Noise evaluation. 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). Table 10 demonstrates the necessity of noise during the training phase. In fact, models build with no extra background benign activity ($Q \nrightarrow N$) 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).

6.3 Discussion

6.3.1 Performance evaluation against host-based tools. We compare our detection results to other up-to-date open source 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 target device δ_{Raspi}, except ULTRA was executed on a remote host agent: Intel Xeon W-2104@4x 3.2GHz CPU with Quadro P2200 GPU. 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 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, 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

Table 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 $\delta_{\text{rasp.}}$, δ_{ci20}, the malware $\mathcal{R} = \{$m0hamed-obed$\}$.

	δ_{ci20}									$\delta_{\text{rasp.}}$								
	MLP			**KPCA + NB**			**KPCA + SVM**			**MLP**			**KPCA + NB**			**KPCA + SVM**		
Scenario	**BA** $[\epsilon_{\text{opt}}]$	**TPR**	**TNR**	**BA** $[\epsilon_{\text{opt}}]$	**TPR**	**TNR**	**BA** $[\epsilon_{\text{opt}}]$	**TPR**	**TNR**	**BA** $[\epsilon_{\text{opt}}]$	**TPR**	**TNR**	**BA** $[\epsilon_{\text{opt}}]$	**TPR**	**TNR**	**BA** $[\epsilon_{\text{opt}}]$	**TPR**	**TNR**
$N \rightarrow Q$	$54.1_{[6]}$	96.1	12.2	$98.7_{[7]}$	100	97.4	$100_{[12]}$	100	100	$100_{[11]}$	100	100	$100_{[10]}$	100	100	$50.0_{[11]}$	100	0.0
$Q \rightarrow N$	$58.1_{[3]}$	100	16.3	$52.9_{[7]}$	100	5.9	$50.0_{[3]}$	100	0.0	$50.0_{[1]}$	100	0.0	$50.0_{[1]}$	100	0.0	$50.0_{[1]}$	100	0.0
$N \rightarrow N$	$99.1_{[6]}$	98.2	100	$100_{[7]}$	100	100	$100_{[12]}$	100	100	$100_{[11]}$	100	100	$100_{[10]}$	100	100	$100_{[11]}$	100	100

the rootkit (except in the keylogger detection scenario), in contrast to LKRG which requires the occurrence of malicious behaviors explicitly.

Table 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‡

6.3.2 Invariant to probe position. Fig. 7 in Appendix 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.5). The bait *open* was used to detect the presence of rootkit *beurk*. Table 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 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.

Figure 5: ULTRA framework is installed with an handcrafted EM-compatible probe (1) placed 10mm above Ci20's processor (2) to detect *beurk* rootkit.

6.3.3 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.*[24] 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 [14] presents a novel technique by exploiting DRAM accesses to shape arbitrary signals out of EM noise from unprivileged software. Therefore, a rootkit may try to tamper by generating noise during execution. However, it still raises a challenge for rootkit authors: finding a solution to generate noise without causing any deviation between infected and benign states, 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 [60], if an advanced rootkit is aware of the occurrence of a bait, it can hide its modifications before the bait is presented and re-activate them afterwards. As per the bait design requirement, the execution of a bait must present no difference from the

Table 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 $\mathcal{R} = \{\textit{beurk}\}$ 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** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**
$\{0,0\} \rightarrow \{0,0\}$	$100_{[2]}$	100	100	$100_{[2]}$	100	100	$100_{[2]}$	100	100
$\{0,0\} \rightarrow \{1,0\}$	$100_{[2]}$	100	100	$100_{[2]}$	100	100	$100_{[2]}$	100	100
$\{0,0\} \rightarrow \{2,1\}$	$60.6_{[2]}$	21.4	99.9	$50.0_{[2]}$	0.0	100	$50.0_{[2]}$	0.0	100
$\{1,0\} \rightarrow \{1,0\}$	$100_{[2]}$	100	100	$100_{[3]}$	100	100	$100_{[2]}$	100	100
$\{2,1\} \rightarrow \{2,1\}$	$100_{[1]}$	100	100	$100_{[4]}$	100	100	$100_{[4]}$	100	100

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.

