Remote Non-Intrusive Malware Detection for PLCs based on Chain of Trust Rooted in Hardware

Prashant Hari Narayan Rajput[†], Esha Sarkar[†], Dimitrios Tychalas[†], Michail Maniatakos[*]
[†]*NYU Tandon School of Engineering,* [*]*New York University Abu Dhabi*
{prashanthrajput, esha.sarkar, dimitris.tychalas, mihalis.maniatakos}@nyu.edu

2021 IEEE European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1491-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/EUROSP51992.2021.00033

***Abstract*—Digitization has been rapidly integrated with manufacturing industries and critical infrastructure to increase efficiency, productivity, and reduce wastefulness, a transition being labeled as Industry 4.0. However, this expansion, coupled with the poor cybersecurity posture of these Industrial Internet of Things (IIoT) devices, has made them prolific targets for exploitation. Moreover, modern Programmable Logic Controllers (PLC) used in the Operational Technology (OT) sector are adopting open-source operating systems such as Linux instead of proprietary software, making such devices susceptible to Linux-based malware. Traditional malware detection approaches cannot be applied directly or extended to such environments due to the unique restrictions of these PLC devices, such as limited computational power and real-time requirements. In this paper, we propose ORRIS, a novel lightweight and out-of-the-device framework that detects malware at both kernel and user-level by processing the information collected using the Joint Test Action Group (JTAG) interface. We evaluate ORRIS against in-the-wild Linux malware achieving maximum detection accuracy of ≈99.7% with very few false-positive occurrences, a result comparable to the state-of-the-art commercial products. Moreover, we also develop and demonstrate a real-time implementation of ORRIS for commercial PLCs.**

***Index Terms*—Rootkit, Malware Detection, Hardware Performance Counters, Hardware Root-of-Trust, JTAG**

I. Introduction

In the last few years, industries from various facets of manufacturing, such as the automotive sector, machinery, and aeronautics, as well as critical infrastructures, like desalination plants and smart grid facilities, have been steadily adopting Industrial Internet of Things (IIoT) devices to facilitate their Operational Technology (OT) sector. This rapid digitization results from the robust integration of Cyber-Physical Systems (CPS) and IIoT, dubbed as *Industry 4.0*. It is a current trend of automation and data exchange in the OT sector to increase productivity and reduce cost by using predictive maintenance scheduling, establishing the foundations of a smart factory [1]. This trend of embracing digitization has grown from 33% in 2016 to about 72% of manufacturers in 2020 [2].

Furthermore, the Industrial Control Systems (ICS) market is estimated to grow from $13.2 billion in 2019 to $18.053 billion by 2024 [3]. The growth of ICS combined with the hasty adoption of IIoT devices has brought along with itself all of its associated vulnerabilities to the OT sector, making it a lucrative target for adversaries [4]. A 2019 survey conducted by Ponemon Institution on 701 OT organizations concluded that 90% of the respondents experienced at least one damaging cyberattack, and 62% of them experienced two or more [5]. ICS has also been a target of cyberattacks in the past. For instance, consider the 2008 attack on oil pipelines. The adversaries compromised Programmable Logic Controllers (PLCs) at valve stations to increase its pressure, culminating in an explosion. This attack resulted in a spill of over 30,000 barrels of oil, costing British Petroleum $5 million a day in transit tariffs [6]. Alternatively, the infamous Stuxnet attack of 2010, which reprogrammed PLCs to modify the operation of the centrifuges for tearing themselves apart [7]. Due to the increase in frequency and scale of such cyberattacks, worldwide expenditure on cybersecurity is forecasted to reach $133.7 billion by 2022 [8].

These cybersecurity incidents follow a typical pattern commencing with a reconnaissance phase to identify an entry point, resulting in malware delivery and followed by a payload launch that exploits uncovered vulnerabilities for remote access [9]. After this, the adversary can perform privilege escalation, create backdoors to maintain persistent access, modify the system for hiding their presence and exfiltrate information from the compromised system. Modern malware, such as Stuxnet, comprises various components, including worms, trojans, ransomware, and rootkits. A rootkit is the primary enabler of malware installed by an adversary to maintain continued and privileged access to the system while concealing its presence. Rootkits can be broadly divided, based on the privilege level they compromise, into user-level and kernel-level. Rootkits are a select type of malware designed to be stealthy and must be handled separately.

While malware detection techniques have been studied extensively in literature [10], [11], [12], this research remains dormant in the context of PLCs, which brings with itself a unique set of restrictions limiting the use of traditional solutions. In general, PLCs have less computing power when compared to general-purpose computers. Consider an industrial automation PFC200 controller from WAGO, which encompasses a mere 256 MB of main memory and a 600 MHz Cortex A8 CPU [13]. Furthermore, these PLCs adhere to strict real-time requirements while controlling critical physical processes, unable to support the overhead of conventional malware detection methods. The operating system (OS) on these PLC devices rarely gets updated and often contains vulnerabilities that are patched in later versions; WAGO still uses Linux kernel 3.18. Moreover, the deployment of Linux in automation controllers is also increasing due to its openness and versatility, evident by its adoption in devices such as

DOI 10.1109/EuroSP51992.2021.00033

Opto 22 EPIC edge programmable industrial controller, PACSystems Rx3i CPL400, Wind River devices, etc. [14]. These constraints on Linux-based PLCs require a specialized solution towards malware detection, yet to be discussed extensively in the literature.

Driven by the unique restrictions put forth by PLCs and a lack of specialized malware detection solutions, we present ORRIS, a framework for protecting against rootkits and malware. Here, static analysis utilizes data from the binary of the rootkit and integrity verification of critical data structures. On the other hand, the behavior-based approach uses semantic and microarchitectural information. ORRIS uses the Joint Test Action Group (JTAG) to collect relevant data from Linux-based PLCs. The use of JTAG facilitates the implementation of ORRIS outside the protected PLC, making the framework more resilient towards tampering attempts by an adversary while not influencing its real-time requirements. Moreover, the use of JTAG establishes a chain of trust rooted in hardware and ensures elevated trust in our framework. For ORRIS to be applicable, JTAG should be accessible (i.e., not locked by the manufacturer). In summary, the main contributions of this paper are:

1) A novel methodology for detecting rootkits and malware on Linux-based PLCs using the JTAG interface as root-of-trust. The JTAG interface enables external and non-intrusive monitoring, allowing ORRIS not to affect the real-time requirements of the target device. Also, hardware breakpoints are added to prevent disabling JTAG from kernel-space malware.
2) Implementation of ORRIS as a real-time malware detection service for a Linux-based PLC controlling part of a desalination plant simulation model.
3) Analysis of the impact in the accuracy of ORRIS due to unseen malware samples and spatial bias.
4) A new dataset of 1,150 malware samples either ported from x86 or collected from multiple sources. We had to develop our own dataset since there are currently no public repositories for ARM rootkits and malware. The dataset does not include actual PLC malware; instead, malware ported from PC. Both ORRIS and the dataset will be open-sourced.

