HADES-IoT: A Practical and Effective Host-Based Anomaly Detection System for IoT Devices (Extended Version)

Dominik Breitenbacher, Ivan Homoliak, *Member, IEEE*, Yan Lin Aung, Yuval Elovici, and Nils Ole Tippenhauer, *Member, IEEE*

***Abstract*—Internet of Things (IoT) devices have become ubiquitous, with applications in many domains, including industry, transportation, and healthcare; these devices also have many household applications. The proliferation of IoT devices has raised security and privacy concerns, however many manufacturers neglect these aspects, focusing solely on the core functionality of their products due to the short time to market and the need to reduce product costs. Consequently, vulnerable IoT devices are left unpatched, allowing attackers to exploit them for various purposes, which include compromising the device users' privacy or recruiting the devices to an IoT botnet. We present a practical and effective host-based anomaly detection system for IoT devices (HADES-IoT) as a novel last line of defense. HADES-IoT has proactive detection capabilities that enable the execution of any malicious process to be stopped before it even starts. HADES-IoT provides tamper-proof protection and can be deployed on a wide range of Linux-based IoT devices. HADES-IoT's main advantage is its low overhead, making it suitable for Linux-based IoT devices where state-of-the-art security solutions are infeasible due to their high-performance demands. We deployed HADES-IoT on seven IoT devices, where it demonstrated 100% effectiveness in the detection of IoT malware, including VPNFilter, IoT Reaper, and Mirai malware, while requiring only 5.5% (on average) of the available memory and consuming just negligible CPU resources.**

***Index Terms*—Host-based anomaly detection, intrusion detection, loadable kernel module (LKM), Security and privacy, system call interception, tamper-proof protection.**

I. Introduction

THE WIDESPREAD adoption of Internet of Things (IoT) devices has raised challenges in various areas, including data storage and maintenance, however it poses even greater challenges to privacy and security [1]–[3]. IoT devices are often designed with a particular purpose in mind, and thus their software and hardware is based solely on meeting the requirements of core functionalities, e.g., using a processor that is only as fast as it needs to be to meet certain constraints. However, when it comes to security and privacy, the emphasis placed on accelerating the time to market and working within budgetary constraints has limited the use of more expensive, and possibly more comprehensive, security and privacy-protecting solutions. Therefore, the embedded technology is accompanied by a tradeoff between cost and security [1].

For these reasons, IoT devices are often released with serious vulnerabilities, an issue that is further exacerbated by the fact that many IoT devices are exposed on the Internet, making them easily accessible by attackers. Once a vulnerable device is compromised, it can be exploited for various purposes. The most common scenario is that the compromised device becomes a part of an IoT botnet that performs DDoS attacks [4]–[6], however attackers have recently begun to utilize IoT devices for cryptocurrency mining as well [7], [8]. Furthermore, data processed by IoT devices are often privacy-sensitive; such data include pictures/videos taken by IP cameras, documents printed by printers, and even programs watched on smart TVs. Privacy-oriented vulnerabilities have already been discovered in printers [9], smart TVs [10], and IP cameras [11]–[13].

Most IoT devices are exploited due to operations security (OPSEC) issues, such as the use of weak or default passwords [14], [15], however they can also be exploited due to vulnerabilities, such as buffer overflow and command injection attacks [16]. These practical examples raise the question of how to increase the security of IoT devices while minimizing the cost and computational resource requirements of the security solutions. One way to raise the bar against attackers is to improve proactive OPSEC countermeasures, such as employing strong passwords and conservative access control. However, in the case of more sophisticated attacks that cannot be prevented by OPSEC countermeasures, IoT devices must be protected by other dedicated means, such as host-based intrusion detection systems or IoT-specific antivirus systems. This may be challenging as there are limited economic incentives for companies that develop low-cost IoT devices to invest in such computationally expensive countermeasures and protection mechanisms. As we have learned after many years of research on IT security (and the

Manuscript received March 30, 2021; revised June 28, 2021 and November 17, 2021; accepted November 29, 2021. Date of publication December 15, 2021; date of current version June 7, 2022. This work was supported in part by the H2020 ECSEL Project Arrowhead Tools under Grant 8A19010 and Grant 826452, and in part by the Internal Project of BUT—IGA FIT-S-20-6427. *(Corresponding author: Ivan Homoliak.)*

Dominik Breitenbacher and Ivan Homoliak are with the Faculty of Information Technology, Brno University of Technology, 61200 Brno, Czech Republic (e-mail: ihomoliak@fit.vutbr.cz).

Yan Lin Aung is with the Singapore University of Technology and Design, Singapore.

Yuval Elovici is with the Ben-Gurion University of the Negev, Beer-Sheva 653, Israel.

Nils Ole Tippenhauer is with the CISPA Helmholtz Center for Information Security, 66123 Saarbrucken, Germany.

Digital Object Identifier 10.1109/JIOT.2021.3135789

security of software specifically), serious vulnerabilities can remain hidden due to the complexity of modern systems and the difficulty in revealing such vulnerabilities. For example, even a securely written application can be vulnerable due to bugs in third-party libraries, compilers, or even operating systems.

Given the above-mentioned issues, we make the following critical observation: in contrast to general computing devices, such as PCs, IoT devices have, by design, well-defined, and stable functionality. Moreover, this functionality is usually provided by a small set of system processes whose behavior is largely stable, since IoT devices are rarely updated. However, when an IoT device is compromised by malware, its behavior changes significantly. Therefore, intrusion detection and anomaly detection systems are promising options for the detection of compromised IoT devices. Although, these systems can also be deployed outside of the device and inspect the network traffic [17]–[20], they can be evaded by various payload-based [21], [22] and nonpayload-based obfuscations [23], [24]. Hence, we argue that behavioral changes are best observed from "within" a device, for example, by monitoring which processes run on a device and what actions they perform. For this reason, we consider host-based behavior analysis an effective last line of defense. More specifically, we aim to adapt an anomaly detection approach in a process-whitelisting manner that profiles the normal behavior of an IoT device and detects all unknown processes as anomalies deviating from this profile.

In this article, we propose a lightweight and practical host-based anomaly[1] detection system for IoT devices (HADES-IoT) that monitors process spawning and stops any unauthorized process before its execution. To achieve real-time response, HADES-IoT was developed in the form of a loadable kernel module (LKM) for Linux-based operating systems. This design contributes to HADES-IoT's tamper resistance, enabling it to withstand attacks aimed at disabling it (see Section V-D for details). Since IoT devices are resource constrained, HADES-IoT was created in a lightweight fashion, preserving the primary functionality of the device.

Contributions: In summary, our contributions are as follows.

1) We present a novel proactive host-based anomaly detection approach based on whitelisting legitimate processes on Linux-based IoT devices.
2) We develop a proof-of-concept as a Linux kernel module and evaluate its effectiveness on several IoT devices; we perform penetration testing on these devices using IoT malware that infects the devices and recruits them to botnets.
3) We show that HADES-IoT can be easily adapted to any Linux kernel version, which enables its use on various types of IoT devices.
4) We demonstrate HADES-IoT's resilience against attacks focused on disabling its protection mechanisms, highlighting its tamper-proof design.

Extensions: In addition to the work described in our preliminary paper introducing HADES-IoT [26], we extend our work as follows.

1) We describe detailed preliminaries that a enable better understanding of HADES-IoT's principles of operation.
2) As part of our evaluation, we describe a precompilation process for HADES-IoT's kernel module, as well as outstanding issues whose resolution will enable HADES-IoT's widespread deployment.
3) We propose a reporting subsystem that enables remote notifications regarding prevented attack attempts that occurred on the devices to owners/administrators of IoT devices.
4) We discuss honeypot capabilities that might simplify the process of signature creation for unknown malware.
5) We propose an extension for the remote control of HADES-IoT. In the extension, we deal with the problem of limited storage and available libraries for authentication, and we utilize a low-demanding Merkle signature scheme.
6) We demonstrate the operation of the proposed remote control application on two use cases—secure firmware update and deployment of HADES-IoT.
7) We propose an extension that handles signal monitoring in a way that prevents malware from disrupting the legitimate operation of a protected IoT device.

Organization: The remainder of this article is organized as follows. Section II is dedicated to related work. We present the problem statement in Section III. Preliminaries of our work are described in Section IV. Then in Sections V and VI we describe our approach and its extensions. We present our evaluation of HADES-IoT in Section VII. Section VIII discusses limitations and possible extensions of HADES-IoT, and Section IX concludes this article.

II. Related Work

In this section, we discuss related research aimed at host-based intrusion detection systems for IoT devices.[2] The summary of the related work is presented in Table I, where we compare it with HADES-IoT in terms of the detection principle, data source utilized for detection, IoT devices included, and detection performance. In the overview provided below, we present the related research based on the detection principle underlying the proposed method [25], [45] as follows: *anomaly based*, *misuse based*, and *classification based*.

Anomaly Based Approaches: All of the approaches in this category only require legitimate data to train the intrusion detection model. In general, a disadvantage seen in this category is a high false-positive rate (FPR), since the behavior of users and processes on conventional hosts may vary over time (see Section VIII-C). However, we argue that IoT devices have stable behavior, which enables this type of detection to

[1]Note that by anomaly detection we refer to the principle on which intrusion detection systems are based, in which a profile of normal behavior is modeled and deviations from the model are detected [25], as opposed to misuse-based detection which relies on evidence of attacks based on the knowledge accumulated from known attacks (e.g., signatures).

[2]For host-based intrusion detection in general, we refer the interested reader to the recent work of Bridges *et al.* [43] and Liu *et al.* [44].

TABLE I
COMPARISON OF RELATED WORK AND HADES-IOT

Intrusion Detection System for IoT	Detection Principle	Technique	Data Source	OS Running on IoT Devices	Target Devices	# of Various Devices	Best TPR [%]	Best FPR [%]	Detection Requires External Device
Yoon et al. [27]	Anomaly	Cluster analysis	System calls	Linux	Raspberry PI 2 with ARM Cortex-A7	1	–	0.23	Yes
Tabrizi et al. [28]	Anomaly	Model checking	System calls	OpenWrt Linux	Smart meter SEGMeter with Broadcom BCM3302 MCU	1	88.00	–	Yes
Agarwal et al. [29]	Anomaly	Control-flow path modeling	System calls	FreeRTOS	Freescales FRDM-K64F	1	–	–	Yes
An et al. [30]	Anomaly	Principal component analysis, 1-class SVM, anomaly detector on n-grams	System calls	Virtual Linux	Unspecified virtualized WiFi router	1	100.00	0.00	Yes
Su et al. [31]	Classification	2-class convolutional neural network	Malware binaries from IoTPOT	–	–	–	93.33	5.33	Yes
Mudgerikar et al. [32]	Classification	Random forest, white-listing	Processes, system calls	–	Web cameras	–	100.00	0.00	Yes
Wang et al. [33]	Classification	XGBoost and LSTM neural networks	System calls	Linux	HiSilicon Hi3516CV300	1	98.10	2.80	Yes
Gassais et al. [34]	Classification	Decision trees, SVM, neural network	System calls, network data	Linux	Raspberry Pi 3 with ARM, Multisensor Z-Stick Gen5	2	99.99	0.01	Yes
Rehman [35]	Classification	Convolutional and recurrent neural networks	CAN-bus frame data	–	–	–	95.30	5.12	Yes
Jeon et al. [36]	Classification	Bidirectional LSTM and spatial pyramid pooling network	Malware binaries from KISA-data challenge 2019	–	IoT devices in the healthcare industry	–	92.33	5.24	Yes
Cosson et al. [37]	Classification	Naive Bayes, random forest, various neural networks, logistic regression	CPU and memory used, # of processes, # of file descriptors open, etc.	Linux	9 Raspberry Pi 4 B (4GB RAM) with various smart home devices attached: light bulb, smoke detector, smart TV, thermostat, weather station, etc.	9	95.00	0.02	Yes
Alasmary et al. [38]	Classification	Logistic regression, random forest, and deep learning (using NLP-based features)	Network traces and shell commands (extracted from IoTPOT malware binaries and benign binaries)	Linux	–	–	99.70	0.07	Yes
Ngo et al. [39]	Classification	Neural networks, Gaussian regression, random forest, SVM	Printable string information graphs extracted from binaries in IoTPOT and VirusShare malware; benign executables extracted from IoT firmwares	Linux	–	–	96.30	7.90	Yes
Tamas et al. [40]	Misuse	Similarity hash functions	Malware binaries provided by Ukatemi Technologies; benign binaries from Ubiquity and D-Link firmwares	Linux	–	–	90.00	0.00	No
Li et al. [41]	Classification	Convolutional neural network	Custom dataset of binaries	–	–	–	98.47	1.21	Yes
Ding [42]	Classification	Deep learning (autoencoder)	Custom dataset of power traces collected by a proposed HW sensor	Linux	IP cammeras D-Link (D-934L) with MIPS, ESCAM(E-G02) with ARM, Xiaofang (X-1S) with MIPS	3	98.70	1.30	Yes
HADES-IoT	Anomaly	White-listing	System calls	Linux	Various Wifi Routers and IP cameras with ARM, MIPS, MIPSell	15	100.00	0.00	No

be successful in the IoT domain. HADES-IoT falls in this category.