7 CONCLUSIONS AND PERSPECTIVES

Stealthy nonactive rootkits constitute a real challenge for (host-based) malware analysis systems. In this paper, 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 wave-and-play. Although the detection rate of 100% for both known and unknown variants 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. 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 (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.

Acknowledgement. The work was supported by the French Agence Nationale de la Recherche (ANR) under reference ANR-18-CE39-0001 (AHMA). We thank our colleague Olivier Zendra and Ronan Lashermes who provided insights that greatly assisted this work. We thank Simone Aonzo and the anonymous reviewers for their careful reading of our paper and their comments and suggestions.

REFERENCES

[1] 2021. GNU Radio. https://www.gnuradio.org/about/. Accessed: 2022-01-01.
[2] 2021. GREAT SCOTT GADGETS HackRF One. https://greatscottgadgets.com/hackrf/one/. Accessed: 2022-01-01.
[3] 2021. Rootkits: evolution and detection methods. https://www.ptsecurity.com/ww-en/analytics/rootkits-evolution-and-detection-methods/. Accessed: 2022-01-10.
[4] Vipindev Adat and Brij B Gupta. 2018. Security in Internet of Things: issues, challenges, taxonomy, and architecture. *Telecommunication Systems* 67, 3 (2018), 423–441.
[5] National Security Agency and Federal Bureau of Investigation. 2021. 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. Accessed: 2022-01-01.
[6] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2011. Detecting Kernel-Level Rootkits Using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing* 8, 5 (2011), 670–684. https://doi.org/10.1109/TDSC.2010.38
[7] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. 2014. NICV: Normalized Inter-Class Variance for Detection of Side-Channel Leakage. In *International Symposium on Electromagnetic Compatibility (EMC '14 / Tokyo)*. IEEE. eprint version: https://eprint.iacr.org/2013/717.pdf.
[8] Michael Boelen and John Horne. 2012. The rootkit hunter project. *Online]. http://rkhunter.sourceforge.net* (2012).
[9] Sergey Bratus, Michael E Locasto, Ashwin Ramaswamy, and Sean W Smith. 2010. VM-based security overkill: a lament for applied systems security research. In *Proceedings of the 2010 New Security Paradigms Workshop*. 51–60.
[10] Rory Bray, Daniel Cid, and Andrew Hay. 2008. *OSSEC host-based intrusion detection guide*. Syngress.
[11] Robert Bridges, Jarilyn Hernández Jiménez, Jeffrey Nichols, Katerina Goseva-Popstojanova, and Stacy Prowell. 2018. 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)*. IEEE, 1680–1684.
[12] Robert Buhren, Julian Vetter, and Jan Nordholz. 2016. The threat of virtualization: Hypervisor-based rootkits on the ARM architecture. In *International Conference on Information and Communications Security*. Springer, 376–391.
[13] Andreas Bunten. 2004. Unix and linux based rootkits techniques and countermeasures. In *16th Annual First Conference on Computer Security Incident Handling, Budapest*.
[14] Giovanni Camurati and AurÃ©lien Francillon. 2022. Noise-SDR: Arbitrary modulation of electromagnetic noise from unprivileged software and its impact on emission security. In *IEEE Symposium on Security and Privacy* (San Francisco, CA). IEEE Computer Society.
[15] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurelien Francillon. 2018. Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto Canada, 163–177. https://doi.org/10.1145/3243734.3243802
[16] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Wenyuan Xu, and Kevin Fu. 2013. WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices. In *2013 USENIX Workshop on Health Information Technologies (HealthTech 13)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/