II. Preliminaries

A. Rootkits

Malware is a blanket term for malicious software such as worms, viruses, or any harmful software. Early rootkits were simplistic and worked solely at the user-level, manipulating the log files [15]. More advanced rootkits, such as *t0rnkit*, replaced legitimate system binaries like `ls`, `netstat`, and `ps` with malicious variants with supplementary logic [16], and were easily detected by tools such as Chkrootkit [17].

Kernel-level rootkits. These operate in the kernel space (Ring 0) and modify the kernel text section to alter regular execution flow. These rootkits are injected into the kernel as Loadable Kernel Modules (LKM), which facilitates on-the-fly soft kernel modifications. It is important to implement kernel-level rootkits as LKM to gain writing privileges into the kernel text section. Such a modification in the syscall table is named *system call hooking*. An example is *Diamorphine*, a kernel-level rootkit which hooks to `sys_getdents`, `sys_getdents64` and `sys_kill` system calls. It hides directories that contain a MAGIC_PREFIX (diamorphine_secret) and does not kill the processes with the task flag of PF_INVISIBLE (`0x10000000`).

User-level rootkits. These operate in the user space (Ring 3) by leveraging the preloading technique for injecting themselves between the kernel and shared libraries. File `/etc/ld.so.preload` contains the absolute path of all the shared libraries to be preloaded by the dynamic linker before any other libraries mentioned in the relocation table of a binary. Moreover, Linux allows multiple definitions for symbols in shared libraries, enabling the user-level rootkit to preload its malicious implementations before the legitimate ones, completely modifying normal execution flow in the userspace. An example is *Umbreon*, a cross-platform user-level rootkit first detected by Trend Micro in September 2016 [18] that preloads its definition of approximately 101 symbols, including `chown`. This modification returns ENOENT (Error NO ENTry) for any operation on the malicious files by a legitimate user, effectively hiding it.

B. Join Test Action Group Interface

JTAG is an IEEE 1149.1 standard for debugging and testing interconnects in printed circuit boards. It facilitates direct communication and low-level access to the embedded System on Chip (SoC). Software such as OpenOCD and Lauterbach Trace32 enable the use of JTAG for performing hardware-assisted software debugging [19]. ORRIS utilizes JTAG to gather semantic information and syscall table information by leveraging non-intrusive memory read functionality and microarchitectural event count by using real-time tracing. Here, *semantic* information refers to system-level counters that capture system operation attributes and performance figures. It also uses JTAG for setting hardware watchpoints over the protected memory region/data structure.

C. Process Control Blocks (PCB)

PCB of the type `struct task_struct` defines every process in Linux, stored in a circular doubly linked list data structure known as *task list*. This data structure contains all the information needed by the kernel to manage currently executing processes. The head of this list can be queried from `System.map` file or `/proc/kallsyms`. Our framework collects necessary semantic information directly from the task list by utilizing JTAG.

D. Embedded Trace Macrocell (ETM)

ETM is an ARM debug solution with real-time trace capabilities that provide information about the operation of the

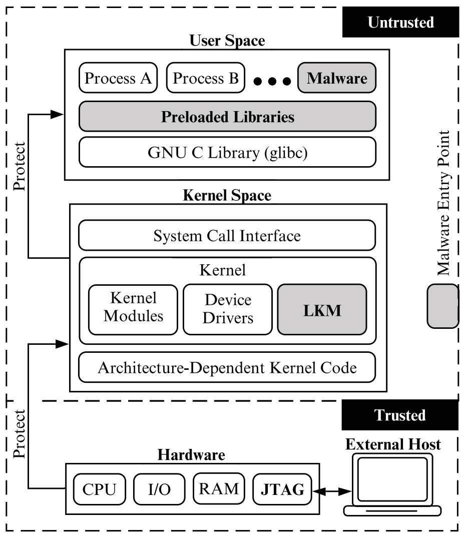


Fig. 1: Chain of trust established by ORRIS.

processor, configurable by the JTAG interface. ORRIS can write the collected trace information to an external host non-intrusively. This approach also allows ORRIS to allocate a buffer on an external host instead of the embedded device, reducing its impact on the protected PLC [20]. ETM counters can count microarchitectural events but are limited in their size and availability. Cortex A8 offers only two 16 bit ETM counters, requiring additional processing logic on the host, adding to the malware detection latency.

E. Spatial Bias

It refers to the unrealistic assumption about the ratio of goodware (benign applications) to malware in the dataset. This ratio is concrete to a particular domain; for instance, in Android, malware represents approximately 10% of the total benign applications [21]. In literature, malware detection experiments are often performed without considering spatial experimental bias, which produces results that are not representative of a real-world scenario.

III. Threat Model

This work targets malware detection on Linux-based ARM PLCs that are steadily gaining popularity [14]. We assume that an adversary has compromised the information technology or the operational technology network and has gained remote access to the PLC device, similar to Stuxnet [7] (malware reached the Windows HMI connected to the Siemens PLC) or to the 2015 attack on the Ukraine power grid attack, where the adversary utilized spear-phishing to gain access to the business network [22]. The adversary can exploit vulnerabilities in the PLC OS and software to obtain kernel-level or user-level privilege for command execution. For example, CVE-2020-6081, known since 01/07/2020, results in remote code execution using specially crafted network requests due to a vulnerability in `PLC_Task` of CODESYS Runtime 3.5.14.30.

TABLE I: ETM versions in various ARM processors.

ETM Version	Processor
ETMv1	ARM7, ARM9
ETMv3	ARM9, ARM11, Cortex-M3/M4, Cortex-M23, Cortex-R4, Cortex-R5, Cortex-A5, Cortex-A7, Cortex-A8
ETMv4	Cortex-M7, Cortex-M33, Cortex-R7/R8, Cortex-R52, Cortex-A3x, Cortex-A5x, Cortex-A7x
PTM	Cortex-A9, Cortex-A15, Cortex-A17

IV. ORRIS Methodology

In this work, we present ORRIS, an out-of-the-device framework for protecting Linux-based ARM PLCs against malware by forming a chain of trust rooted in hardware. As illustrated in Fig. 1, ORRIS creates a chain of trust beginning at the hardware level with JTAG, which protects the kernel space against rootkits. This protected kernel space becomes the next link in the chain to protect the userspace against user-level rootkits and malware. Each layer depends on the layer below for protection and, in turn, protects the layer above it, establishing a chain of trust.

A. Prerequisites

ORRIS utilizes the JTAG interface for setting hardware watchpoint, extracting critical static data structures such as the syscall table from the main memory for integrity verification. Moreover, it also gathers semantic and microarchitectural information for malware detection.

A broad set of processors, such as Cortex-A5/A8/A7/A9/A15, among others, allow external debuggers access to the Debug Access Port (DAP), enabling real-time JTAG access without halting the CPU. This initial connection enables ORRIS to start monitoring all the critical registers required for the uninterrupted functionality of JTAG, ensuring reliable connection henceforth. Often, JTAG is disabled in software at boot and might require changes in the kernel source code. Furthermore, for real-time access to microarchitectural event counts, ORRIS relies on the support of ETM in ARM. Table I shows a summary of the ETM versions supported by various ARM processors [23]. Finally, we also assume that ORRIS knows the address `init_task` which points to the `task_struct` of the Swapper/Idle process (PID = 0). This address does not change and can be extracted by querying `/proc/kallsyms`. It helps ORRIS to identify the head of the task list and aids in collecting semantic information from PCBs of all the currently running processes.