A lightweight anomaly based approach was proposed by Yoon *et al.* [27], who based their work on the distribution of system call frequencies in Linux-based IoT devices. By utilizing cluster analysis, the authors first learn the benign execution context, and then a device is monitored in real time to detect anomalies. However, the authors only considered attacks that alter system calls in benign programs. A lightweight IDS that focuses on smart meters was proposed by Tabrizi and Pattabiraman [28]. This research is based on modeling benign system call sequences with a finite state machine (FSM) model. In FSM, comparing the current system calls with the model is very resource intensive, and therefore the authors trigger their approach just once every 10 s, and moreover they only filter specific system calls. In this study, the evaluation is performed on just a single device. Agarwal *et al.* [29] presented an anomaly detection system approach in which control flow path modeling of system calls is performed using the Ball–Larus path profiling. This study includes just a preliminary evaluation of the overhead imposed on two programs—*tcpecho* and consumer health IoT gateway, and the detection performance is not reported. An *et al.* [30] proposed behavioral anomaly detection of IoT malware using three semi-supervised algorithms (i.e., principal component analysis, one-class SVM, and a naïve detector based on unseen *n*-grams). The approach inputs captured kernel-level system calls to determine whether a device has been compromised. The authors evaluate classifiers on two IoT malware families—MrBlack and Mirai. The results show that all three algorithms employed achieved a 100% detection rate with a low FPR, but the overhead inflicted by the approach is not evaluated. The downside of the approach is that full kernel recompilation with enabled *ftrace* support is required.

Classification-Based Approaches: All of the approaches in this category require both legitimate and (a variety of) malicious data samples to train a supervised classifier. We argue that the requirement of the malicious samples can be viewed as a downside, since the classifiers can only recognize new malware that is similar to known malicious samples or different from known legitimate samples—common examples are adversarial attacks [46].

Su *et al.* [31] presented a lightweight image recognition technique for malware classification. The proposed approach transforms a program's binary into images of 64×64 pixels. Such images then serve as input to a two-class convolutional neural network that determines whether the analyzed program is malicious or benign. To evaluate the performance, the authors used malware captured by the IoTPOT honeypot [47]. They achieved 94% accuracy for two-class classification and 81% accuracy for three-class classification (i.e., benign, Mirai, or Gafgyt). A similar approach was proposed by Li *et al.* [41], who achieved an F1-score of 98.62% on a custom data set with a combination of IoT and PC-based malware. However, these approaches are susceptible to complex code obfuscation. Mudgerikar *et al.* [32] proposed a system-level intrusion detection system for IoT devices called E-Spion. E-Spion relies on a three-layer baseline module: 1) a process whitelisting module; 2) a process behavior module; and 3) a system call behavior module. The baseline profile for each IoT device is built using three-layer system-level logs recorded by the system monitor module running on the device. The authors reported detection efficiency ranging from 78% to 100% depending on the layers of detection employed. Wang and Lu [33] proposed a network-based IDS that couples extreme gradient boosting and a long short-term memory (LSTM) neural network for attack classification based on system call data collected on IoT devices. The proposed stacked model identifies abnormal behavior hidden in the system call sequences, and according to the authors, it achieves an AUC score of 0.983 with data collected from one IP camera. Gassais *et al.* [34] proposed a host-based intrusion detection framework for smart devices, which combines information from the user space and kernel space using LTTng and machine learning techniques. The authors tested the proposed solution with a home automation system based on a Raspberry Pi with various attacks. Javed *et al.* [35] dealt with intrusion detection on the controller area network (CAN) by analyzing messages exchanged on the CAN bus. The authors employed a deep learning model containing convolutional and recurrent neural networks to detect fuzzy, DOS, and impersonation attacks. However, the authors do not report the performance overhead (training or operation) or the costs of their solution in terms of additional hardware. Jeon *et al.* [36] suggested a hybrid intrusion detection system for IoT devices, which utilizes features of static and dynamic analysis to train supervised neural network classifiers. The system employs a bidirectional LSTM and spatial pyramid pooling networks, and the authors report classification accuracy of over 92%. Cosson *et al.* [37] proposed SENTINEL, an intrusion detection system for IoT networks. SENTINEL monitors and collects kernel-level information on IoT devices, such as CPU usage, RAM usage, and total load. The authors tested their approach against five types of IoT attacks, such as data exfiltration, a black hole attack, and network scanning. The results show that their approach achieves over 96% of accuracy while keeping resource consumption at a minimum. Alasmary *et al.* [38] described an approach of detecting attacks on IoT devices based on collected Linux shell commands issued by the attacker. The authors used deep learning classifiers on NLP-based features extracted from shell commands and network traces of the IoTPOT data set. The authors report obtaining an F1-value of over 99.9%. However, there is a chance that the detection performance of this approach might degrade due to string obfuscation techniques, which are often seen in malware. In contrast, HADES-IoT does not perform any binary analysis and therefore is not prone to such obfuscations. Ngo *et al.* [39] introduced an approach for the detection of IoT botnets using printable string information graphs extracted from malware binaries on the IoTPOT and VirusShare data sets. The authors used several classifiers, such as neural network, random forest, and SVM classifiers, and they report a true positive rate (TPR) of 96.3% with an FPR of 7.9%. Ding *et al.* [42] proposed DeepPower, a method for deep learning-based detection of IoT malware that uses a power side channel. DeepPower utilizes a new hardware sensor connected to an IoT device, which collect data and sends it to the processing server. The server converts the power signals to Linux shell commands and then utilizes them as input for a deep learning model that detects malware. The authors achieved ∼90% accuracy for malware detection on three different IoT devices.

Misuse-Based Approaches: In this category (also referred to as signature-based approaches), only malicious samples are needed to train the intrusion detection model. A disadvantage of these approaches is that they can only detect known attacks, which are usually represented by signatures.

Tamás *et al.* [40] presented a signature-based intrusion detection method for IoT devices, called SIMBIoTA. SIMBIoTA uses similarity hash functions to create a database of signatures, which is then utilized on IoT devices to detect malware. The authors evaluated the detection capabilities of SIMBIoTA and achieved a TPR of 90% and an FPR of 0%. A disadvantage of SIMBIoTA is that requires periodic signature updates of new malware samples from remote sources, which, however, is the inherent downside of all misuse-based intrusion detectors.

A. Comparison With HADES-IoT

Here, we highlight the unique features of HADES-IoT in contrast to related work.

Extra Detection Device: Based on our analysis of existing literature, we found out that all prior studies[3] proposed approaches for intrusion detection on IoT devices require an extra machine to perform the detection itself, while IoT devices only export collected data to such an extra machine (see the last column in Table I). Therefore, all of the proposed approaches introduce a single point of failure for the entire monitored network and moreover impose an additional cost for the required hardware and machine operation. In contrast, HADES-IoT operates autonomously on an IoT device and does not require any additional machine or hardware to perform analysis, detection, or signature updates, which makes it more distributed and thus failure resistant.

Software Dependencies: It is noteworthy that the majority of research on IoT-oriented host-based intrusion detection systems relies on system calls collected using tracing and system monitoring tools, such as *strace*, *ftrace*, *atop*, and *LTTng*, which might not be present on IoT devices. Instead of such constraining dependencies, HADES-IoT is implemented as a dedicated kernel module and works independently of installed libraries and tools.

Proactive Real-Time Detection and Prevention: HADES-IoT's unique capability is that it intercepts execve system calls in real-time, which enables it to proactively detect and prevent (potentially zero-day) malware from even being executed. In contrast, other approaches require the attack to be executed first, and then they assume that the collected data is delivered to a data processing machine.

III. Problem Statement

The objective of this work is to design and develop a security solution that protects the bulk of existing IoT devices from malware that remotely exploits their vulnerabilities. According to surveys performed by Eclipse IoT [48], [49], Linux is the most common OS of IoT devices, and therefore we target Linux-based IoT devices in this study.

Usually, the exploitation of vulnerabilities in IoT devices is accomplished by password brute forcing of services, such as Telnet or SSH. The creator of Sora and Owari IoT malware [50] claims that Telnet is abused by password brute-forcing to such a great extent that attackers are competing with each other to exploit more IoT devices. Consequently, more advanced attackers have shifted their focus to the exploitation of services that are not as heavily exploited. Such advanced attackers have begun to integrate exploit scanning tools in their malware. This provides them with an additional attack vector (beyond Telnet misuse) and improves their ability to compromise vulnerable IoT devices, despite the devices being protected with strong passwords. For example, VPNFilter takes advantage of 19 vulnerabilities, enabling this malware to compromise about 70 models of ten different vendors [16].

A. Attacker Model

We assume that a Linux-based IoT device protected by our approach is connected to the Internet with a public IP address. Therefore, the attacker is able to discover the device using her scanner or a public service such as Shodan.[4] The attacker can scan the IoT device to reveal open ports and identify running network services that contain known or zero-day vulnerabilities. Next, the attacker can remotely exploit a vulnerability on the IoT device. Specifically, we assume that the default executables of the IoT device may contain a vulnerability enabling the execution of arbitrary binaries, i.e., binaries that already exist on a device or those that were delivered by the attacker (such as in VPNFilter and IoT Reaper). To accurately reflect real-world attack scenarios, we further assume that once the attacker "is inside" the IoT device, she is granted superuser privileges. However, we do not assume potential vulnerabilities in the OS kernel, and therefore they are out of the scope of this article. Covering those vunlerabilities requires in-depth understanding of the Linux kernel in terms of ensuring proper functionality, stability, etc., without compromising performance significantly.

Furthermore, we assume that: 1) all of the default executables installed on the IoT device are benign and 2) the attacker does not tamper with the device before or during the deployment of our proposed approach. Finally, we assume that our approach is the only security solution deployed on the IoT device, since it is not common practice for manufacturers to enable security features (such as SELinux and Auditd) either for performance reasons or due to the complexity of configuring them.

B. Requirements for IoT Defense Solution

In this section, we specify desirable properties and requirements for a host-based IoT defense solution that can handle the above-mentioned attacker model. In particular, a defense solution should meet the following requirements.

1) *Real-Time Detection:* Since we aim to prevent all unknown actions on an IoT device, the defense solution must be capable of detecting any unknown program upon its execution.
2) *Low Overhead:* IoT devices are extremely resource constrained and thus provide only limited processing and

[3]Note that the only method proposed in prior work that does not require an external machine for the detection itself is that of Tamás *et al.* [40]. However, the external machine is required for regular signature updates.

[4]https://www.shodan.io/

TABLE II
IoT DEVICES USED IN THE EVALUATION

Device	**Type**	**Kernel Version**	**CPU Architecture**	**CPU Performance** $[BogoMips]$	**Total Memory** $[MB]$	**Available Memory** $[MB]$
NETGEAR WNR2000v3	Router	2.6.15	MIPS	265.2	32	16.06
ASUS RT-N16	Router	2.6.21	MIPSel	239.2	128	87.22
ASUS RT-N56U	Router	2.6.21	MIPSel	249.3	128	78.56
Cisco Linksys E4200	Router	2.6.22	MIPSel	239.2	64	18.63
D-Link DCS-942L	IP Camera	2.6.28	ARM	534.5	128	38.75
SimpleHome XCS7-1001	IP Camera	3.0.8	ARM	218.7	32	1.90
Provision PT-737E	IP Camera	3.4.35	ARM	218.7	32	3.88

storage resources. Therefore, it is not possible to utilize conventional security approaches used in PC environments (e.g., machine learning or complex heuristics). With this in mind, the defense solution should be conservative in terms of resource consumption and should only utilize existing dependencies, such as already installed libraries.

3) *Tampering Protection:* Since the attacker has superuser privileges, she can terminate or bypass a defense solution deployed on an IoT device. Therefore, the defense solution should be resilient against such tampering.
4) *Wide Coverage:* It is important to protect a wide range of IoT devices (e.g., printers, IP cameras, and Wi-Fi routers), taking into account the fact that a significant portion of existing IoT devices are already considered legacy and moreover may not receive manufacturer updates.
5) *Independence:* The deployment of the defense solution must not be dependent on the manufacturer; both the user and the manufacturer should be capable of deploying the defense solution.
6) *Ease of Bootstrapping:* With regard to the deployment of the defense solution mentioned above, we further argue that the defense solution should be deployable with minimal effort and should not require recompilation of the kernel of the IoT device's OS.