healthtech13/workshop-program/presentation/clark
[17] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux malware. In *S&P 2018, 39th IEEE Symposium on Security and Privacy, May 21-23, 2018, San Francisco, CA, USA*, IEEE (Ed.). San Francisco. 2018 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE..
[18] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 20–38.
[19] Fei Ding, Hongda Li, Feng Luo, Hongxin Hu, Long Cheng, Hai Xiao, and Rong Ge. 2020. 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*. 33–46.
[20] Shawn Embleton, Sherri Sparks, and Cliff C Zou. 2013. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks* 6, 12 (2013), 1590–1605.
[21] Nicolas Falliere, Liam O Murchu, and Eric Chien. 2011. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response* 5, 6 (2011), 29.
[22] Ibraheem Frieslaar and Barry Irwin. 2017. Recovering AES-128 encryption keys from a Raspberry Pi. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*. 228–235.
[23] Ibraheem Frieslaar and Barry Irwin. 2017. Recovering AES-128 encryption keys from a Raspberry Pi. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*. 228–235.
[24] I Frieslaar and B Irwin. 2018. Developing an electromagnetic noise generator to protect a Raspberry Pi from side channel analysis. *SAIEE Africa Research Journal* 109, 2 (2018), 85–101.
[25] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. 2006. Templates vs. Stochastic Methods. In *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4249)*, Louis Goubin and Mitsuru Matsui (Eds.). Springer, 15–29. https://doi.org/10.1007/11894063_2
[26] Mordechai Guri, Yuri Poliak, Bracha Shapira, and Yuval Elovici. 2015. JoKER: Trusted detection of kernel rootkits in android devices via JTAG interface. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. IEEE, 65–73.
[27] Seunghun Han. 2020. Adore-NG v2.5. https://github.com/kkamagui/adore-ng Accessed: 2022-02-10.
[28] Seunghun Han and JH Park. 2018. Shadow-box v2: The practical and omnipotent sandbox for arm. *2018, slideshow at Blackhat Asia* (2018).
[29] David Harley and Andrew Lee. 2007. The root of all evil?-rootkits revealed.
[30] Annelie Heuser and Michael Zohner. 2012. Intelligent Machine Homicide - Breaking Cryptographic Devices Using Support Vector Machines. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7275)*, Werner Schindler and Sorin A. Huss (Eds.). Springer, 249–264. https://doi.org/10.1007/978-3-642-29912-4_18
[31] Trammell Hudson and Larry Rudolph. 2015. Thunderstrike: EFI firmware bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*. 1–10.
[32] Arun Prakash Jana. 2021. spy v1.8. https://github.com/jarun/spy Accessed: 2022-02-10.
[33] Xingbin Jiang, Michele Lora, and Sudipta Chattopadhyay. 2020. 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*. 433–438.
[34] George H. John, Ron Kohavi, and Karl Pfleger. 1994. Irrelevant Features and the Subset Selection Problem. In *Machine Learning, Proceedings of the Eleventh International Conference, Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994*, William W. Cohen and Haym Hirsh (Eds.). Morgan Kaufmann, 121–129. https://doi.org/10.1016/b978-1-55860-335-6.50023-4
[35] Juho Junnila. 2020. Effectiveness of Linux Rootkit Detection Tools. (2020).
[36] Haider Adnan Khan, Nader Sehatbakhsh, Luong N. Nguyen, Milos Prvulovic, and Alenka Zajić. 2019. Malware Detection in Embedded Systems Using Neural Network Model for Electromagnetic Side-Channel Signals. *Journal of Hardware and Systems Security* (Aug. 2019). https://doi.org/10.1007/s41635-019-00074-w
[37] Iain Kyte, Pavol Zavarsky, Dale Lindskog, and Ron Ruhl. 2012. Enhanced side-channel analysis method to detect hardware virtualization based rootkits. In *World Congress on Internet Security (WorldCIS-2012)*. 192–201.
[38] John Levine, Julian Grizzard, and Henry Owen. 2004. 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.* IEEE, 107–125.
[39] Patrick Luckett, J Todd McDonald, William B Glisson, Ryan Benton, Joel Dawson, and Blair A Doyle. 2018. Identifying stealth malware using CPU power consumption and learning algorithms. *Journal of Computer Security* 26, 5 (2018), 589–613.
[40] m0hamed. 2015. lkm-rootkit: A rootkit implemented as a linux kernel module. https://github.com/m0hamed/lkm-rootkit Accessed: 2022-02-10.