Protecting JTAG against adversaries. A question naturally arises, whether kernel-privileged malware can disable the JTAG interface, effectively disabling ORRIS. The ARM debugging model, since ARMv7, allows memory-mapped access to debug registers to enable DAP access for the on-chip processor. This allows an on-chip processor to act as a debug host for another processor [24]. An adversary with privileged access can lock debug register access to other debug monitors

TABLE II: Monthly distribution of malware dataset.

Year	**2016**						**2017**						**2018**											**2019**							
Month	Apr	May	Jun	Jul	Oct	Nov	Mar	May	Jun	Aug	Sep	Oct	Jan	Feb	Mar	Apr	May	Jun	Jul	Sep	Oct	Nov	Dec	Mar	Apr	May	Jun	Jul	Aug	Oct	Nov
Count	8	33	84	36	2	87	1	1	1	9	118	5	3	15	140	5	8	72	191	5	24	34	18	8	40	124	9	3	2	1	47

TABLE III: Summarized malware dataset.

Threat	**Family**	**Variants**								
		a^*	b^*	c^*	d^*	e^*	f^*	n^*	y^*	$Other$
Backdoor	Mirai	43	340	15	-	-	7	108	-	6
	Gafgyt	239	80	-	32	11	-	-	-	26
	Dofloo	-	-	15	45	-	4	-	-	1
	Tsunami	-	12	-	-	-	-	-	-	1
	Hajime	-	5	-	-	-	-	-	-	-
	LuaBot	-	3	-	-	-	-	-	-	-
	HideNSeek	-	-	-	-	-	-	-	-	1
Trojan	Ddostf	21	-	-	-	-	-	-	-	-
	DnsAmp	-	-	8	-	-	-	-	-	-
	Agent	-	-	-	-	-	-	-	-	1
Miscellaneous		11								
Total		1,135								

‘ * ’: Encompasses all the variants beginning with the preceding character. For instance Mirai.au, and Mirai.ad are counted under the variant a^*.

TABLE IV: Ported rootkits and their functionality.

Rootkit	**Type**		**Functionality**				
	UL	**KL**	**BD**	**HID**	**REC**	**REE**	**CLE**
Azazel	●		●	●	●	●	
Bdvl	●		●	●	●	●	●
Beurk	●		●	●		●	●
Umbreon	●		●	●	●	●	●
AFkit		●		●	●		●
Basic-rootkit		●		●	●		●
Deadlands		●			●		●
Diamorphine		●	●	●	●		●
Randkit		●					●
Rk.erb		●			●		●
Sutekh		●	●				●
Suterusu		●	●	●	●		●
Toorkit		●			●		●
Toykit		●	●	●	●		●
Uber-rootkit		●	●	●	●		●

UL: User-level KL: Kernel-level BD: Backdoor HID: Hiding REC: Reconnaissance REE: Reentry CLE: Clean on exit

by writing `0xC5ACCE55` to Lock Access Register, preventing write to the debug registers, except for the ones generated by the external debugger (does not affect ORRIS). On the other hand, writing `0xC5ACCE55` to Operating System Lock Access Register (OSLAR) can terminate debug access even for the external debuggers [25]. The modern Linux kernel does not allow an LKM to overwrite these crucial memory-mapped registers even with kernel privileges, a fact we verified for our Linux kernel. This kernel-level protection prevents an adversary with kernel privilege from disabling external debugger access by using an LKM. Still, the attackers may find another way to exploit the OS. To address this, ORRIS allocates a hardware watchpoint to monitor memory-mapped addresses of Watchpoint Control (`0x180`), Watchpoint Value (`0x1C0`) and OSLAR (`0x300`) registers, which are mapped consecutively. Memory locations in between these registers are marked as read as all zeroes (RAZ), allowing ORRIS to allocate a single watchpoint to monitor them all [26]. Any attempt at altering these protected memory locations halts the CPU and returns control to ORRIS, effectively blocking any attempted write operation. This enables reliable JTAG access to the target embedded device, further securing our hardware root of trust. Such a watchpoint-based approach requires an established JTAG connection to the device before an attack initiation by the adversary. This assumption allows ORRIS to enable the watchpoint to protect critical debug registers from malicious modifications by the adversary.

B. Dataset and Preprocessing

Goodware and malware. In literature, malware research predominantly focuses on Windows or Linux OS running on x86_64 and Android on ARM. Due to the limited interest in malware detection for Linux on ARM, we did not come across any prior openly available dataset of ARM ELF malware samples, limiting the overall dataset size. We manually collected 1,135 malware ELFs from VirusTotal [27] and VirusShare [28]. All the malware ELF files were processed on VirusTotal to retrieve metadata information, allowing us to perform additional analyses. The metadata collected consists of timestamp information, first_seen, the date on which the VirusTotal scanners first saw a specific malware as presented in Table II. This additional information enables us to create a dataset based on the monthly release of the malware and study the performance of ORRIS against previously unseen malware samples. Furthermore, the metadata has enabled us to classify the malware dataset into various families, as shown in Table III. This dataset is a diverse collection of malware belonging to various families, each family containing numerous variants. On the other hand, we use 884 goodware Linux applications such as `git`, `nano`, `vim`, and some common Linux binaries such as `ps`, `ls`, `cat`, and more.

Also, the availability of ARM rootkits is further limited, forcing us to port a diverse set of x86-based rootkits we found available in online repositories. Porting user-level rootkits required changes in the installation script, whereas x86-based kernel-level rootkits demanded modifications in the syscall table hooking mechanism. Table IV presents more information about the type of system calls targeted and their modified functionality. The limited number of samples of rootkits is due to the lack of their source code availability. We require the source code of these rootkits to make them compatible with ARM, and therefore compiled binaries (present in x86 rootkit datasets) were not suitable for our experimentation. In total, we have 425 shared libraries, 4 user, and 11 kernel-level rootkits in our dataset. We want to emphasize that the collected malware samples target Linux on ARM, an environment actively utilized by deployed PLCs in the field as shown in [29] and do not specialize in attacking industrial process logic (not PLC malware).

Collecting traces. As mentioned previously, we assume that the adversary can escalate their privilege on the compromised PLC and execute malicious binaries. This requires ORRIS to utilize hardware (trusted), protect the kernel-level from rootkits, build trust, and then use the kernel-level information to protect against malware running at the user-level.

For kernel-level rootkits, ORRIS monitors *write* operations on the static code region of the Linux kernel. In our dataset, kernel-level rootkit samples target the syscall table, which redirects the execution flow of regular system calls to their malicious counterparts. An exception is raised each time an entity attempts to write in this region. This event is captured by relying on hardware watchpoints. On the other hand, user-level rootkits write their absolute path to `/etc/ld.so.preload`. This file is then read by the dynamic linker each time a new userspace process is executed, preloading the user-level rootkit for all the subsequent processes. To protect against user-level rootkit injection, we track it at the kernel level by monitoring system calls such as `open`, `read`, `write`, `close` and `delete_module`. It is important to monitor the `open` system call to acquire the file descriptor assigned to `/etc/ld.so.preload`. It should be noted that the detection is only triggered when `sys_write` is called on `/etc/ld.so.preload` file.