C. Design Problems and Design Options

Based on the specified requirements, we analyzed various strategies for developing a defense solution and identified additional constraining factors. Initially, we conjectured that the most straightforward option would be to utilize features provided by Linux, such as *KProbe*[5] or *inotify*.[6] This would provide us with the support of the Linux kernel and enable us to utilize existing features. However, after examining several IoT devices (see Table II), we found that those features are not supported by any of them.

Low Resource Utilization: Other design aspects we considered are the lightweight complexity and low resource requirements of the defense solution. The most important resources for a defense system are the CPU and memory. We measured the normal utilization of these resources by the IoT devices examined in this study and found out that while there is a reasonable reserve of CPU utilization, the CPU performance is often low. Therefore, we must ensure that the defense solution only increases the CPU consumption to the extent needed; a larger increase in the CPU consumption could affect other applications, which could further lead to performance degradation and device availability issues. We also observed that there is only a small amount of free memory on IoT devices (e.g., less than 2 MB); moreover, not all of the free space can be utilized for a defense solution due to the requirements of other applications that may be running on the device.

Process Scheduler: Since the challenge is to detect unknown processes in real time (when they are spawned), the Linux process scheduler is another limiting factor. In the user-space environment, processes compete for the CPU, and the process scheduler makes decisions regarding the CPU and time allocated for the processes. Therefore, if a defense solution was implemented in the user space, there would be no guarantee that it would be running when a new process is spawned; hence, malicious processes might be missed or detected too late. This issue could be naïvely mitigated by setting the highest priority to the defense solution, so the scheduler would prefer the defense solution over the other processes. However, this solution does not resolve the issue, since the attacker's controlled process possesses the same capabilities as the defense solution and thus has an equal chance of being selected by the scheduler. Therefore, to ensure that no newly spawned processes are missed, the defense solution cannot be dependent on the scheduler and its planning algorithm. Below, we show that an LKM and *libc library hooking* are not dependent on the process scheduler.

System Call Interception: System call interception is a technique capable of addressing the issue of execution priority within the scheduling algorithm; in this case, code with the defender's desired functionality is inserted between the caller's invocation of a system call and the system call itself. Thus, with an appropriate set of intercepted system calls, this technique enables all new processes to be "checked" when they are spawned. In general, there are two options for system call interception. The first option is libc library hooking, which is performed in the user space, and the second option is the *interception of system calls through an LKM* running in the kernel space. Although libc hooking is more straightforward in terms of development than the LKM technique, it is not suitable for our case. This is due to the diversity of IoT device customization, where each manufacturer

[5]https://www.kernel.org/doc/Documentation/kprobes.txt
[6]http://man7.org/linux/man-pages/man7/inotify.7.html

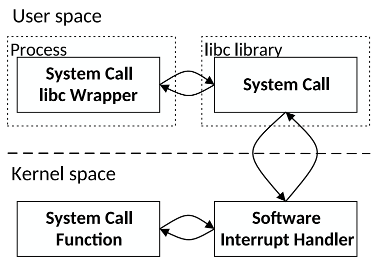


Fig. 1. System call execution flow.

uses its own custom Linux that can only be compiled with specific (or custom) libraries and their versions (including libc). This renders libc hooking an unusable option for the detection of the majority of IoT malware.[7] Therefore, we identify the LKM technique as the most suitable option for our purposes.

IV. Background and Preliminaries

Based on the analysis above, we decided on the system call interception option that utilizes the LKM technique. In this section, we define preliminaries, provide background on the LKM technique, and system calls, as well as provide a few examples.

A. System Calls

In Linux systems, every process starts in a nonprivileged mode [51]. In this mode, a process is restricted and only capable of using the memory address space assigned to it by the OS. Access to the memory space of other processes, as well as access to the kernel, are thus prohibited and result in an exception being raised. Therefore, when a program wants to access a resource outside of its allocated memory space (e.g., to read a file or execute another process), it must perform a call to the kernel, which is done by invoking a system call. Currently, the Linux kernel provides the user with more than 300 different system calls [52]. Upon the invocation of a system call, a software interrupt is triggered, which results in switching to the *system mode*. In the system mode, the process is granted superuser privileges, which enables it to perform restricted actions. Next, the requested operation is performed by the OS kernel, ensuring that the user-space process does not interfere with the restricted resources.

System Call Execution: The system call execution flow is depicted in Fig. 1. When a process invokes a system call, in most cases it does not invoke the system call directly but rather invokes one of the wrappers provided by the libc library. These wrappers may perform other actions, such as preparing the received data for the system call. For example, the libc library provides six different wrappers for *execve* [53]. Once the data are prepared, the wrapper calls the system call through a software interrupt that transfers control to the kernel. The interrupt is caught by the software interrupt handler which may perform further actions (e.g., save a processor's registers).

[7]Note that attackers address these issues by compiling their malware statically (i.e., all required libraries are included in the malware's binary).

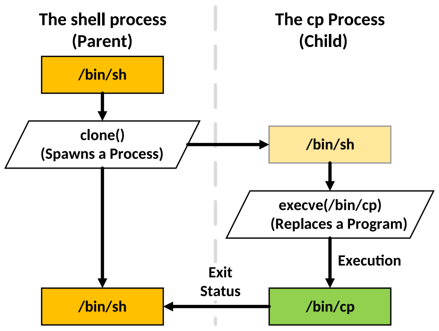

The shell process (Parent)
The cp Process (Child)
/bin/sh
clone() (Spawns a Process)
/bin/sh
execve(/bin/cp) (Replaces a Program)
Execution
Exit Status
/bin/sh
/bin/cp

Fig. 2. Example of spawning a new process *cp*.

Afterward, the handler looks into the system call table to find the address of the pertinent system call and jumps into that address, initiating the system call's execution. Finally, when the system call has been completed, the result is propagated back to the calling process.

System Calls That Spawn New Processes: The Linux kernel provides three system calls that spawn new processes.

1) *Fork():* Upon its invocation, *fork* creates a new process (i.e., child) by duplicating the calling process (i.e., parent). Immediately after a *fork* call, the parent and child run in separate memory spaces, but the spaces contain the same content.
2) *Vfork():* Similarly to *fork*, *vfork* creates a child process of the calling process, but in contrast to *fork*, the child and parent share the memory space after invocation thus saving computational and memory resources. In *vfork*, the parent is suspended until either the child terminates normally or it calls the *execve* system call. Therefore, *vfork* is often used in performance-sensitive programs, in which a child immediately makes a call to *execve* after *vfork* has been invoked.
3) *Clone():* In contrast to the previous system calls, *clone* can create a new thread in addition to a new process. The *clone* system call is more versatile than *fork*, and thus, libc library implementations like *glibc* provide a *fork* wrapper that internally calls *clone*.

A System Call that Replaces a Program's Code: Execve() is a system call that does not have the ability to create a new process, but it often participates in process creation. When a process calls *execve*, *execve* executes another program whose path was passed as one of the arguments. The program of a calling process is thus replaced with new program code, and the stack, heap, and data segments are newly initialized. Although *execve* can be called by a process at any time, it is usually called after one of the aforementioned system calls. An example of process spawning is depicted in Fig. 2, where the Unix utility *cp* is executed. First, the shell process is executed, which then calls *clone*. *Clone* creates a copy of the calling process itself (i.e., a shell). Next, this copy calls *execve* with *cp* and its parameters as arguments. Once *execve* successfully returns, the code of a new process is replaced by the *cp* code. When the *cp* process terminates, its return code is passed to the parent process (i.e., shell) to indicate whether *cp* terminated successfully or with an error.

B. Loadable Kernel Modules

An LKM is an object file that can be installed in a Linux kernel to extend the kernel's base functionality, e.g., LKMs might provide drivers for peripheral devices. The installation of an LKM is performed at runtime, and once an LKM is installed, its functionality is integrated in the kernel. The advantage of LKMs is that they can be inserted or removed from the kernel at any time, without the need to recompile the kernel.

Compilation and Installation of an LKM: Since an LKM is integrated into the Linux kernel upon insertion, the LKM must be compatible with that kernel version. An LKM is always compiled against the matching Linux kernel version of the targeted device. In addition to the kernel version, the configuration of the kernel features and options must match as well (e.g., ext4 support and kernel debugging). In contrast to PCs where such data can be directly extracted, the situation is more difficult for IoT devices, since their OS is usually provided in a minimized and customized form. As a result, many necessary files are missing, including the configuration file. Therefore, the necessary information must be parsed from system files (e.g., */proc/kallsyms*). Upon insertion of an LKM, the Linux kernel checks its compatibility with the LKM using information extracted from the LKM's binary. However, this check does not cover all critical parts that must match. Although the LKM may pass these checks and be installed in the kernel, there is still no guarantee that it will work properly within the kernel. For this reason, it is crucial for the LKM to match the configuration of the kernel to the greatest possible extent.

V. Proposed Approach

We propose HADES-IoT, a host-based anomaly detection system aimed at Linux-based IoT devices. This solution fulfills most of the requirements specified above (see Section III-B), since we adopted the LKM approach which utilizes the system call interception technique and, more specifically, intercepts the *execve* system call. Using the LKM approach allows us to install HADES-IoT into a Linux kernel at any time, without the need to recompile the kernel. The only requirement is that HADES-IoT needs to be distributed in precompiled binaries (see Section VII-A).

HADES-IoT is based on a whitelisting approach, which means that it only allows the device to run benign programs that are known to be present and actively used on an "uninfected" off-the-shelf device. However, to build a white list of benign programs, profiling must be performed once for each device model. During profiling, the device operates normally for a certain amount of time, while HADES-IoT captures all executed binaries. This may be considered insufficient due to the possibility that some benign programs may be missed during profiling. Therefore, HADES-IoT includes a feature that copes with this situation and allows the white list to be updated at runtime (see Section V-E2).

A. Deployment

First, HADES-IoT needs to be deployed on the kernel. During deployment (see Fig. 3), HADES-IoT is precompiled

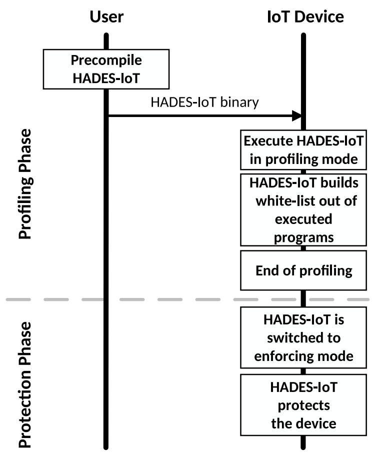

User
IoT Device
Precompile HADES-IoT
HADES-IoT binary
Profiling Phase
Execute HADES-IoT in profiling mode
HADES-IoT builds white-list out of executed programs
End of profiling
Protection Phase
HADES-IoT is switched to enforcing mode
HADES-IoT protects the device

Fig. 3. Deployment of HADES-IoT.

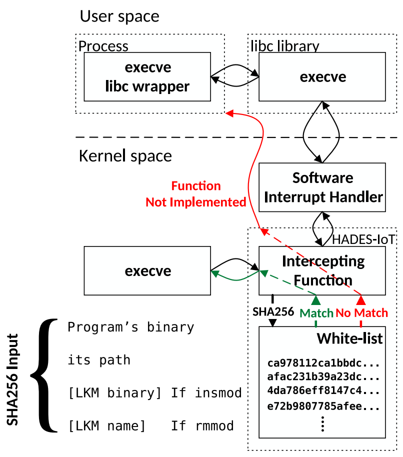


Fig. 4. *Execve* system call execution flow with HADES-IoT installed in the kernel.

and delivered to the device, and then the kernel's initialization file is modified accordingly to ensure that HADES-IoT is always executed when the device is booted.

After deployment, HADES-IoT enters the *profiling mode*, where it monitors and collects information about all calls to *execve*, and the white list is updated. The profiling mode ends when no new processes are detected for a specified period of time. We emphasize that during profiling, the device needs to be restarted, which enables the white list to be updated with all of the programs executed at boot time. In the second phase of deployment, HADES-IoT switches to the *enforcing mode*, in which it starts to protect the device using the white list. As mentioned above, we assume that HADES-IoT is not compromised before or during its deployment; this is a standard assumption, since deployment is a one-time action.