[41] m0nad. 2021. Diamorphine: a LKM rootkit. https://github.com/m0nad/Diamorphine Accessed: 2022-02-10.
[42] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power analysis attacks - revealing the secrets of smart cards.* Springer.
[43] Ciarán McNally. 2015. maK_it-Linux-Rootkit. https://web.archive.org/web/20190119045332/https://r00tkit.me/ Accessed: 2022-02-10.
[44] mempodippy. 2019. vlany: a Linux LD_PRELOAD rootkit. https://github.com/mempodippy/vlany Accessed: 2022-02-10.
[45] Nelson Murilo and Klaus Steding-Jessen. 2001. 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)*. 133–139.
[46] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic. 2017. EDDIE: EM-based detection of deviations in program execution. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 333–346. https://doi.org/10.1145/3079856.3080223
[47] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. 2021. A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM Implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 4 (2021), 676–707. https://doi.org/10.46586/tches.v2021.i4.676-707
[48] Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo. 2010. Bait your hook: a novel detection technique for keyloggers. In *International workshop on recent advances in intrusion detection*. Springer, 198–217.
[49] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. 2004. Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor.. In *USENIX security symposium*. San Diego, USA, 179–194.
[50] Duy-Phuc Pham, Damien Marion, Mathieu Mastio, and Annelie Heuser. 2021. Obfuscation Revealed: Leveraging Electromagnetic Signals for Obfuscated Malware Classification. In *Annual Computer Security Applications Conference*.
[51] Valerian Rey, Pedro Miguel Sánchez Sánchez, Alberto Huertas Celdrán, and Gérôme Bovet. 2021. Federated Learning for Malware Detection in IoT Devices. 204 (2021), 108693. https://doi.org/10.1016/j.comnet.2021.108693
[52] Khaled Riad, Teng Huang, and Lishan Ke. 2020. A dynamic and hierarchical access control for IoT in multi-authority cloud storage. *Journal of Network and Computer Applications* 160 (2020), 102633.
[53] Joanna Rutkowska. 2006. Introducing blue pill. *The official blog of the invisiblethings. org* 22 (2006), 23.
[54] Asanka Sayakkara, Nhien-An Le-Khac, and Mark Scanlon. 2019. Leveraging Electromagnetic Side-Channel Analysis for the Investigation of IoT Devices. *Digital Investigation* 29 (July 2019), S94–S103. https://doi.org/10.1016/j.diin.2019.04.012
[55] Tobias Schneider and Amir Moradi. 2016. Leakage assessment methodology - Extended version. *J. Cryptogr. Eng.* 6, 2 (2016), 85–99. https://doi.org/10.1007/s13389-016-0120-y
[56] N. Sehatbakhsh, A. Nazari, M. Alam, F. Werner, Y. Zhu, A. Zajic, and M. Prvulovic. 2020. REMOTE: Robust External Malware Detection Framework by Using Electromagnetic Signals. *IEEE Trans. Comput.* 69, 3 (2020), 312–326.
[57] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. 2017. 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*. 483–493.
[58] Unix-Thrust. 2017. Unix-Thrust/Beurk: Beurk Experimental unix rootkit. https://github.com/unix-thrust/beurk Accessed: 2022-02-10.
[59] Danilo Valerio. 2008. Open source software-defined radio: A survey on gnuradio and its applications. *Forschungszentrum Telekommunikation Wien, Vienna, Technical Report FTW-TR-2008-002* (2008).
[60] Xueyang Wang and Ramesh Karri. 2013. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.
[61] Xiao Wang, Quan Zhou, Jacob Harer, Gavin Brown, Shangran Qiu, Zhi Dou, John Wang, Alan Hinton, Carlos Aguayo Gonzalez, and Peter Chin. 2018. Deep learning-based classification and anomaly detection of side-channel signals. In *Cyber Sensing 2018*, Vol. 10630. International Society for Optics and Photonics, 1063006.
[62] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. 2009. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*. 545–554.
[63] Adam Zabrocki. 2018. Linux Kernel Runtime Guard (LKRG) under the Hood. In *CONFidence Conference*.
[64] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2018. Hardware performance counters can detect malware: Myth or fact?. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 457–468.