We run Lauterbach Trace32, our software debugger of choice, on an external host and connect JTAG to the test device for malware detection. We collect semantic information from the PCB at the kernel level and microarchitectural event counts from ETM. Due to the limited availability of ETM counters, two in our case, and its 16-bit limited size, we only obtain counts for two events at a time and repeat the same data collection process for all the available microarchitectural events. ORRIS identifies anomalies in the overall system operation rather than on a specific test application. So, we collect information about the entire system and perform preprocessing on it.

Manual scrubbing. ORRIS appropriates information such as the number of segments, dynamically linked functions in the Procedure Linkage Table (PLT), boolean value for the presence of symbol resolution functions `dlsym` and `dlvsym`, function count, function degree, cumulative instruction count, instructions per function, load, store, move, shift and branch instruction count data collected by a disassembly tool, in our case, Radare2, aiding in the detection of user-level rootkits.

On the other hand, we collect counts of all the 49 microarchitectural events available for malware detection and perform a manual scrubbing to shortlist promising candidates of semantic features. Following in the footsteps of previous work such as [30], we remove candidates that do not provide any insight into the activities of a process, for instance, parameters such as constants, static identifiers, parameters with value zero, process identifiers, and memory addresses. This process of elimination resulted in a total of 54 selected semantic parameters, some of which are `se_vruntime` (runtime of a thread), `majFlt` (major page faults), `minFlt` (minor page faults), and `ioac_syscr` (number of read syscalls). Some microarchitectural events are `ERETURN` (exception return instruction), `DUNALIGNED` (unaligned data access), `NEONWORK` (integer unit not idle), and `UNALIGNEDREPLAY` (replay events from unaligned access). All the features and their brief explanations are presented in the Appendix, Table IX and Table X. After selecting the attributes, we take the average values of the events aggregated over measurement cycles. We normalize the average values for the number of processes running at the time of data collection and remove any recorded measurement of less than 1 second since a small quantity can misguide the training process.

Pre-processing technique. The attributes discussed before have widely different numerical ranges. Therefore, we apply standard data pre-processing schemes applicable to such integer-valued functions to empirically find the best pre-processing technique. Standard scaling (s) performs the best for our dataset, which involves removing the mean of the training data from each data-point and dividing it with its standard deviation. Additionally, we also perform Principal Component Analysis (PCA) on the 13 features collected for user-level rootkit detection. We prepare the data into training and testing sets with an 80-20 ratio to evaluate our malware detection methodology. There are a total of 103 features considering both high-level and low-level features.

C. Detection Methodology

Approach. For kernel-level rootkits, ORRIS protects the syscall table from being modified by setting a hardware watchpoint over the address range of the syscall table, instructing the Memory Management Unit (MMU) to establish the memory pages containing the monitored address range as write-protected. MMU raises an exception when a kernel-level rootkit attempts to write in this address range. ORRIS then calculates the address of the attacked system call and the hook address. This calculation is intrusive because it halts the CPU for reading memory and registers.

Whereas, the `sys_write` condition on `/etc/ld.so.preload` triggers the protection against the injection of user-level rootkits, performing outlier detection. It uses Radare2 to perform static analysis for extracting features from the binary. We use the top 3 principal components in one-class SVM (OCSVM) for outlier detection. It should be accentuated that we only use in-the-wild rootkits for our experiments, resulting in limited samples for user-level rootkits. We use one-class SVM outlier detection instead of traditional machine learning approaches to counter the imbalance in samples available for shared libraries and user-level rootkits. Such techniques have also been used in literature for malware detection [31]. In general, one-class techniques are used in severely skewed class distribution, fitted on the input examples from the majority class in the training dataset, and evaluated on a test dataset. The model uses a Radial Basis Function (RBF) kernel, commonly used with SVM classifications. This model serves as the signature for legitimate shared libraries, and all the classified outliers are considered user-level rootkits.

TABLE V: A summary of best performing models.

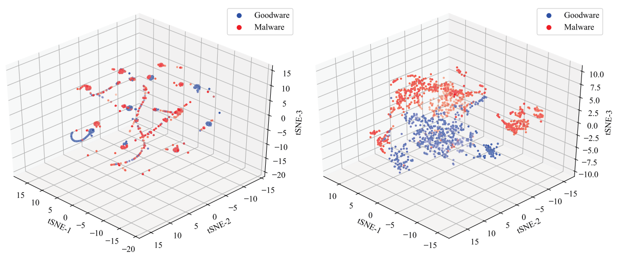
Pre-processing	ML algorithm	Feature extraction	Dataset	No. of features	Test accuracy	FNR
Standard	SVM	No statistic	Sem + HPC	103	0.997525	0.004484
Standard	SVM	No statistic	HPC	49	0.987624	0.013453
Standard	SVM	No statistic	Sem	54	0.928218	0.044843
Standard	SVM	MI	Sem + HPC	64	0.997525	0.004484
Standard	SVM	f-score	Sem + HPC	103	0.997525	0.004484
Min-Max	logreg	MI	Sem + HPC	71	0.980198	0.017937
Max-Abs	logreg	f-score	Sem + HPC	79	0.977723	0.026906
Max-Abs	GNB	f-score	HPC	12	0.851485	0.03139
Max-Abs	GNB	MI	Sem + HPC	13	0.881188	0.049327
Min-Max	logreg	χ^2	Sem + HPC	103	0.980198	0.013453

We force the ML model to learn different malware and goodware features using Support vector machines (SVM) with a linear kernel for malware detection. We also report the False Negative Rate (FNR) as it captures the percentage of malware deemed genuine, causing damage. Other statistics like false-positive rates determine whether benign software was detected as malware, which hampers usability but does not damage the system.

Best pre-processing technique. Standard scaling performs the best to detect malware, providing the highest accuracy of 99.75%, followed by max-abs and min-max scaling with the best test accuracy of 98.01%. Therefore, in our dataset, standard-scaling makes the two clusters more distinguishable. We perform t-distributed Stochastic Neighbor Embedding (t-SNE) to visualize the data. Fig. 2a shows that when the features are unprocessed, it is challenging to have a clear decision boundary between the malware and goodware. However, when we standardize the data and sort it according to mutual information score, the resulting transformation of the data aids in classification, as shown in Fig. 2b.

Base architecture. We choose OCSVM for user-level rootkit detection after extensively testing other outlier detection approaches. Isolation Forest (ISO), Minimum Covariance Determinant (MCD) (in this case, Elliptic Envelope implementation), and Local Outlier Factor-based approaches do not perform well when deployed. ISO, MCD, and LOF give an accuracy of 88.4%, 82.9%, and ≈88%, respectively.

For detecting malware, we apply three statistical scores, namely Mutual Information (MI), f-score (F), and chi-squared (χ^2) statistic, to select k best features before training a machine learning model. We deploy three supervised ML classification algorithms commonly used in classification tasks: Support Vector Machine (SVM), Logistic Regression (LR), and Gaussian Naive Bayes (GNB). We report the best performing models with different combinations of pre-processing techniques, ML algorithm used, underlying feature-extraction methods taking microarchitectural/hardware performance counter (HPC) events, and semantic features, individually and together in Table V. We achieve the maximum test accuracy of 99.75% with SVM even without using an underlying feature extraction technique (row 1). However, applying MI between output labels and features



(a) TSNE on un-processed data. (b) TSNE of scaled features sorted according to mutual information scores.