B. Detection Process

The most important feature of the detection process is the intercepting function. The interception process is depicted in Fig. 4. Upon deployment, HADES-IoT locates the system call table and saves the address for *execve* that appears in the table. Next, *execve*'s address in the table is replaced by the address of the intercepting function. This ensures that each time *execve* is called, the software interrupt handler calls the intercepting function instead of the original *execve*. Once the intercepting function is executed, it first reads the parameters passed to the *execve* system call (i.e., the path of the program to be executed). Next, the function computes a SHA256 digest from the program's binary contents, its path, and other data, depending on the specific circumstances (see Section V-C). Using the digest computed, the intercepting function looks for a match in the white list of all authorized programs. In the case of a match, the process is allowed to run, and therefore the intercepting function performs a call to the original *execve* system call. However, in the opposite case, the intercepting function returns an "*-ENOSYS*" error code, which tells the process that the *execve* system call is not implemented. This naturally terminates the process, thus stopping the execution of any unauthorized (e.g., potentially malicious) action.

Restoration of CPU Context: It is important to note that the Linux kernel is not aware that HADES-IoT changed the address of the *execve* in the system call table to the address of the intercepting function. Therefore, when *execve* is called by a process, the environment is prepared for the execution of this system call. This means that the intercepting function must act transparently, and after performing the authorization check, it must restore any tainted processor register to its original state. Otherwise, this might lead to an inconsistent state, causing the kernel to freeze or crash.

C. White List Design

After HADES-IoT is successfully deployed, each call to the *execve* is intercepted. During interception, HADES-IoT searches the white list for a program that is requested to run. An inefficiently designed search process would cause the IoT device to have a slow response time, particularly when the device has a large number of periodically spawned processes. For example, if the white list was naïvely designed as a linked list, the asymptotic time complexity of the search routine would be $O(n)$. This means that large white lists would cause longer search delays than small white lists. To cope with such delays, we designed the white list as a hash table, which provides a constant asymptotic time complexity $O(1)$ for a search routine, meeting the low overhead requirement.

IDs in the White List: Each item in the white list contains an ID and represents a program authorized to run on a device. The ID of an item is an SHA256 digest computed from a program's binary, concatenated with the path to the program and, in certain circumstances, concatenated with other additional data. The reason for this computation is to distinguish symbolic links from the executables they point to. BusyBox[8] is an example of such an executable that is heavily utilized in Linux-based IoT devices. BusyBox combines a set of common Unix utilities under a single executable, while particular utilities are accessible through symbolic links. Hence, if we were to compute the digest out of just the binary content of the passed program, we would obtain the BusyBox digest for all of the utilities. However, after adding a path element to the digest computation, the resulting digest is different for each of the utilities, which resolves the problem.

Nevertheless, there are cases in which this approach is insufficient. Such cases require a more fine-grained whitelisting, in which additional context-dependent data must be added to the SHA256 input (see Sections V-D2 and V-D3).

D. HADES-IoT's Tamper-Proof Features

We assume that the attacker is provided with superuser privileges once the IoT device has been compromised. Therefore, we need to ensure that the attacker, who is aware of the presence of HADES-IoT, cannot terminate or modify it. Below, we list possible attacks and describe how HADES-IoT protects itself against tampering.

1) Binary Manipulation: When an IoT device boots, HADES-IoT is loaded from its binary to the kernel. Therefore, a possible attack involves deleting/moving this binary, thereby preventing the installation of HADES-IoT. Moreover, instead of deleting or moving the binary file, the attacker can attempt to tamper with HADES-IoT's binary. For example, the attacker might modify the kernel version included in HADES-IoT, causing the kernel to refuse its installation. Further, the attacker could corrupt the HADES-IoT binary, which could result in a permanent DoS due to the crashing or freezing of the kernel.

Protection: To prevent a manipulation of HADES-IoT's binary by the attacker, HADES-IoT loads its binary into the memory as its first action upon booting. Similarly, when a restart of a device is requested, HADES-IoT's binary is (re-) written to the storage as its last action before the restart (regardless of whether the original version was modified or not). Therefore, any malicious modification, removal, or movement of the original binary is prevented.[9]

2) Loading Malicious LKMs: A common practice for IoT devices is to install a few LKMs at boot time. These LKMs usually represent drivers. The utility for installing LKMs (i.e., *insmod*) is then included in the HADES-IoT white list. This means that the attacker can exploit the insmod utility to install her own LKM that "overrides" *execve* interception with a malicious callback function and thus effectively take HADES-IoT out of the game.

Protection: Each execution of *isnmod* must be verified against the allowed and known executions, requiring more fine-grained indexing of the white list. In contrast to the white list indexing of standard binaries, in the case of *insmod*, we compute the index of the white list as a cryptographic hash from: 1) the binary content of *insmod*; 2) its path; and 3) the binary content of an LKM requested to be loaded to a kernel. This ensures that only known LKMs from the profiling phase are allowed to be loaded again.

3) HADES-IoT Uninstallation: Since IoT devices might install several LKMs at boot time (e.g., drivers), they might also uninstall some LKMs during runtime. When this occurs,

[8]https://busybox.net/

[9]Note that this procedure also ensures that the attacker does not deplete the free space in the storage before writing the HADES-IoT binary. This can be achieved by various means (e.g., keeping the list of all known files in the memory and deleting unknown files when there is no free space left).

the utility for uninstalling LKMs (i.e., *rmmod*) is included in the white list. This means that without proper protection, both the drivers and HADES-IoT itself could be uninstalled by the attacker.

Protection: The prevention technique for this attack is almost the same as in the previous case. The only difference is that instead of adding the binary content of the LKM to the hash computation, it is sufficient to use the name of the LKM, as the name unambiguously identifies a kernel module that has already been installed. On the other hand, this solution does not prevent the attacker from uninstalling a kernel module already included in the white list and thus disable some functionality of an IoT device. Although it might affect only a few particular types of IoT devices, we consider this issue out of the scope of this study, and we plan to address it in future work.

4) Init Script Manipulation: Another possible attack is a modification of the *init* script. The *init* script is executed when an IoT device is booted, and it contains commands to configure and prepare the device for use. The command to install HADES-IoT's kernel module is one of the commands executed during booting. Therefore, if the attacker manages to remove the command from the *init* script and subsequently restarts the device, HADES-IoT will not be installed at boot time.

Protection: To prevent this attack, we propose loading the *init* script into the memory of HADES-IoT at boot time. Therefore, when an IoT device reboots, the script is rewritten to the persistent storage, regardless of whether it was modified or not.

5) Memory Tampering: Memory tampering is another potential attack. However, to prevent it, HADES-IoT takes advantage of the fact that it is integrated in the kernel's memory, and user-space programs cannot reach the kernel. The only chance for the attacker to tamper with HADES-IoT's memory is if the attacker gets into the kernel space as well. Countermeasures that prevent the attacker from loading anything into the kernel were presented in Section V-D2.

E. Extensions

The core functionality of HADES-IoT presented in this article effectively detects and terminates any unauthorized process spawned on a protected IoT device. However, in its current form, HADES-IoT may be difficult for users to operate. Therefore, we present extensions that we have incorporated into HADES-IoT aimed at both making it more convenient to use and further improving its detection capabilities.

1) Reporting Subsystem: Although HADES-IoT protects an IoT device, the owner of the device should be informed about any attempted attack, so she can perform further actions. Therefore, we introduce a reporting subsystem that informs the owner when an attack has been detected. The reporting subsystem is optional and contains:

1) an application running on the owner's machine;
2) a user-space application which is deployed, along with HADES-IoT, on an IoT device.

Once an attack attempt is detected, HADES-IoT immediately notifies the reporting subsystem, which then forwards this information to the owner of the device.

2) Remote Control and White List Updates: When an IoT device is updated with new firmware, HADES-IoT must be redeployed and reprofiled (see Section VIII-A). In the case of reprofiling, HADES-IoT must first be terminated (i.e., uninstalled). However, such termination and reprofiling must be done in a secure way to prevent the attacker from taking advantage of it. To address this issue, we further extended the user-space application described previously. This extension allows the application to listen on a specified port, enabling the user to connect through the remote control application to the listening port. After connecting, the user must authenticate herself in order to issue certain commands (e.g., stop, start, and profile). Moreover, with the help of the remote control application, the user can add new entries to the white list if needed.

Also, the remote control enables the user to update the IoT device as follows. First, the user stops HADES-IoT and performs the update. Once the update is done, the user instructs HADES-IoT to run in the profiling mode, which causes the white list to be rebuilt. After reprofiling, the user sends a command to turn the protection on by switching to enforcing mode again.

Since IoT devices rarely provide dependencies for asymmetric cryptography, we propose using the Merkle signature scheme [54] to authenticate messages sent by the owner. The Merkle signature scheme only requires a cryptographically secure hash function, and moreover it provides resilience against quantum computing attacks (see details in Section VI).

3) Signal Monitoring: In Linux, signals are software interrupts that inform a process about an event that has occurred. For each signal, there is a defined default action that a program performs once the signal is received (e.g., stop, continue, or terminate). A signal can be sent to a process by invoking the *kill* system call. Another option is to use the *kill utility*. Since sending a signal to a process can lead to its termination, the attacker can take advantage of this by sending the *SIGKILL* signal to a service, such as *lighttpd* and cause a DoS attack. However, when an IoT device is protected by HADES-IoT, the attacker cannot execute any malicious program, since it would be stopped. Therefore, using the kill utility is the attacker's only option. According to the POSIX standard, the kill utility should always be provided as a standalone binary. If the kill utility is never part of the normal profile of an IoT device,[10] HADES-IoT will also detect such a process termination attempt and prevent it from happening. Note that the kill utility is just a wrapper for the kill system call, however some shells call a built-in function that directly invokes the kill system call instead of executing the kill utility. Therefore, the malicious termination of a process would not be detected. To resolve this issue, HADES-IoT has to intercept the kill system call. That allows HADES-IoT to detect an invocation of the kill system call, and if it is not authorized, its invocation is prevented.

[10]This is true for all of the IoT devices examined in this work.

VI. Details on HADES-IoT Remote Control

First, we briefly describe Lamport one-time signatures and their aggregation by the Merkle signature scheme. Then, we explain the integration of this scheme into HADES-IoT for the purpose of authenticating messages sent to the remote control application. Finally, we provide a detailed description of HADES-IoT's deployment with this scheme and a use case representing a secure firmware update.

A. Lamport One-Time Signatures

The Lamport signature scheme [55] is a quantum-resistant construct of asymmetric cryptography, which serves to authenticate a single message. The private key is generated by a cryptographically secure pseudorandom number generator (CSPRNG). The private key consists of a pair of K numbers, each of which is K bits long, where K represents the output size of a cryptographically secure hash function $h(.)$. Therefore, the size of the key is $2K^2$ bits (e.g., if $K = 256$, the size of a private key is 16 kB). Next, the public key is computed from the private key by computing a hash of each number in the key, obtaining the same size and pairwise structure as the public key, as in the case of the private key.

The signature of a message m is created by selecting K numbers from a private key as follows: for each pair of numbers at position $i = \{1, \ldots, K\}$, one number is selected according to the value of a bit at position i in the hash $h(m)$ computed from the message m. More specifically, the first number of the private key at position i is selected if the ith bit of $h(m) = 1$, while the second number is selected otherwise. Hence, a signature contains K numbers, and the resulting size of the signature is K^2 bits (e.g., if $K = 256$, the size of a signature is 8 kB).

Verification of the signature associated with $h(m)$ starts by selecting one number from each number pair of the public key according to the value of each bit in $h(m)$. Then, the obtained K values are compared to the K hash values computed from the signature. The signature is valid in the case of a match; otherwise, it is invalid.

B. Merkle Signature Scheme

Merkle signatures [54] extend one-time signature schemes (such as Lamport signatures [55]) to support multiple messages. More specifically, N public/private key pairs of one-time signatures are generated by a cryptographically secure pseudorandom function $F(S||i)$, where S represents the secret seed, $i = \{1, \ldots, N\}$, $||$ represents string concatenation, and N is equal to an arbitrary power of two that is large enough for the particular application (e.g., we assume $N = 2^{15}$ in our case).[11] Next, the hash value is computed from each public key, and then these N hashes are aggregated by a Merkle tree into a root hash, which represents a master public key associated with all of the leaf public keys and their corresponding private keys. The size of the root hash is 32 B.

The signing of a message m is performed as described in the previous section, but the signature is also extended by the (Merkle) proof[12] associated with a particular leaf in the Merkle tree. The proof consists of $log_2(N)$ hash values and indications of their left/right positions within the Merkle tree. Given the proof and a particular leaf public key, it is possible to verify whether this leaf public key is present at a particular position of the Merkle tree by computing the root hash value. If the root hash value computed matches the (master) public key, verification is successful. In sum, the size of the signature is $K^2 + K\log_2(N)$ bits (e.g., if $K = 256$ and $N = 2^{15}$, then the signature size is 8.67 kB).