APPENDICES

Bait specification

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

Table 13: 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

Code 1: Network TCP bait script

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

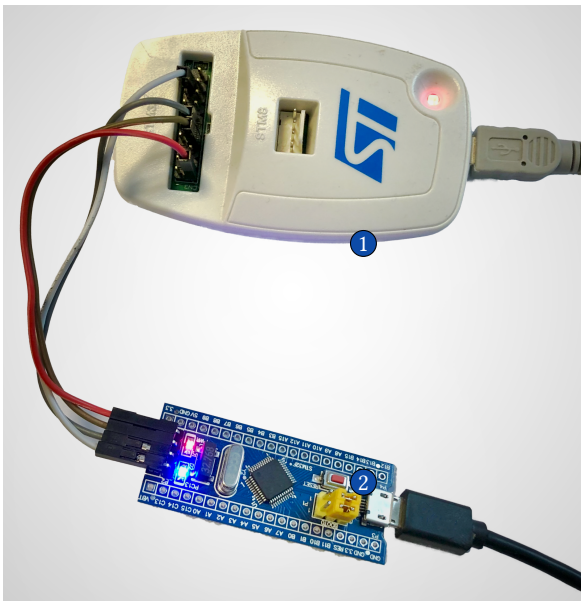


Figure 6: Hardware keyboard emulator bait consists of one Blue Pill STM32 board (1) connected to a ST-link v2 (2) which under control of ULTRA host agent.

Probe dislocation

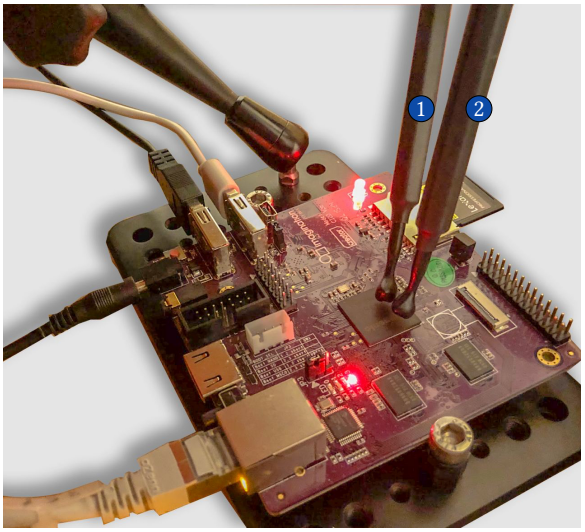


Figure 7: Invariant to probe position. Framework setup with 2 probes of the same type placing contactless at 2 different locations placed 10mm above the processor (1) (2).