Fig. 2: Dataset visualization using TSNE.

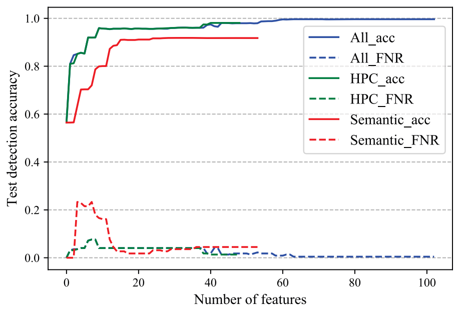


Fig. 3: Change in test accuracy and FNR as a function of standard scaled features.

reduces the feature set by 39 while achieving the same maximum test accuracy, decreasing the latency in detection. While comparing the different models chosen to perform malware detection, we observe that SVM and logistic regression perform similarly with the best test accuracy as 99.75% and 98.01%, respectively. However, the performance of GNB is significantly lower, with its best being 88.11%.

Best feature extraction approach. The smallest number of features required for fast detection is with the combination of mutual information and GNB, but the highest test accuracy achieved with this technique is 85.14%. The lowest value of FNR, 0.484%, is accomplished using SVM. MI and f-score perform the best with 99.75% and 0.484% as test accuracy and FNR, respectively, while the best test accuracy achieved by χ^2-statistic is 98.01%. Considering the number of features, MI outperforms the f-score by 39 fewer features.

Dominant feature type. For malware detection, we observe that a model performs better in terms of higher test accuracy and lower FNR when high-level and low-level features are taken together instead of individually. We achieve the highest test accuracy and the lowest FNR of 99.75% and 0.484%, respectively, with standard-scaled features using SVM. However, if we consider the HPC or the semantic features alone, the performance drops to 98.76% and 92.82% for test accuracy, and 1.34% and 4.48% for FNR, respectively. Individually, HPCs perform better than the semantic features, which ex-

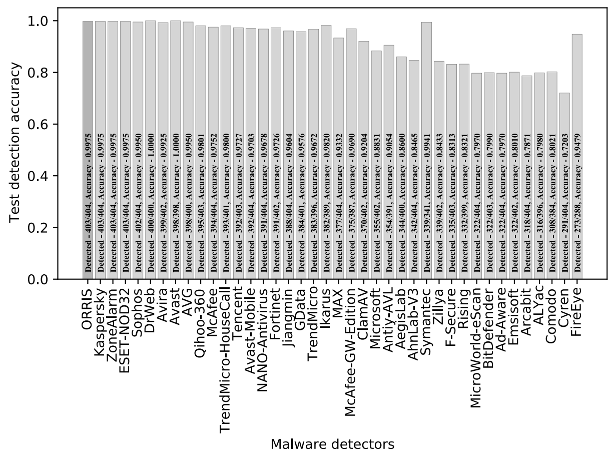


Fig. 4: Accuracy of ORRIS compared to commercial products.

plains their popularity in the literature. According to this experiment, the top 27 features based on their scores are microarchitectural event counts, followed by sparingly few semantic features. Nevertheless, in this study, we observe that a malware detection algorithm performs the best when the microarchitectural and the semantic features are combined. We depict the increase in test accuracy and decrease in FNR as a function of using relevant features in Fig. 3.

D. ORRIS vs. Commercial Products

As shown in Fig. 4, our framework ORRIS is comparable in accuracy with the best performing commercial malware detection solutions. For instance, ORRIS, Kaspersky, ZoneAlarm, and ESET-NOD32 all have a high test accuracy of $\approx 99.75\%$. It should be mentioned that DrWeb and Avast give an accuracy of 100% but do not process all the 404 test samples; instead, they handle only 400 and 398 samples, respectively. Due to this drawback, DrWeb and Avast are not the top-performing malware detection solutions in our result comparison. While the signatures of commercial malware detectors are regularly updated, the results on VirusTotal are calculated once (when the malware appears on the scanner for the first time) and returned upon a query unless explicitly specified by the user rescan a particular file. This behavior is evident from the results where some malware detectors cannot detect specific dated malware samples. So, such a comparison with commercial malware detectors helps us understand how ORRIS compares to the state-of-the-art solutions and is worthwhile.

E. Proof-of-Concept Real-time Implementation

For kernel-level rootkit, ORRIS utilizes a hardware watchpoint set directly through JTAG on the syscall table and calculates information such as a hooked system call address. This solution prevents kernel-level rootkit injection into the kernel space, sanitizing it and creating the first link in the chain of trust from hardware to the kernel level.

Whereas for protecting against user-level rootkits, ORRIS runs in the kernel space of the PLC. We envision this solution as a Linux kernel component, including the Radare2 binary and the required outlier detection libraries. Nevertheless, for rapid prototyping, we implement it as an LKM. All the system calls to be monitored are hooked by the LKM, which include `open`, `read`, `write`, `close`, `delete_module` and `dup2`. When `sys_write` condition is satisfied on `/etc/ld.so.preload`, ORRIS creates a kernel helper thread for executing userspace processes. This process spawns with the *UMH_WAIT_PROC* condition, which waits to complete the client process to finish before returning control. This prevents the initial `sys_write` on `/etc/ld.so.preload` until the analysis concludes. Moreover, we also assume that the adversary might gain kernel privilege for injecting rootkits. This consecutively also enables the adversary to remove the LKM by using the `rmmod` command. To prevent the removal of our proof-of-concept (PoC), the LKM also hooks to `delete_module` syscall. Upon deletion, it throws EBUSY: device busy or locked error to the adversary, preventing its injection. In the final product, user-level rootkit detection in ORRIS should ship as a part of the kernel and not as an LKM, to prevent its removal.

```
User-level Protection Injected ...
SYS_OPEN: /etc/ld.so.preload fd: 3
SYS_DUP2: /etc/ld.so.preload Old fd: 3 New fd: 1
SYS_WRITE: /lib/bdvl-UIBDEr5hJgUU.so.armv7l
User-level Client Spawning
User Level Message: Hello
User Level Message: MALICIOUS
User-level Rootkit Detected
RM: File Deletion Process - /lib/bdvl-UIBDEr5hJgUU.
    ↪ so.armv7l /etc/ld.so.preload
User-level Protection Removed ...
```

Listing 1: Debug messages from user-level rootkit protection.

Listing 1 shows the sequence of events that occur inside the LKM. As shown, ORRIS tracks calls to `sys_open` for the file `/etc/ld.so.preload`. It spawns a client process in userspace upon detection of `sys_write` on this file. The client then communicates with the LKM, gets the file path for the user-level rootkit binary, and returns the final result. If it is detected as malicious, the LKM deletes both the user-level rootkit (in this case /lib/bdvl-UIBDEr5hJgUU.so.armv7l) as well as the file `/etc/ld.so.preload`.