Verification of the signature associated with the message m has two steps: 1) verification of the Lamport signature is performed (as described in Section VI-A) and 2) if the Lamport signature is correct, then verification of the expected public/private key pair is performed by the derivation of the root hash, which is compared to the expected root hash value (i.e., master public key).

C. Integration With HADES-IoT

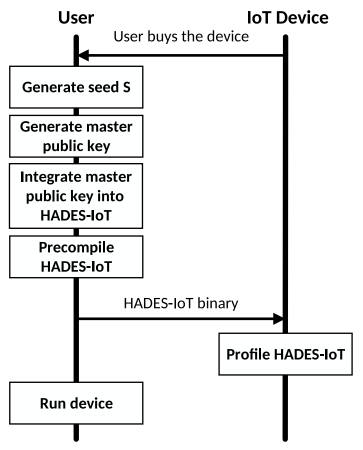
The Merkle signature scheme is integrated with HADES-IoT, as this scheme has only minimal requirements regarding the SW dependencies available on an IoT device—the only requirement is a cryptographically secure hash function. In the case in which HADES-IoT is deployed by a manufacturer (see Fig. 5), the secret seed generation is performed by the manufacturer. This may cause an issue related to the secure delivery of S from the manufacturer to the user who requires it to generate all public/private keys and reconstruct the Merkle tree. However, we consider this issues as out of the scope of this article, and we assume that the seed S can be delivered securely. Note that this issue does not exist when HADES-IoT is deployed by the user (see Fig. 5), as the user is the only one who knows S.

When all of the leaf public keys are generated, the master public key is obtained by computing the root hash from all of the leaf public keys, and then it is integrated into HADES-IoT's source code. After the integration of the master public key into HADES-IoT and HADES-IoT's deployment on a device, the device owner (possessing S) can send commands (i.e., messages) to the user-space application running on the device. This application passes the message to HADES-IoT, which verifies the authenticity of the message, and moreover it verifies whether the ID of the message (i.e., the ID of a leaf on the Merkle tree) has been used before; for this reason, HADES-IoT stores the ID of the last valid message; this prevents reply attacks. If the verification of a signature is successful, HADES-IoT executes the command (e.g., adding a new entry to the white list).

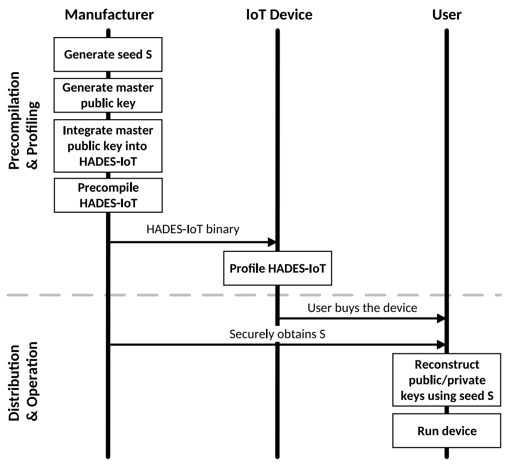
Firmware Update: In Fig. 6, we depict the proposed authentication mechanism for a firmware update performed by the user. First, the leaf secret key SK_i and the leaf public key PK_i are computed. Then, SK_i is used to sign the command that disables HADES-IoT's enforcing mode. The signed command, together with PK_i, the proof π_i, and the ID of the leaf i, are sent to the user-space application that passes them to HADES-IoT's kernel space application. HADES-IoT verifies

[11]The SHA3 hash function is an example of function $F(.)$.

[12]This is also referred to as the *authentication path* or the *authenticator*.



(a)



(b)

Fig. 5. Deployment of HADES-IoT (with integrated Merkle signature scheme). (a) By the user. (b) By a manufacturer.

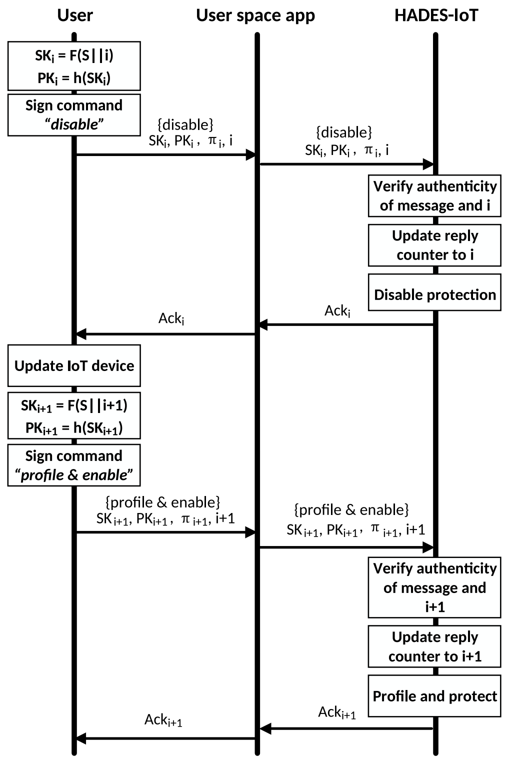

User
User space app
HADES-IoT
SK_i = F(S||i)
PK_i = h(SK_i)
Sign command "disable"
{disable} SK_i, PK_i, π_i, i
{disable} SK_i, PK_i, π_i, i
Verify authenticity of message and i
Update reply counter to i
Disable protection
Ack_i
Ack_i
Update IoT device
SK_i+1 = F(S||i+1)
PK_i+1 = h(SK_i+1)
Sign command "profile & enable"
{profile & enable} SK_i+1, PK_i+1, π_i+1, i+1
{profile & enable} SK_i+1, PK_i+1, π_i+1, i+1
Verify authenticity of message and i+1
Update reply counter to i+1
Profile and protect
Ack_i+1
Ack_i+1

Fig. 6. Firmware update procedure.

the authenticity of the message, using the content delivered and the embedded master public key (see Section VI-B). As part of the verification process, the ID of the message (i) is compared to the reply counter r, and if $i \leq r$, the authentication fails due to the reply attack check; otherwise, the authentication is successful, and r is updated to i. After successful authentication of the message, protection by HADES-IoT is disabled, and the user can execute a firmware update. Once the device is updated, the user sends a signed command instructing HADES-IoT to perform profiling and subsequently enable the protection. The signature is verified in the same way as the previous case, and if successful, the protection is enabled again. Note that we assume that the device is not compromised during the firmware update. Although the likelihood that an Internet-connected IoT device will be compromised during a firmware update is very low, one can avoid potential compromise by disconnecting the device and performing the update from a flash drive.

VII. Evaluation

We implemented a proof-of-concept of HADES-IoT and tested it on seven IoT devices (see Table II). In this section, we start by demonstrating how HADES-IoT can be deployed by a manufacturer, as well as by an end user. We then describe our experiments with various profiling time periods to determine the minimum amount of time required to extract an accurate profile. Then, we evaluate HADES-IoT's detection performance by exploiting vulnerabilities and measuring its resource consumption.

A. Precompilation of HADES-IoT

To ensure that the configuration options and features of HADES-IoT's LKM match the kernel of the target IoT device (see Section IV-B), we determined the minimum set of such features and options needed to run HADES-IoT. First, we selected several IoT devices (i.e., D-Link DCS-942L, Provision PT-737E, and SimpleHome XCS7-1001), for which the manufacturers provided the kernel source code and configuration.[13] With the selected set of devices, we cross-compiled HADES-IoT against the manufacturer's custom Linux kernels using the default configuration provided by the manufacturer. We used the manufacturer's toolchain to compile HADES-IoT, and then we verified its successful deployment. In the next step, we downloaded the source code of the generic Linux kernel[14] for the version matching that of the manufacturer. Then, we selected the default configuration file of the architecture model closest to the architecture model of the IoT

[13]In the case of legacy devices, manufacturers may not provide it.
[14]https://www.kernel.org/

TABLE III
SIZE OF THE WHITE LIST PRODUCED BASED ON THE LENGTH OF THE PROFILING PERIOD

IoT Device	Total # executables found	# of IDs in white-list (1 hour)	# of IDs in white-list (2 hours)	# of IDs in white-list (4 hours)	# of IDs in white-list (1 hour & User Interaction)
NETGEAR WNR2000v3	526	12	12	12	61
ASUS RT-N16	638	5	5	5	38
ASUS RT-N56U	375	3	3	3	6
Cisco Linksys E4200	399	9	9	9	11
D-Link DCS-942L	1256	20	20	20	105
SimpleHome XCS7-1001	588	4	4	4	29
ProVision PT-737E	482	5	5	5	9

device, modified it to match the manufacturer's configuration as much as possible, and compiled HADES-IoT against the kernel. After that, we selectively removed parts of the manufacturer's configuration to determine the minimum number of options that must match in order to successfully deploy HADES-IoT. We found that for all of the tested IoT devices just a few options must match, and more importantly, we found that the information about such options can also be extracted directly from the IoT device.

Afterward, we investigated whether HADES-IoT can be deployed on an IoT device without the support of the manufacturer. In order to accomplish this, we used a publicly available generic Linux kernel and toolchain for precompilation. We were able to precompile and deploy HADES-IoT with generic resources for all of the tested IoT devices, which indicates that HADES-IoT can be deployed not only by a manufacturer but also by the owner of an IoT device. Therefore, HADES-IoT is also capable of protecting legacy devices that are no longer supported by the manufacturers.

1) Configuration: We identified two critical options that must be adjusted to run HADES-IoT successfully on all of the tested devices. First, the embedded-application binary interface (EABI) must be enabled for successful deployment. Second, the optimization of the compiler must be set to *performance* instead of *size*. If the optimization is set incorrectly, HADES-IoT will be installed successfully in the kernel but will not function properly. The details about specific configuration options for the tested devices are presented in the next section.

2) Model-Specific Configuration Options: Some device models require specific configuration options that must be adjusted in the configuration file for the successful compilation and deployment of HADES-IoT. We provide details about such devices below.

1) *D-Link DCS-942L:* Linux is configured in a preemptive mode on this device. Therefore, when configuring Linux options for compilation, the preemptive mode must be enabled. Note that the value of this setting can be extracted from file */proc/version*.
2) *SimpleHome XCS7-1001:* We found only one specific configuration that must be modified. Specifically, the *CONFIG_ARM_UNWIND* must be disabled; if this is not done, the installation of HADES-IoT will fail on compatibility check.
3) *Provision PT-737E:* As in the previous case, we disabled the *CONFIG_ARM_UNWIND* option, however in contrast to the previous case, we also had to enable the *CONFIG_FS_POSIX_ACL* option. To determine whether it is necessary to enable this option, the following command should output a nonempty result: *cat /proc/kallsyms|grep acl*.

B. Profiling Period

After HADES-IoT's deployment, it runs in the profiling mode, and the longer it runs in this mode, the more accurate the profile extracted is. However, long profiling periods might be inconvenient for the user. Therefore, we conducted an experiment aimed at determining the amount of profiling time needed to obtain an accurate profile. We experimented with profiling times of one, two, and four hours. The results of this experiment are presented in Table III. We can see that after 1 h of profiling, no new processes were found on any of the devices, which indicates that an accurate profile can be obtained even after just 1 h of profiling. On the other hand, there is a possibility that a new program might be executed or a new signal might be sent after the profiling period (e.g., a scheduled job). However, with the update mechanism described in Section V-E2, any missing program or signal can be added to the white list even after profiling has been completed.[15] In this experiment, we were not interested in the programs executed during the booting of the device, since these are added to the white list regardless of the length of the profiling period.

User Interaction: Next, we measured the difference in the white list size when a user interacts with the Web GUI of a device and cases without any user interaction. The results show that on some devices (e.g., *NETGEAR WNR2000v3* and *D-Link DCS-942L*), the white list size increases significantly, while for other devices (e.g., *Cisco Linksys E4200* and *ASUS RT-N56U*), the increase is insignificant. These results suggest that it is important to interact with the device during the profiling period, otherwise many programs and signals may be missed, leading to a less accurate profile. We emphasize that such interaction can be simulated by an additional profiling script provided by a manufacturer, and thus the user does not need

[15]We note that the user (i.e., the administrator of the device) is notified by the remote control application whenever some binary is prevented from being executed. Then, the user can decide to add such a binary to the white list, especially if the device would lose some functionality after the addition to the white list.

to interact with the device during the profiling performed after a firmware update.