Patch snippet for static string obfuscation

Code 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*/
```

Enhancement through meaning testing traces

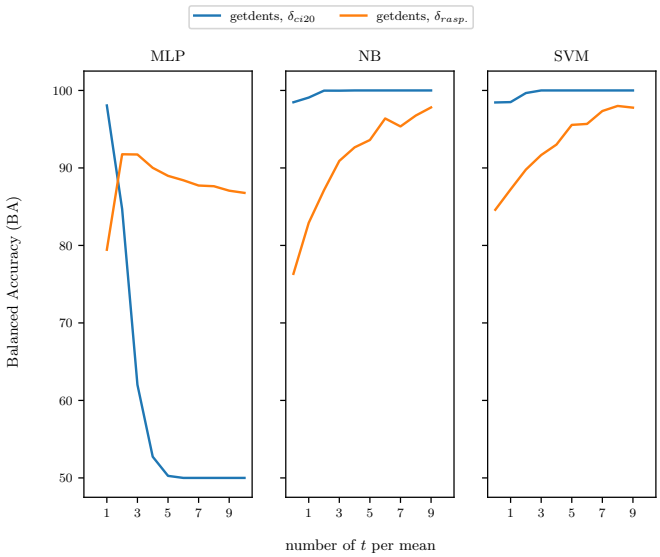


Figure 8: Balanced accuracy (BA) of the Table 5 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

Neural network architectures

Table 14: MLP architecture

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)

ULTRA's bill of materials

Table 15: ULTRA's bill of materials

Equipment	Rate/Unit	Count	Amount (Euro)
HackRF One SDR	309	1	309
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.*

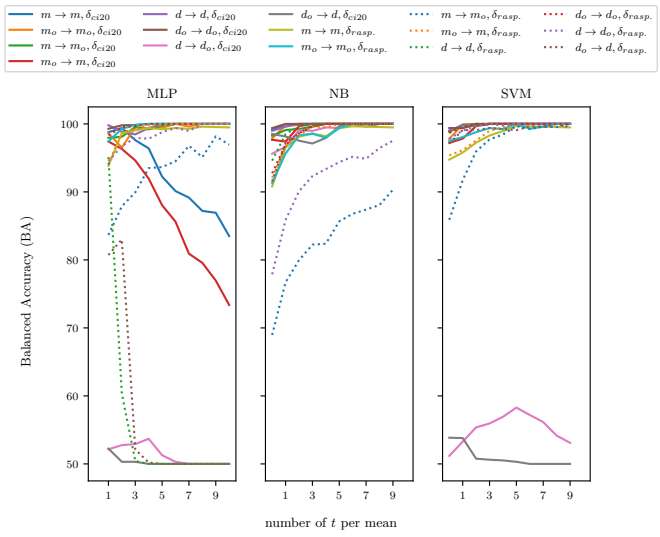


Figure 9: Balanced accuracy (BA) of the Table 7 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

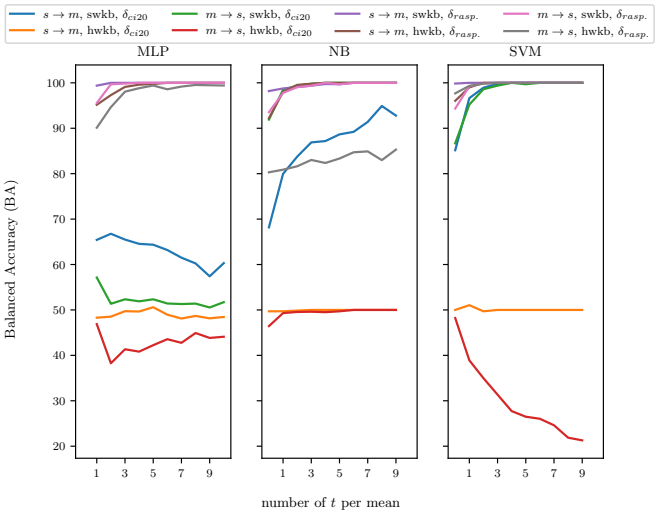


Figure 10: Balanced accuracy (BA) of the Table 8 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