Since the PoC implementation for user-level rootkits requires syscall table hooking for tracking write system calls on `/etc/ld.so.preload`, it will not function alongside kernel-level rootkit detection. We introduce a formal notation that allows modifying some basic initial values such as watchpoint address range and individual system call addresses, making ORRIS extensible and easy to use.

```
[sys_write]
exception = enable
address = read_memory
```

Listing 2: Formal notation for specifying syscall exceptions.

We use this functionality to integrate user-level and kernel-level rootkit protection, as shown in Listing 2. We specify the system call whose default signature is modified

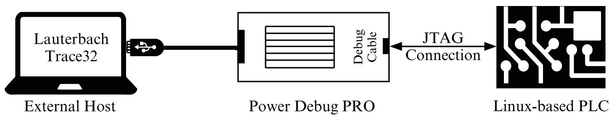


Fig. 5: Experiment setup with the external host running ORRIS and JTAG connection utilized for data collection.

(`sys_write`). Specifier exception can be either set to *enable* or *disable* for each system call. Finally, the specifier address instructs ORRIS on how to modify this signature. System call addresses can be hard-coded in the specification, or *none*, for directing ORRIS to ignore this system call entirely during detection. It can also be *read_memory* where ORRIS uses JTAG (without halting the CPU by using processor memory bus) to fetch the current system call handler address from the syscall table, enabling both solutions to operate in conjunction.

As mentioned before, our malware dataset consists of trace values collected during malware execution, utilized for training the SVM-based ML model. *Sampling rate* and the *number of features* to be collected are the two critical parameters recognized for real-time implementation. Here, the sampling rate refers to the interval between subsequent trace collection, contributing to the detection latency. Real-time malware detection requires swift data collection and processing without significant intrusiveness on the protected device. We observed that microarchitectural/HPC features contribute more than semantic ones toward detecting malware. As mentioned before, the top 27 features selected per MI experiment are low-level event counts. Moreover, gathering semantic information accrues a higher latency for a real-time mode of operation. This is because ORRIS traverses through the PCB of all the individual processes running on the test device. Based on the feature scores of microarchitectural event counts and the latency requirement for deployment, we decided to use only microarchitectural event counts in PoC implementation. We varied the sampling rate from 0.5 seconds to 10 seconds, increasing it by 0.5 seconds and studying malware detection accuracy by gradually increasing the number of features sorted based on MI. Such an exploratory study allowed us to create different ML models for a diverse sampling rate and the number of features collected.

V. Results

A. Experimental Setup

For our experiments, we replicated a Linux-based PLC by using a BeagleBone Black Rev C (BBB) loaded with CODESYS PLC stack running a custom JTAG enabled real-time Linux kernel. BBB carries the TI AM335x chipset and can run Linux OS, a combination commonly available in industrial automation devices, for instance, WAGO PFC100 Controller. Furthermore, BBB provides us with a readily available and accessible JTAG port.

To utilize the JTAG connection, we use Lauterbach Trace32 running on the external host as our software debugger of choice, paired with Power Debug Pro as the JTAG adapter. We

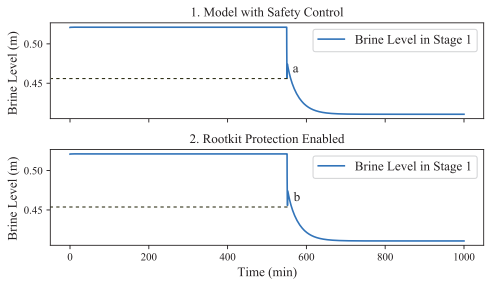


Fig. 6: Impact of ORRIS on a desalination plant.

use Lenovo T440s with Intel Core i7-4600U CPU running at a base clock of 2.10 GHz and a RAM of 8 GB as the external host for our experiments. To facilitate the development of ORRIS, we created a wrapper around the Python API provided for Trace32, aiding the communication with JTAG.

As shown in Fig. 5, Trace32 connects to Power Debug PRO with a USB connection, which facilitates communication to the BBB with a JTAG connection. Here, BBB represents the PLC device to be protected against malware attacks. To study the impact of intrusive operations performed by kernel-level rootkit protection on real-world OT devices, we modify a Hardware-in-the-loop (HIL) setup for the Multi-Stage Flash (MSF) desalination plant model as used in [32]. The primary objective is to test the effect of the intrusive operations on a critical control function of a 22 stage thermal desalination plant model. This safety control monitors the steam flow to the brine heater and maintains a safe operating temperature.

B. Kernel-level Rootkit Protection

ORRIS detected all the 11 tested kernel-level rootkits. This process takes an average of around 94.5 ms. This latency is due to the calculation of the malicious hook and targeted system call addresses. Latency further decreases to 8.64 ms by skipping this information-gathering step. On the other hand, when decompiling the malicious hooked function, the latency increases to 527.18 ms.

Quantifying intrusiveness. Finally, we also analyze the impact of intrusive operations in ORRIS on a HIL model of an MSF desalination plant. There is safety control on the steam input to the brine heater for closing its valve when the top brine temperature increases or decreases below a certain threshold. The regular operation of this model with this safety control implemented in HIL on a BBB running CODESYS control is as shown in subplot 1 of Fig. 6. Point a in subplot 1 shows the instance where safety control is activated to shut off the valve leading to a graceful decrease in the brine level. We run ORRIS on a remote host and connect to the JTAG interface of BBB running the safety control. Finally, *Diamorphine*, a kernel-level rootkit, is injected simultaneously when the safety control is triggered. This instance is shown by point b in subplot 2 of Fig. 6, and as seen, ORRIS does not affect the real-time requirements of the device while preventing rootkit injection.

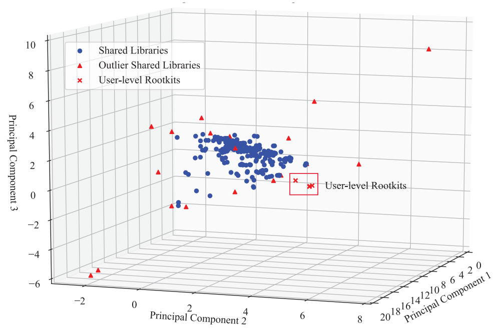


Fig. 7: Outlier detection to identify user-level rootkits.

C. *User-level Rootkit Protection*

We extract static features from binaries of 425 shared libraries, 4 user-level rootkits and run the OCSVM algorithm for detecting outliers. We get an average accuracy of 96.3% with a true positive rate (TPR) of 100% and a true negative rate (TNR) of 96.2%. A TPR of 100% implies precise identification of all the tested user-level rootkit samples. In contrast, a TNR of 96.2% emphasizes that 3.8% of legitimate shared libraries were detected as outliers (user-level rootkits), ≈16 out of 425 shared library samples. According to our hypothesis, user-level rootkits are identified as outliers as they are functionally different from legitimate shared libraries, culminating in particular changes in their static features. Fig. 7 visualizes the results for our user-level rootkit detection. As shown, shared libraries form a cluster (shown in blue circles). In contrast, user-level rootkits are grouped close together (shown in red X) and distant from shared libraries. Some shared libraries are detected as outliers and are considered false positives, a common drawback of classification, also evident in modern antivirus solutions. For instance, according to a recent report published by AV Comparatives, half of the 12 products tested gave high false positives, including products from Microsoft and Trend Micro [33]. ORRIS removes these user-level rootkits after detection.