Actively Used Executables: Finally, we compared the number of all executables to the number of executables included in the white list. Table III shows that each device contains a large number of executables that are never used. If the attacker compromises the device, none of these executables can be used due to the protection provided by HADES-IoT, hence the attacker is strictly limited to the executables in the white list.

C. Effectiveness of Detection and Prevention

To demonstrate HADES-IoT's attack prevention capabilities, we executed attacks that exploit vulnerabilities using real-world IoT malware whose goal is to compromise the device and recruit it to a DDoS botnet. All of the attacks were performed in laboratory conditions. We describe them in the following.

1) Enabled Telnet With Default Credentials and Mirai: The Mirai IoT malware takes advantage of the fact that many IoT devices connected to the Internet have Telnet open by default and many devices have configured default credentials (e.g., the SimpleHome IP camera). However, with HADES-IoT, even such a default misconfiguration does not cause harm, as execution of any unauthorized binary (e.g., Mirai binary) is terminated upon being spawned, as shown in our evaluation.

2) [CVE-2017-8225] and IoT Reaper, Persirai: CVE-2017-8225 represents a vulnerability of a custom HTTP server that does not properly check access to *.ini* files and allows the attacker to retrieve them. Authentication can be bypassed by providing an empty string for a user name and password. By exploiting this vulnerability, the attacker can read the credentials of the superuser from the *system.ini* file, as they are stored in plain text. When the attacker is in possession of the credentials, she can further misuse them to execute any command on the target device (e.g., Telnet service). This vulnerability was exploited by Persirai and IoT Reaper.

We bootstrapped HADES-IoT on an IP camera (i.e., ProVision PT-737E) and executed the exploit. In the first step of the exploit, the credentials were retrieved by reading the *system.ini* file. Since this is handled by the HTTP server included in the white list, HADES-IoT allows this action. In the second step, the remote command is sent through the FTP configuration CGI. According to HADES-IoT's logs, this executes the *chmod* utility, as well as *ftpupload.sh* which executes the command. However, since none of these executables are in the white list, HADES-IoT terminates both of them upon their execution and thus effectively stops the attack.

3) [CVE-2013-2678] and IoT Reaper, VPNFilter: The CVE-2013-2678 vulnerability enables the attacker to execute an arbitrary command on the affected device. Therefore, attackers usually exploit this vulnerability to start the Telnet service and then deliver the malicious binary on the device. We used the Metasploit framework to exploit this vulnerability on an unprotected device (Cisco Linksys E4200). We observed that the *SIGUSR1* signal was sent by *httpd* to */sbin/preinit*, and subsequently, Telnet was enabled. We then reproduced the attack with HADES-IoT, but the attack failed at the moment *SIGUSR1* was sent; the signal was not in the white list, and thus HADES-IoT stopped it.

4) [CVE-2014-9583] and VPNFilter: CVE-2014-9583 represents a vulnerability in the *infosrv* service running on ASUS routers. The service enables the attacker to execute an arbitrary command with root privileges on the vulnerable routers. Therefore, this vulnerability can be exploited by the VPNFilter IoT malware, which infects the device and recruits it to a botnet. We tested exploitation of this vulnerability on an ASUS RT-N56U device protected by HADES-IoT in two scenarios: 1) with disabled and 2) enabled Telnet. With Telnet disabled (default option), an attempt to compromise the device was detected upon the exploit's execution. However, in the case in which Telnet was enabled, the attack was detected in its later stage—upon malware download or during its execution, depending on the user's typical interaction with the device.

5) TelnetEnable Magic Packet and VPNFilter: NETGEAR routers allow a user to enable the Telnet service via a specially crafted (magic) packet. This "feature" is exploited by the creators of VPNFilter malware for its deployment. We tested the exploitation of this vulnerability on an NETGEAR WNR2000v3 device. Like the previous vulnerability, if the Telnet service is disabled, the attack by VPNFilter is detected by HADES-IoT when it begins. With enabled Telnet, the attack is thwarted as soon as any unauthorized process is spawned.

D. CPU and Memory Overhead

An important aspect of a host-based defense system for IoT devices is low overhead. We conducted an experiment in which we measured the CPU utilization of HADES-IoT and its memory demands (see Fig. 7). More specifically, we measured an average CPU load representing the value of how much work has been done on the system in the previous N minutes [56]—in our case we used an interval of 5 min. Although this metric has no clear boundaries, it is the option most available on all IoT devices; common utilities, such as *top* do not show LKMs and their resource utilization; hence, this information cannot be obtained from them.

Idling Versus User Interaction: Our measurements show that when a device is idle, there is usually a small amount of overhead. However, for some devices (e.g., *ProVision PT-737E*), the CPU load during the idle state is significantly higher than on other devices; this is caused by the periodic execution of hundreds of processes that perform various activities, e.g., extract various information from the device. Since HADES-IoT checks every executed process, the overhead imposed is higher than that of less active devices; the imposed overhead comprises 34.6% and 36.9% of the total overhead for the idle case and the case of user interaction, respectively.[16] Furthermore, we can see that on some devices (e.g., *SimpleHome* and *D-Link* devices), HADES-IoT imposes greater overhead when the user interacts with the device; this is caused by the need to spawn additional processes to handle the user's requests. This is not the case for other devices where the requests are handled by processes that are already running.

[16]We ignore devices with negligible total overhead (i.e., below 0.1).

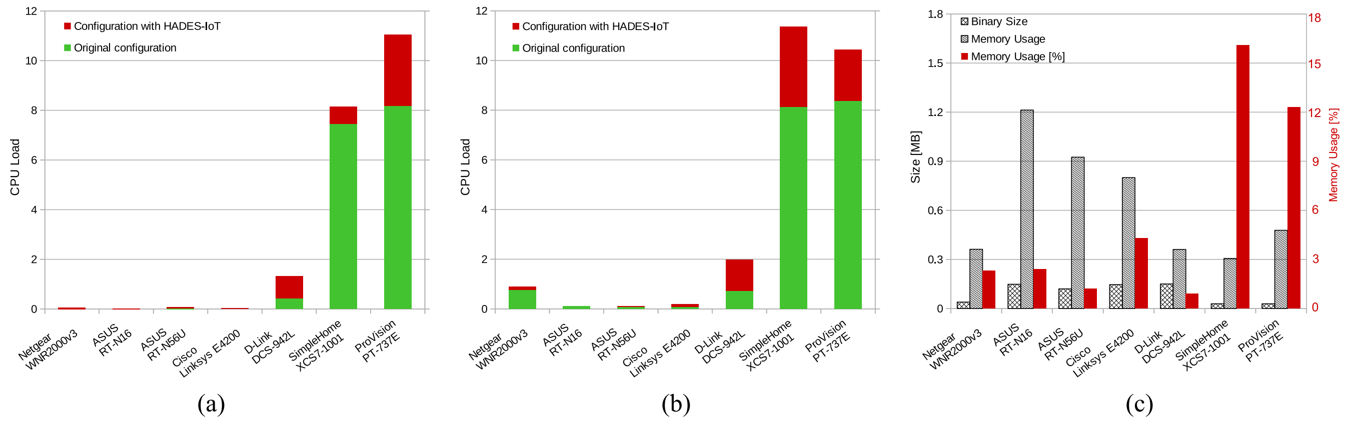


Fig. 7. Utilization of resources by HADES-IoT. (a) CPU load (idle). (b) CPU load (with user interaction). (c) Memory and storage utilization.

Memory and Storage: Next, we measured the memory and storage demands of HADES-IoT, and the results (see Fig. 7) demonstrate its low memory demands in contrast to the available space: 5.5% on average. Regardless of whether devices have the same source code, the binary size and average memory usage varies for each device. One of the reasons for this is that various cross-compilers were used to compile HADES-IoT for each device. However, the configuration of the Linux kernel against which HADES-IoT is compiled has the highest impact on the binary size. For example, debug symbols are included in the HADES-IoT's binary due to the kernel configuration, which increases the size of the binary. Also, note that differences in memory usage across various devices are caused by the tamper-proof mechanisms introduced—the HADES-IoT's binary and *init* file that must be loaded into the memory; the size of these files differs for each device, therefore loading them results in different memory consumption for each device.

Signing Scheme: All of the public and private keys of the Merkle signature scheme (see Section VI) are generated on the machine, externally of HADES-IoT, during deployment, while HADES-IoT must store only 32 B, representing the master public key (i.e., root hash of the Merkle tree) and 4 B representing the order of the last executed command to prevent reply attacks; in each case, this is only a minimal storage requirement. When the signed command is delivered to HADES-IoT, signature verification has to be performed. The computational overhead of signature verification has logarithmic time complexity in the number of leaves (i.e., the number of possible commands) in the Merkle tree. For example, we assume the number of leaves to be 2^{15}, which is more than enough for the lifetime of any IoT device, assuming that a command issued through the remote control subsystem is a rare action that might occur once, or just a few times, a year. In this case, HADES-IoT has to compute one hash computation to obtain the hash of a message with the command $h(m)$, 256 hash computations for the derivation of the Lamport public key from the one-time private key provided, and 15 hash computations to derive the root hash from the Lamport public key. Although we were unable to precisely profile the time required for signature verification, we estimate that signature verification should not take longer than 100 ms (even for the Merkle tree with the greatest number of leaves), and thus it has a negligible impact on the device performance.

VIII. Discussion

A. Firmware Updates

Occasionally an IoT device is updated to reflect improvements and bug fixes provided by a manufacturer. During the update, the executables on a device are modified, and new ones could be added. Consequently, the behavior, in terms of the processes executed, may change as well; thus, HADES-IoT reprofiling should reflect the changes.

1) Update Types: We consider two types of firmware update procedures: 1) in the *naïve update procedure*, the flash of the device is fully rewritten, which means that HADES-IoT is wiped out and needs to be deployed again and 2) in the *advanced update procedure*, only relevant files are changed, while the rest remain untouched, including the configuration of services. This makes the update more convenient, as there is no need to bootstrap HADES-IoT again, but nevertheless, reprofiling must take place. Note that before updating a device, HADES-IoT must be disabled by the remote control mechanism (see Section V-E2).

2) Frequency of Updates: Another aspect that must be considered is the frequency of firmware updates. If IoT devices were updated very frequently, then it would be inconvenient for the end user to update HADES-IoT after each firmware update. To evaluate how often firmware updates are released, we inspected the release dates of firmware updates for five selected IoT devices produced by three major IoT manufacturers (i.e., D-Link, TP-Link, and ASUS). As seen in Table IV, the average amount of time between updates is over five months. Users, however, usually update the firmware of their devices less frequently, and in some cases they do not update the firmware at all. Even if users were to update their devices each time a new firmware update was released, the effort required to update HADES-IoT is unlikely to be significant in contrast to the effort needed to perform the firmware update itself.

3) Deployment and Updates by Manufacturer: In addition to being deployed by the user, HADES-IoT could be deployed by a manufacturer off-the-shelf. The updates of an IoT device

TABLE IV
AVERAGE TIME BETWEEN UPDATES

Manufacturer	Model	Avg. Time Between Updates [Months]
ASUS	RT-N10E [57]	15.03
	RT-N10 [58]	1.1
	RT-N56U [59]	6.03
	RT-N66U [60]	5.10
	RT-AC66U [61]	4.77
D-Link	DIR-890L [62]	2.95
	DCS-930L [62]	6.67
	DAP-1860 [62]	4.96
	DCS-936L [62]	5.77
	DCS-942L [62]	10.01
TP-Link	NC220 [63]	9.00
	TL-WR1043ND V1 [64]	5.98
	Archer C2300 [65]	9.21
	Archer D7 V1 [66]	2.55
	Deco M5 V1 [67]	5.8
Average		**6.32**

can also be executed remotely by a manufacturer, who would then inform the user about it. Doing so would benefit both manufacturers and end users, as manufacturers could claim that their IoT device includes a security solution and users would need to invest less time and effort on updating their devices. HADES-IoT management can also be handled by the user, who can disable/limit manufacturer support.

B. Expanding the White List

If a legitimate binary was not included in the white list during the profiling period, the user can add the binary to the white list by using the remote control application (see Section V-E2). Note that the user is informed about all binaries prevented from being executed through the remote control application.