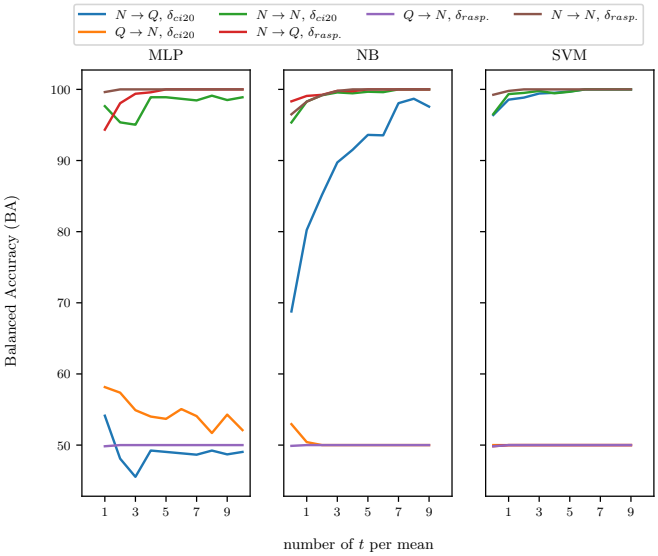


Figure 12: Balanced accuracy (BA) of the Table 10 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

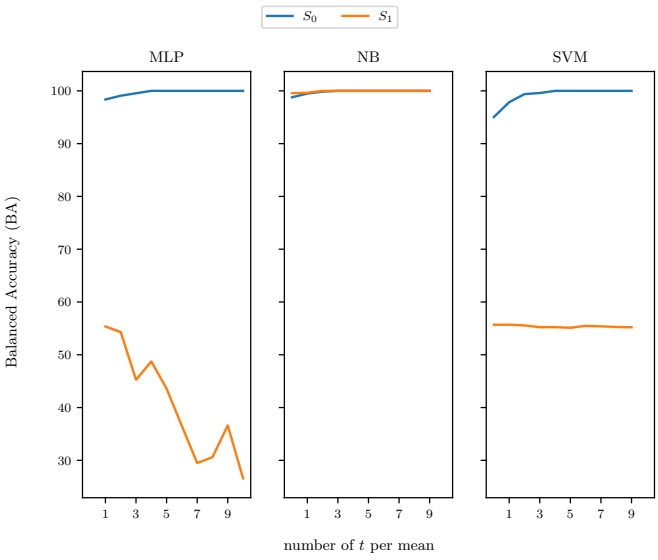


Figure 11: Balanced accuracy (BA) of the Table 9 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

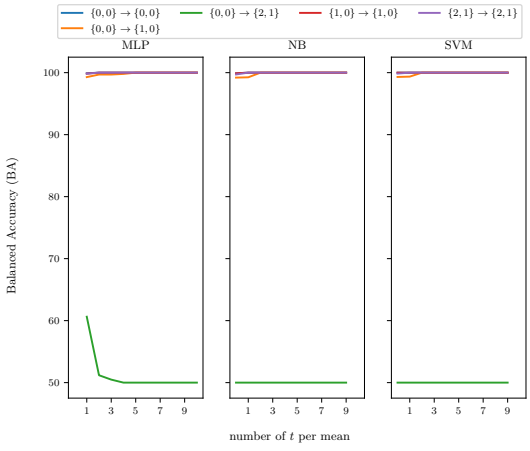


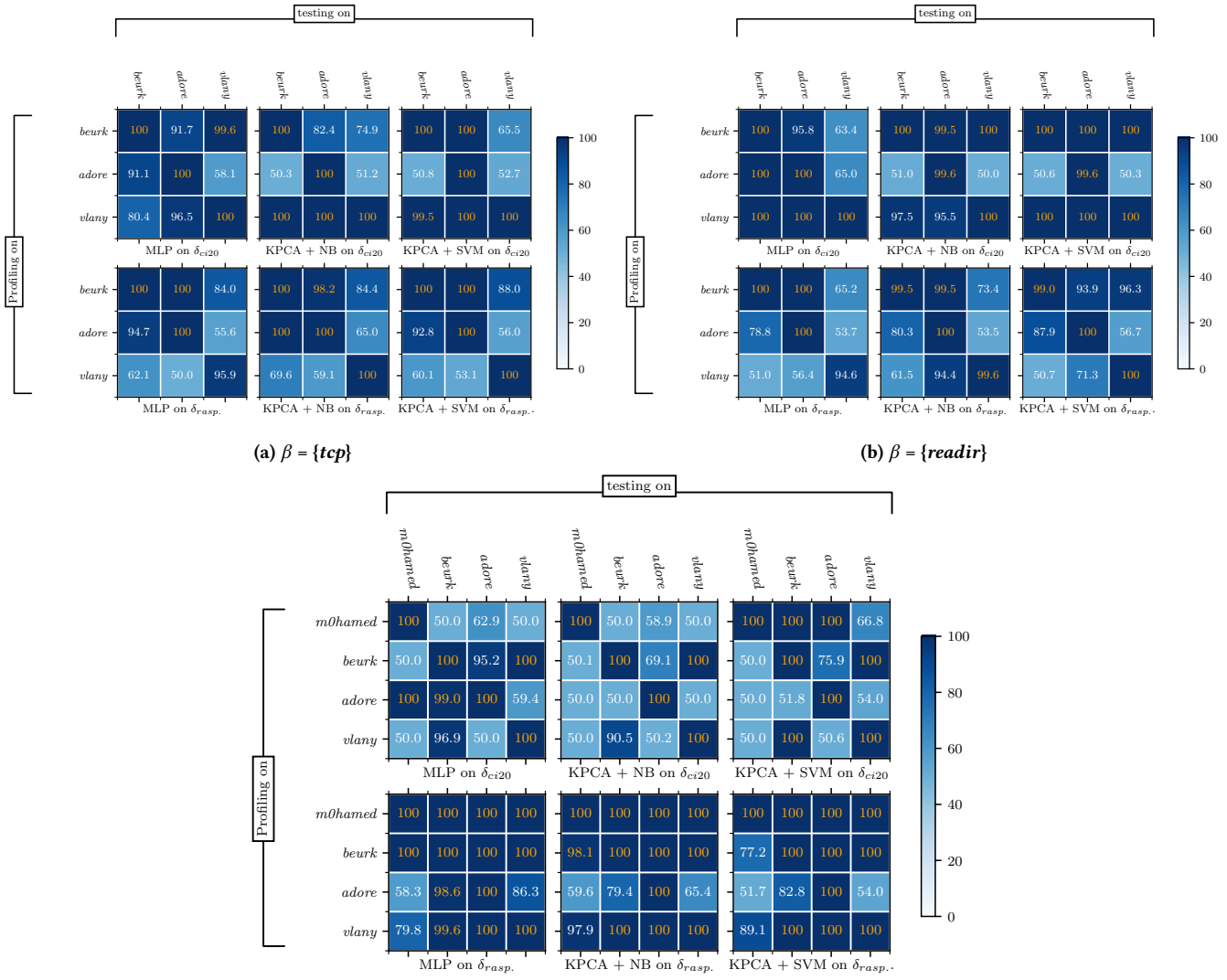
Figure 13: Balanced accuracy (BA) of the Table 12 displaying the mean process over $t = [1, 2, \ldots 10]$ samples per class (infected or clean) in the test dataset.

Algorithm 2 Bandwidth extraction procedure

```
nicv = NICV (learning_set) ∈ ℝ^{F×D}        ▷ the learning set is composed of labed spectrogram of dimension F × D
max_nicv = max_D(nicv) ∈ ℝ^F
sorted_bandwidth = argsort (max_nicv)

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
```

Table 16: Classification scenario distinguishing kernel-space and user-space rootkits in (S_0), then in (S_1) we add benign samples.

	δ_{ci20}						$\delta_{\text{rasp.}}$					
	KPCA + NB			**KPCA + SVM**			**KPCA + NB**			**KPCA + SVM**		
Scenario	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**	**BA** $_{[\epsilon_{\text{opt}}]}$	**TPR**	**TNR**
S_0	$100_{[95]}$	100	100	$97.6_{[95]}$	97.6	97.6	$82.7_{[1]}$	86.8	82.7	$98.4_{[5]}$	98.4	98.4
S_1	$99.2_{[50]}$	99.2	99.2	$96.5_{[25]}$	96.5	96.5	$65.9_{[9]}$	74.3	65.9	$90.5_{[8]}$	91.4	90.5



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

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

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

Figure 14: Novelty rootkit detection. Same description as the Fig. 4, but with differents baits β.