D. *Malware Protection*

We study different combinations of sampling rate and the number of features to present a snapshot of the result in Fig. 8a with a maximum of 10 features considered at a time. From the results, we make some critical observations that guide our approach towards real-time malware detection.

1) All the presented sampling rates reach an accuracy of more than 90% when considering at least 10 features.
2) Sampling rate of 6 seconds (represented by black in Fig. 8a) achieves the best performance with a limited number of features. This accrued latency might not be acceptable in real-time malware detection, so instead, we choose a sampling rate of 2.5 seconds, which provides the right balance between performance and latency.

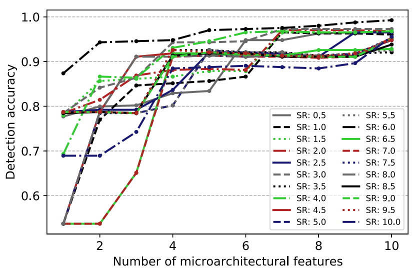
Furthermore, to maintain the non-intrusive operation of ORRIS, we chose to use ETM instead of the performance monitoring unit (PMU) counters, restricting the feature set to just two microarchitectural event counts. We create different ML models for various configurations, ready to be utilized based on the requirements, enabling ORRIS's diverse application. Based on the configuration, ETM counters generally vary between one to four for ARM processors; on the other hand, modern Intel CPUs support three fixed and four programmable counters per core. ORRIS supports such distinct configurations by creating separate ML models for various available microarchitectural counters. The sampling rate of 2.5 seconds performs well while accruing reasonable latency with only two microarchitectural features and is chosen for our requirements. From our experiments, we ascertain that microarchitectural events `NEONWORK` (NEON and integer unit not idle) and `AXIWRITE` (AXI write active), when used together, give reasonable accuracy for a sampling rate of 2.5 seconds with limited features. Using these two features, we train the SVM model with the collected traces.

Accuracy of malware detection. We deploy the model trained with `NEONWORK` and `AXIWRITE` microarchitectural traces sampled at a rate of 2.5 seconds, achieving a testing accuracy of 85.64% with a true positive rate of ≈0.90 and a true negative rate of ≈0.81.

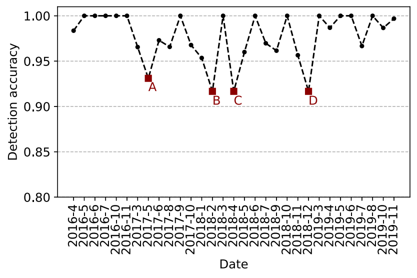
TABLE VI: Latency for real-time malware detection.

Sampling Rate (T_S)	**Processing Latency (T_M)**	**Prediction Latency (T_P)**	**Total (T_S+T_M+T_P)**
0.5	0.258698	0.158736	0.917434
1	0.226739	0.166595	1.393334
1.5	0.227891	0.170549	1.89844
2	0.258784	0.164858	2.423642
2.5	0.208440	0.238542	2.946982
3	0.227576	0.150379	3.377955
3.5	0.201644	0.153658	3.855302
4	0.173599	0.161399	4.334998
4.5	0.409511	0.166679	5.07619
5	0.253542	0.145607	5.399149
5.5	0.228152	0.157058	5.88521
6	0.257622	0.146325	6.403947
6.5	0.240420	0.164183	6.904603
7	0.207519	0.180537	7.388056
7.5	0.246482	0.204154	7.950636
8	0.221263	0.192079	8.413342
8.5	0.229527	0.205732	8.935259
9	0.220742	0.148972	9.369714
9.5	0.272092	0.152298	9.92439
10	0.238663	0.170186	10.408849
Average	0.2404453	0.1699263	-

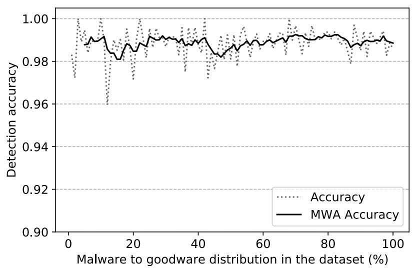
Latency in real-time malware detection. Microarchitectural traces are collected after every T_S seconds, dubbed as the sampling rate. A buffer on the external host stores these trace values, adding to the latency by T_M. These traces are pre-processed and passed to the trained SVM model for prediction. T_P denotes the time taken for pre-processing and prediction. So, the processing latency for online module can be roughly given by $T_S + T_M + T_P$. Both T_M and T_P are dependent on the configuration of the external host. The values presented here apply to our configuration and might differ for other systems. Table VI presents the average latency for all the tested sampling rates. Our selected model takes a total of ≈2.95 seconds for detecting malware in real-time.



(a) Exploring ML models for real-time detection. *SR* represents the sampling rate in seconds.



(b) Points A, B, C, and D represent the points with accuracy drop.



(c) Accuracy for the dataset by varying malware to goodware ratio, studying spatial experimental bias.

Fig. 8: Studying real-time malware detection models, performance on unseen malware, and Spatial experimental bias analysis

Performance against previously unseen malware. For studying the performance of the SVM model against unseen malware, we divide the malware dataset according to the timestamp metadata. The timestamp used in our study is called first_seen, which corresponds with the date when VirusTotal online malware scanners first saw a particular malware. Table II shows a snapshot of this dataset, divided according to the timestamps.

We use an incremental learning approach [34] that updates the ML model upon the arrival of new test samples, adding to all the other prior samples. To aid this experiment, for each month, we divided the samples into two groups: one containing samples up to a particular month of a year used for training and the other one that encompasses samples from that specific month utilized for testing. Consider 2017-5, where all the samples before May are training the SVM model, and all the samples collected in May are for validation. However, goodware applications are distributed equally across all the months. This experiment represents the real-world scenario where new malware and the variants of previously known malware release at regular intervals. Such situations might lead to a decrease in the accuracy of an untrained ML classifier. As shown in Fig. 8b, points A, B, C and D are the months that cause significant accuracy dip, requiring model retraining. We manually analyzed the test datasets corresponding to these points and present a summary of our findings in Table VII.

Although accuracy deteriorates at points A, B, C, and D, the overall accuracy of malware detection does not drop below 91.6%. This consistency in the performance of ORRIS is because our solution does not rely on static binary features that change significantly between different malware and their variants. Instead, it utilizes system behavior on the kernel and microarchitectural levels for malware detection. This behavior might remain consistent across malware variants, culminating in reduced degradation of the ML model.

Studying spatial bias. For studying spatial experimental bias, we need to estimate the malware to goodware ratio for Linux on ARM. Statistics about the percentage of malware to goodware have been reported in the literature for Android environments. Unfortunately, such studies do not exist for Linux on ARM. As a result, in this analysis, we vary the malware-to-goodware ratio from 0.01 to 1 and study the accuracy of the ML model for each step.

TABLE VII: Summary of reasons for accuracy decrease.