C. Deployment on Linux-Based Machines

In general, HADES-IoT could run on any Linux-based machine, including a PC. However, it is important to note that HADES-IoT was designed specifically for IoT devices, which have limited resources. In contrast to IoT devices, PCs provide us with features, such as *KProbe*, *Auditd*, and *SELinux*, which could be used to devise an approach similar but easier to develop than HADES-IoT. Also, while IoT devices have a rather small and fixed set of applications, the application environment on PCs changes more frequently. To examine the differences in these environments, we installed HADES-IoT on two PCs (see Table V). We observed a significantly higher number of unique processes on the PCs compared to IoT devices, however the set of unique processes is unstable. Another limiting factor for the deployment of HADES-IoT on PCs is the high frequency of updates, which would necessitate frequent reprofiling. For these reasons, we argue that HADES-IoT is not appropriate for volatile environments such as those of PCs.

D. Potential Attacks on HADES-IoT

1) Buffer Overflow and Code Injection: Buffer overflow attacks take advantage of an unhandled write to a program's buffer and write the malicious data beyond the boundary of a buffer. Code injection attacks exploit an improper input validation and enable the injection of malicious code into the program. In both cases, the attacker can modify a program's execution flow by crafting a specific payload and writing it to the memory. In general, HADES-IoT does not cover this kind of attack, since it does not check the memory content of a process. However, it is important to consider the goal of the attacker. If the attacker performs a code injection attack to manipulate the vulnerable service itself, without executing any other program, HADES-IoT would not detect it. On the other hand, there are simple protection countermeasures, such as address space layout randomization (ASLR) and nonexecutable stack, which can mitigate these kind of attacks. If the goal of the attacker is to remotely execute a shell command, e.g., with an intention of opening a reverse shell or downloading malicious binaries via the FTP client, the attacker will fail, as these programs would not be included in the white list, and thus HADES-IoT would prevent them from being executed. Although HADES-IoT is unable to detect the initial phase of such an attack, it will protect the device later, when an unknown binary is executed.

2) Disguising Malicious Binaries: The attacker might think of disguising malicious binaries so they appear as benign ones. However, since we utilize a cryptographically secure hash function which is applied on the contents of the binary (plus arguments) when searching in the white list, it is computationally infeasible to create a disguised binary with the same hash as any of the existing binaries in the white list.

3) Utilization of White Listed Programs: Assuming that the attacker might know the set of programs in the white list, she can potentially chain these programs in order to conduct an attack (often referred to as *living off the land*) which is similar, in principle, to return-oriented programming (ROP) [68]. Since HADES-IoT only checks the parameters of programs in specific cases, such an attack would not be detected. Although this attack cannot be fully prevented by HADES-IoT, it can be mitigated with more fine-grained logic of the index to the white list, similar to the way we handled this in the case of *insmod* and *rmmod* commands (see Sections V-D2 and V-D3), where we include the arguments of binaries in the hash computation.

4) Writes to /dev/kmem: Since we consider legacy IoT devices, where all processes might be executed under the root account, the attacker might directly write to the kernel memory through the */dev/kmem* file and potentially rewrite HADES-IoT's code. To provide full protection against this attack, HADES-IoT would have to intercept the *open()* system call, explicitly checking whether the name of the file being opened equals to the */dev/kmem* file.

5) Side-Loading of Dynamic Libraries: In the case in which a white listed application (e.g., a Web server) contains a vulnerability that would enable the attacker to load an attacker's dynamic library at runtime, HADES-IoT would not prevent

TABLE V
COMPARISON OF IoT DEVICES AND PCs

Device	Device Type	Linux	Kernel Version	Unique Process	Non-Kernel Processes	Kernel Processes Included
-	PC	Ubuntu 16.04 (Development)	4.15.0	115	118	183
-	PC	Kali 2.0 (Entertainment)	4.17.17	102	132	190
NETGEAR WNR2000v3	IoT	Custom	2.6.15	20	30	40
ASUS RT-N16	IoT	Custom	2.6.21	30	35	47
ASUS RT-N56U	IoT	Custom	2.6.21	20	31	48
Cisco Linksys E4200	IoT	Custom	2.6.22	20	20	35
D-Link DCS-942L	IoT	Custom	2.6.28	30	30	62
SimpleHome XCS7-1001	IoT	Custom	3.0.8	9	10	45
ProVision PT-737E	IoT	Custom	3.4.35	6	6	43

the execution of the malicious code from the library. To provide protection against such a scenario, HADES-IoT would have to intercept one of the system calls invoked when the dynamic library was loaded [i.e., *mmap()* or *open()*]. HADES-IoT would then have to maintain a dedicated white list for the dynamic libraries loaded and compute the integrity of the libraries from their contents. In the case in which there was a request to load a malicious library, HADES-IoT would stop this action before it occurred. We left practical experimentation on this problem for future work.

6) Attacker With Physical Access: In contrast to a remote attacker, an attacker with physical access has several options for compromising an IoT device protected by HADES-IoT. Although we ensure that a remote attacker is unable to tamper with the HADES-IoT binary (see Section V-D1), an attacker with physical access can do so; this capability, however, depends on the availability of the secure boot protection settings on the device. Another option for such an attacker is to obtain confidential data stored on the device by performing a hardware side-channel attack—these might be, for example, hashed passwords to the administrative interface, which can be further brute-forced and utilized to reconfigure the device. However, this attacker model is out of the scope of our work, and it is the subject of research on hardware-level protection measures.

E. Practical Extensions

In this section, we present possible extensions aimed at improving HADES-IoT's practical features; they are not part of the proof-of-concept, as they do not improve the detection performance.

1) Malware Collection: It is a common practice for malware to delete itself after its execution to hide its traces. Since HADES-IoT pauses program execution in its initial stage, it can retrieve the binary of malware before it is (potentially) deleted. Therefore, in addition to the reporting subsystem (see Section V-E1), HADES-IoT can be enhanced by the capability of collecting and reporting suspicious binaries. Such a capability would enable us to collect new malware samples.

2) Automated Extraction of Configurations: To ensure HADES-IoT's compatibility with a Linux kernel on a particular IoT device, some configuration options must be extracted from the device (see Section IV-B) and stored in the kernel configuration file. Since the configuration file consists of information about all of the options, it can be parsed and adjusted in an automated fashion instead of being adjusted by manual configuration. Therefore, the kernel configuration process performed on the user's computer could be automated by running a script that would connect to an IoT device, issue the necessary commands for extracting the important data, and based on that, modify the stock configuration file of a vanilla Linux. The configuration file obtained would be used for the compilation of the required kernel's parts, enabling proper compilation of HADES-IoT.

IX. CONCLUSION

In this article, we proposed HADES-IoT, a host-based anomaly detection system for Linux-based IoT devices, which provides proactive detection and tampering protection. HADES-IoT is based on whitelisting and utilizes a system call interception which is performed within the LKM. Thanks to the LKM, HADES-IoT gains complete control of the execution of all user space and kernel space programs, and any execution of an unauthorized binary can be thwarted when it starts. HADES-IoT is lightweight in terms of its size, memory, and CPU demands. Computational overhead is only influenced by the number of processes spawned on the device but not by accessing the white list—searching in the white list has a constant time complexity. In the evaluation, we showed that extraction of an accurate device profile can be performed in an hour. Then, we demonstrated that HADES-IoT has 100% effectiveness in the detection of five kinds of IoT malware executed on seven Linux-based IoT devices. We also proposed a mechanism for remote control of HADES-IoT, which enables, e.g., updating the white list and reporting malware attempts to compromise an IoT device.

REFERENCES

[1] D. N. Serpanos and A. G. Voyiatzis, "Security challenges in embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 1s, pp. 1–10, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2435227.2435262
[2] G. V. Hulme. "Embedded System Security Much More Dangerous, Costly than Traditional Software Vulnerabilities." 2012. [Online]. Available: http://bit.ly/30ij4yZ
[3] R. Newell. "The Biggest Security Threats Facing Embedded Designers." 2016. [Online]. Available: http://www.electronicdesign.com/iot/biggest-security-threats-facing-embedded-designers
[4] C. Cimpanu. "You Can Now Rent a Mirai Botnet of 400,000 Bots." 2016. [Online]. Available: https://www.bleepingcomputer.com/news/security/you-can-now-rent-a-mirai-botnet-of-400-000-bots/

[5] D. Gooding. "New IoT Botnet Offers DDoses of Once-Unimaginable Sizes for $20." 2018. [Online]. Available: https://arstechnica.com/information-technology/2018/02/for-sale-ddoses-guaranteed-to-take-down-gaming-servers-just-20/

[6] "The Cost of Renting an IoT Botnet." Nuvias. 2017. [Online]. Available: https://www.nuviasblog.com/main-category/security/cost-renting-iot-botnet/

[7] D. Canellis. "Crypto-Jacking Epidemic Spreads to 30K Routers Across India." 2018. [Online]. Available: https://thenextweb.com/hardfork/2018/10/05/crypto-jacking-malware-coinhive/

[8] F. Merces. "Miner Malware Targets IoT, Offered in the Underground." 2018. [Online]. Available: http://bit.ly/3rnoapD

[9] J. Müller, V. Mladenov, J. Somorovsky, and J. Schwenk, "SoK: Exploiting network printers," in *Proc. IEEE Symp. Security Privacy (SP)*, San Jose, CA, USA, May 2017, pp. 213–230. [Online]. Available: https://doi.org/10.1109/SP.2017.47

[10] J. Graham. "Your Smart TV May Be Prey for Hackers and Collecting More Info Than You Realize, 'Consumer Reports' Warns." 2018. [Online]. Available: http://bit.ly/2PCBE2V

[11] Y. Seralathan *et al.*, "IoT security vulnerability: A case study of a Web camera," in *Proc. 20th Int. Conf. Adv. Commun. Technol. (ICACT)*, Chuncheon, South Korea, 2018, pp. 172–177. [Online]. Available: https://doi.org/10.23919/ICACT.2018.8323686

[12] L. Wyatt and C. Bozzato. "Your Cheap IP Camera May Be Leaking Video." 2019. [Online]. Available: https://blog.talosintelligence.com/2019/08/vuln-spotlight-nest-camera-openweave-aug-2019.html

[13] G. Fleishman. "Your Cheap IP Camera May Be Leaking Video." 2015. [Online]. Available: https://www.macworld.com/article/3012018/your-cheap-ip-camera-may-be-leaking-video.html

[14] M. Antonakakis *et al.*, "Understanding the mirai botnet," in *Proc. 26th USENIX Security Symp. (USENIX Security)*, Vancouver, BC, Canada, Aug. 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[15] "Owasp IoT Top 10." OWASP Internet Things Team. 2018. [Online]. Available: https://www.owasp.org/images/1/1c/OWASP-IoT-Top-10-2018-final.pdf

[16] "VPNFilter-Affected Devices Still Riddled with 19 Bugs." TrendMicro. 2018. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/vpnfilter-affected-devices-still-riddled-with-19-vulnerabilities/

[17] "Cisco: SNORT." 2019. [Online]. Available: https://www.snort.org/

[18] "Suricata." 2019. [Online]. Available: http://suricata-ids.org/

[19] T. Shon and J. Moon, "A hybrid machine learning approach to network anomaly detection," *Inf. Sci.*, vol. 177, no. 18, pp. 3799–3821, 2007. [Online]. Available: https://doi.org/10.1016/j.ins.2007.03.025

[20] I. Homoliak, "Intrusion detection in network traffic," Ph.D. dissertation, Dept. Inf. Technol., Univ. Technol., Brno, Czechia, 2016. [Online]. Available: http://dx.doi.org/10.13140/RG.2.2.25780.24963/1

[21] P. Fogla, M. I. Sharif, R. Perdisci, O. M. Kolesnikov, and W. Lee, "Polymorphic blending attacks," in *Proc. 15th USENIX Security Symp.*, Vancouver, BC, Canada, Jul./Aug. 2006, pp. 241–256. [Online]. Available: https://www.usenix.org/conference/15th-usenix-security-symposium/polymorphic-blending-attacks

[22] G. Vigna, W. Robertson, and D. Balzarotti, "Testing network-based intrusion detection signatures using mutant exploits," in *Proc. 11th ACM Conf. Comput. Commun. Security (CCS)*, 2004, pp. 21–30. [Online]. Available: https://doi.org/10.1145/1030083.1030088

[23] M. Boltz, M. Jalava, and J. Walsh, "New methods and combinatorics for bypassing intrusion prevention technologies," Stonesoft, Helsinki, Finland, Rep., 2010. [Online]. Available: https://silo.tips/download/new-methods-and-combinatorics-for-bypassing-intrusion-prevention-technologies

[24] I. Homoliak, M. Teknøs, M. Ochoa, D. Breitenbacher, S. Hosseini, and P. Hanacek, "Improving network intrusion detection classifiers by non-payload-based exploit-independent obfuscations: An adversarial approach," *EAI Endorsed Trans. Security Safety*, vol. 5, no. 17, p. e4, Dec. 2018. [Online]. Available: https://doi.org/10.4108/eai.10-1-2019.156245

[25] H. Debar, M. Dacier, and A. Wespi, "Towards a taxonomy of intrusion-detection systems," *Comput. Netw.*, vol. 31, no. 8, pp. 805–822, 1999. [Online]. Available: https://doi.org/10.1016/S1389-1286(98)00017-6

[26] D. Breitenbacher, I. Homoliak, Y. L. Aung, N. O. Tippenhauer, and Y. Elovici, "HADES-IoT: A practical host-based anomaly detection system for IoT devices," in *Proc. ACM Asia Conf. Comput. Commun. Security (AsiaCCS)*, Auckland, New Zealand, Jul. 2019, pp. 479–484. [Online]. Available: https://doi.org/10.1145/3321705.3329847

[27] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, "Learning execution contexts from system call distribution for anomaly detection in smart embedded system," in *Proc. 2nd Int. Conf. Internet Things Design Implement. (IoTDI)*, 2017, pp. 191–196. [Online]. Available: https://doi.org/10.1145/3054977.3054999

[28] F. M. Tabrizi and K. Pattabiraman, "A model-based intrusion detection system for smart meters," in *Proc. 15tj Int. Symp. High Assur. Syst. Eng. (HASE)*, Miami Beach, FL, USA, 2014, pp. 17–24. [Online]. Available: https://doi.org/10.1109/HASE.2014.12

[29] A. Agarwal, S. Dawson, D. McKee, P. Eugster, M. Tancreti, and V. Sundaram, "Detecting abnormalities in IoT program executions through control-flow-based features: Poster abstract," in *Proc. 2nd Int. Conf. Internet Things Design Implement.*, 2017, pp. 339–340. [Online]. Available: https://doi.org/10.1145/3054977.3057312

[30] N. An, A. Duff, G. Naik, M. Faloutsos, S. Weber, and S. Mancoridis, "Behavioral anomaly detection of malware on home routers," in *Proc. 12th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Fajardo, PR, USA, 2017, pp. 47–54. [Online]. Available: https://doi.org/10.1109/MALWARE.2017.8323956

[31] J. Su, D. V. Vasconcellos, S. Prasad, S. Daniele, Y. Feng, and K. Sakurai, "Lightweight classification of IoT malware based on image recognition," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Tokyo, Japan, 2018, pp. 664–669. [Online]. Available: https://doi.org/10.1109/COMPSAC.2018.10315

[32] A. Mudgerikar, P. Sharma, and E. Bertino, "E-Spion: A system-level intrusion detection system for IoT devices," in *Proc. ACM Asia Conf. Comput. Commun. Security (AsiaCCS)*, Auckland, New Zealand, Jul. 2019, pp. 493–500. [Online]. Available: https://doi.org/10.1145/3321705.3329857

[33] X. Wang and X. Lu, "A host-based anomaly detection framework using XGBoost and LSTM for IoT devices," *Wireless Commun. Mobile Comput.*, vol. 2020, pp. 1–13, Oct. 2020. [Online]. Available: https://doi.org/10.1155/2020/8838571

[34] R. Gassais, N. Ezzati-Jivan, J. Fernandez, D. Aloise, and M. Dagenais, "Multi-level host-based intrusion detection system for Internet of Things," *J. Cloud Comput.*, vol. 9, p. 62, Nov. 2020. [Online]. Available: https://doi.org/10.1186/s13677-020-00206-6

[35] A. R. Javed, S. U. Rehman, M. U. Khan, M. Alazab, and T. Reddy, "CANintelliIDS: Detecting in-vehicle intrusion attacks on a controller area network using CNN and attention-based GRU," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1456–1466, Apr.–Jun. 2021. [Online]. Available: https://doi.org/10.1109/TNSE.2021.3059881

[36] J. Jeon, B. Jeong, S. Baek, and Y.-S. Jeong, "Hybrid malware detection based on bi-LSTM and SPP-net for smart IoT," *IEEE Trans. Ind. Informat.*, early access, Oct. 14, 2021. [Online]. Available: https://doi.org/10.1109/TII.2021.3119778

[37] A. Cosson, A. K. Sikder, L. Babun, Z. B. Celik, P. McDaniel, and A. S. Uluagac, "Sentinel: A robust intrusion detection system for IoT networks using kernel-level system information," in *Proc. Int. Conf. Internet Things Design Implement.*, 2021, pp. 53–66. [Online]. Available: https://doi.org/10.1145/3450268.3453533

[38] H. Alasmary *et al.*, "SHELLCORE: Automating malicious IoT software detection using shell commands representation," *IEEE Internet Things J.*, early access, Jun. 3, 2021. [Online]. Available: https://doi.org/10.1109/JIOT.2021.3086398

[39] Q.-D. Ngo, H.-T. Nguyen, H.-A. Tran, N.-A. Pham, and X.-H. Dang, "Toward an approach using graph-theoretic for IoT botnet detection," in *Proc. 2nd Int. Conf. Comput. Netw. Internet Things*, 2021, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3468691.3468714

[40] C. Tamás, D. Papp, and L. Buttyán, "SIMBIoTA: Similarity-based malware detection on IoT devices," in *Proc. 6th Int. Conf. Internet Things Big Data Security (IoTBDS)*, Apr. 2021, pp. 58–69. [Online]. Available: https://doi.org/10.5220/0010441500580069

[41] Q. Li, J. Mi, W. Li, J. Wang, and M. Cheng, "CNN-based malware variants detection method for Internet of Things," *IEEE Internet Things J.*, vol. 8, no. 23, pp. 16946–16962, Dec. 2021. [Online]. Available: https://doi.org/10.1109/JIOT.2021.3075694

[42] F. Ding *et al.*, "DeepPower: Non-intrusive and deep learning-based detection of IoT malware using power side channels," in *Proc. 15th ACM Asia Conf. Comput. Commun. Security (ASIA CCS)*, Taipei, Taiwan, Oct. 2020, pp. 33–46. [Online]. Available: https://doi.org/10.1145/3320269.3384727

[43] R. A. Bridges, T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, and Q. G. Chen, "A survey of intrusion detection systems leveraging host data," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 1–35, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3344382

[44] M. Liu, Z. Xue, X. Xu, C. Zhong, and J. Chen, "Host-based intrusion detection system with system calls: Review and future trends," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–36, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3214304

[45] I. Homoliak, K. Malinka, and P. Hanacek, "ASNM datasets: A collection of network attacks for testing of adversarial classifiers and intrusion detectors," *IEEE Access*, vol. 8, pp. 112427–112453, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3001768
[46] I. Corona, G. Giacinto, and F. Roli, "Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues," *Inf. Sci.*, vol. 239, pp. 201–225, Aug. 2013. [Online]. Available: https://doi.org/10.1016/j.ins.2013.03.022
[47] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: A novel honeypot for revealing current IoT threats," *J. Inf. Process.*, vol. 24, no. 3, pp. 522–533, 2016. [Online]. Available: https://doi.org/10.2197/ipsjjip.24.522
[48] "2020 IoT Developer Survey Key Findings." Eclipse IoT. 2020. [Online]. Available: https://outreach.eclipse.foundation/eclipse-iot-developer-survey-2020
[49] "IoT Developer Survey 2019 Results." Eclipse IoT. 2019. [Online]. Available: https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2019.pdf
[50] A. Anubhav. "Understanding the IoT Hacker—A Conversation with Owari/Sora IoT Botnet Author." 2018. [Online]. Available: http://bit.ly/3qsZICd
[51] "System Call Definition." Bellevue Linux. 2006. [Online]. Available: http://www.linfo.org/system_call.html
[52] K. Michael. *Linux Programmer's Manual*. (2017). [Online]. Available: http://man7.org/linux/man-pages/man2/syscalls.2.html
[53] "Executing a File." GNU. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html (Accessed: Feb. 17, 2021).
[54] R. C. Merkle, "A certified digital signature," in *Proc. Conf. Theory Appl. Cryptol.*, 1989, pp. 218–238. [Online]. Available: https://doi.org/10.1007/0-387-34805-0_21
[55] L. Lamport, "Constructing digital signatures from a one-way function," SRI Int., Palo Alto, CA, USA, Rep. CSL-98, 1979.
[56] N. J. Gunther. "Unix Load Average, Part 1." 2010. [Online]. Available: https://www.teamquest.com/import/pdfs/whitepaper/ldavg1.pdf
[57] "BIOS & FIRMWARE for Asus RT-N10E." ASUSTeK Computer Inc. 2017. [Online]. Available: https://www.asus.com/supportonly/rt-n10e/HelpDesk_BIOS/
[58] "BIOS & FIRMWARE for ASUS RT-N10." ASUSTeK Computer Inc. 2017. [Online]. Available: https://www.asus.com/supportonly/rt-n10/HelpDesk_BIOS/
[59] "BIOS & FIRMWARE for ASUS RT-N56U." ASUSTeK Computer Inc. 2017. [Online]. Available: https://www.asus.com/us/SupportOnly/RT-N56U/HelpDesk_BIOS/
[60] "BIOS & FIRMWARE for ASUS RT-N66U." ASUSTeK Computer Inc. 2020. [Online]. Available: https://www.asus.com/supportonly/RT-N66U
[61] "BIOS & FIRMWARE for ASUS RT-AC66U." ASUSTeK Computer Inc. 2021. [Online]. Available: https://www.asus.com/supportonly/RT-AC66U/HelpDesk_BIOS/
[62] "Firmware Downloads." D.-Link Corporation, 2021. [Online]. Available: https://tsd.dlink.com.tw/
[63] "Firmware for NC200 V1." TP-Link. 2020. [Online]. Available: https://www.tp-link.com/support/download/nc200/#Firmware
[64] "Firmware for TL-WR1043ND V1." TP-Link. 2014. [Online]. Available: https://www.tp-link.com/support/download/tl-wr1043nd/v1/#Firmware
[65] "Firmware for Archer C2300 V1," TP-Link. 2019. [Online]. Available: https://www.tp-link.com/support/download/archer-c2300/v1/#Firmware
[66] "Firmware for Archer D7 V1." TP-Link. 2015. [Online]. Available: https://www.tp-link.com/support/download/archer-d7/#Firmware
[67] "Firmware for Deco M5 V1," TP-Link. 2021. [Online]. Available: https://www.tp-link.com/support/download/deco-m5/v1/#Firmware
[68] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Security*, vol. 15, no. 1, p. 2, 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

Dominik Breitenbacher received the M.Sc. degree from the Faculty of Information Technology, Brno University of Technology, Brno, Czechia, in 2014, where he is currently pursuing the Ph.D. degree.

He is a Malware Researcher with ESET, Brno, where he focuses mainly on APT detection. Prior to joining ESET, he worked as a Research Assistant with iTrust, Centre of Cyber Security, Singapore University of Technology and Design, Singapore, where he focused on intrusion detection in IoT devices.

Ivan Homoliak (Member, IEEE) received the Ph.D. degree in adversarial intrusion detection in network traffic from the Faculty of Information Technology (FIT), Brno University of Technology (BUT), Brno, Czechia, in 2016.

He is a Research Scientist with FIT, BUT. Prior to joining FIT, BUT, he was worked with Singapore University of Technology and Design, Singapore, on various projects focusing on the security of blockchains and insider threat detection. His research interests include cryptocurrency wallets, distributed ledgers, e-voting, and consensus protocols.

Yan Lin Aung received the Bachelor of Engineering and Ph.D. degrees in computer engineering from Nanyang Technological University, Singapore, in 2007 and 2016, respectively.

He is a Postdoctoral Research Fellow with iTrust, Centre of Cyber Security, Singapore University of Technology and Design, Singapore. His research interests include Internet of Things and cyber–physical systems security, cloud computing and security, machine learning, reconfigurable, and custom computing.

Yuval Elovici received the Ph.D. degree in information systems from Tel-Aviv University, Tel Aviv, Israel, in 1998.

He is the Director of the Telekom Innovation Laboratories, BenGurion University of the Negev, Beersheba, Israel, where he is the Head of the BGU Cyber Security Research Center and a Professor with the Department of Software and Information Systems Engineering. His primary research interests are computer and network security, cyber security, Web intelligence, information warfare, social network analysis, and machine learning.

Nils Ole Tippenhauer (Member, IEEE) received the Ph.D. degree from ETH Zürich, Zürich, Switzerland.

He is a Faculty of CISPA Helmholtz Center for Information Security, Saarbrücken, Germany. Prior to joining CISPA, he held a position as an Assistant Professor with Singapore University of Technology and Design, Singapore. He is interested in information security aspects of practical systems, in particular cyber–physical systems, such as industrial control systems.