Drift Point	Reason	Malware
A	Limited training on test malware variant	Mirai.N
B	New malware variants	Dofloo.D, Mirai.Au, Gafgyt.Az, Gafgyt.Ak, Mirai.Ax, Gafgyt.Aj, Dofloo.F and Tsunami.Bh
	New malware	Mirai.B and DnsAmp.C
C	New malware variant	Tsunami.Br
D	Limited training on test malware variants	Tsunami.Bh and Mirai.Au

As shown in Fig. 8c, when the ratio is small, the classifier is mostly guessing the label of the executing sample. As a result, the accuracy fluctuates with considerable variations between 96% and 100%. We note that the accuracy decreases when introducing more malware samples into the dataset relative to the goodware samples. We used a moving window average (MWA) with a window size of 5 to observe the average accuracy. As evident, the MWA accuracy stays relatively impervious to spatial experimental bias, between 98% to 100%.

VI. Extending ORRIS

A. *Non-intrusive Rootkit Detection*

Similar to [35] and [36], any *proactive* rootkit protection mechanism needs to be intrusive to some extent to stop the rootkit injection in the first place. *Reactive* solutions (i.e., detection of an already injected rootkit) suffer from the fact that kernel rootkits aware of the detection mechanism can disguise themselves accordingly (a moving target). In any case, intrusive behavior might sometimes not be desirable. Here, we also discuss an alternative *reactive* approach to kernel-level rootkit protection with decreased latency and non-intrusive operations.

We create a golden model of the syscall table, described by the system call number, system call name, memory address, and system call handler address. The solution then scans the memory region of the syscall table using JTAG for a reactive detection of kernel-level rootkits. This method is non-intrusive since JTAG does not halt the CPU for regularly scanning the syscall table memory region as it can directly access the CPU memory bus. This is only possible for ARM processors such as Cortex-A5/A7/A8/A9/A15 and more, supporting built-in DAP on the chip with a memory bus connection. After the extraction of the memory region, this content is matched with the golden model of the syscall table. When an integrity violation is detected, the analysis module can traverse through the syscall table to identify the exact modified system call.

The approach mentioned above is reactive because it does not prevent the initial hooking of system calls; instead, it only detects it after successful rootkit injection. We also implement syscall table patching via JTAG to remove the injection of the kernel-level rootkit. The average time taken by the reactive approach for detecting kernel-level rootkits is around 4.7 ms. In contrast, when patching the syscall table is enabled, it increases significantly to about 145.5 ms. This increase in latency when compared to the default non-intrusive reactive detection approach is due to the halted CPU required for patching the syscall table by modifying memory content. While this approach provides low latency detection of a rootkit, it does not prevent the initial injection of the kernel-level rootkit into the kernel space. This shortcoming makes reactive approaches vulnerable to skillful attacks from rootkits targeting OSLAR register to disable JTAG access.

B. *Rootkit Detection Generalization*

In this research effort, we did not come across kernel-level rootkits that modify dynamic kernel data structures. So, ORRIS focuses on kernel-level rootkits that modify static regions in kernel memory, specifically ones that hook to the syscall table. However, there are variations of these rootkits that hook to other critical static components of the Linux kernel, for instance, the Exception Vector Table (EVT). This type of kernel-level rootkit tries to modify the execution flow of a system call before it reaches the syscall table by hooking to the EVT. To test the extensibility of ORRIS on kernel-level rootkits, we decided to analyze it on the EVT-based rootkit *arm-evt*, which is available online. This rootkit modifies the SWI exception vector to branch to a specific address that stores the adversary's backdoor. When triggered, this backdoor checks for the content of the `R7` register, and if it matches `0xb0000000`, it elevates the privilege level of the calling process. This type of kernel-level rootkit also writes to the kernel text section for modifying execution flow and is detected by ORRIS.

VII. Discussion

Limitations of ORRIS. Malware detection in ORRIS utilizes a combination of semantic and microarchitectural event counts. However, malware with an LKM counterpart can unlink itself from the PCB, removing its footprint from the semantic features. Its operations will still impact the microarchitectural state being recorded by ORRIS. Furthermore, it can also miss certain user-level rootkits that utilize the `LD_PRELOAD` technique, an environment variable that preloads shared libraries. However, this affects only the current process, not impacting the entire system, and consequently, we did not find any such user-level rootkits in-the-wild. Finally, Kernel Address Space Randomization (KASLR) can create complications in obtaining the address of `init_task`, hindering the collection of semantic event counts. Pre-4.8 KASLR shifts physical and virtual addresses with the same random offset, whereas 4.8 and later utilize variable offset, which can also be determined, a feature already available as a plugin in Volatility [37].

Advantages and limitations of JTAG. JTAG allows the malware detection process to be extended to legacy devices without significant modifications since JTAG is an IEEE standard. It also enables ORRIS to run out-of-the-device without relying on any other physical connection to the protected PLC. Utilizing a different port might require continuous supported OS updates from the manufacturer. Sometimes peripheral connections might become inaccessible in legacy devices due to a lack of software updates. Nonetheless, JTAG can still be used for establishing a connection with the device due to its low-level interface and minimal dependence on OS if already enabled in software. Moreover, JTAG also allows us to create a malware detection framework with a minimum overhead on the protected device, as it offloads the computation to an external device. On the other hand, JTAG ports are sometimes disabled physically in production devices or software by disabling certain debug flags in the kernel source code, necessitating manufacturer support. For instance, in BBB, we enabled *DEBUGSS* module clock in the kernel source code to allow communication through the JTAG interface. However, JTAG has been accessed in commercial off-the-shelf PLCs in [38] for Allen Bradley CompactLogix 5370 and [39] for Honeywell Experion C200.

Challenges of in-the-wild dataset. Rather than relying on synthetic malware, we experimented with in-the-wild samples, which limited the dataset's size. During the dataset creation, we encountered numerous problems. ORRIS requires samples that can execute on ARM architecture, while most of the openly available rootkit datasets consist of compiled binaries for the x86_64 architecture. As evident in Table VIII, in general, rootkit datasets are not massive. Furthermore, the availability of compiled binaries in these datasets also limits the porting of these rootkits to the ARM architecture. Therefore, we manually collected rootkit samples for x86_64 with available source code and ported them to the ARM architecture, which required modification to the syscall hooking mechanism. Furthermore, these collected rootkits only utilized the syscall hooking mechanism and did not modify any other dynamic kernel data structure. It affirms the fact that rootkits targetting the syscall table are more prominent than their counterparts.

Advantages over a pure software approach. Studies in literature utilize software techniques for collecting traces, as

TABLE VIII: A summary of malware detection mechanisms proposed in the literature.

Work		[40]	[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[35]	[36]	[49]	[50]	[51]	[52]	[53]	[54]	[55]	[56]	[57]	[58]	[59]	[30]	[60]	[61]	[62]	[63]	[10]	[12]	[64]	[65]	[47]	[66]	[48]	[67]	[68]	[69]	[70]	[71]	[21]	This Work
Plat-form	X	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○	●	○	●	●	●	●	●	●	●	●	●	●	○	○
	R	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	●	○	●	○	○	○	○	○	○	○	○	○	○	●	●
OS	W	○	●	○	○	○	○	○	●	○	○	●	●	○	○	●	○	○	○	●	○	●	●	○	●	○	●	○	●	○	●	●	●	○	○	○	○	●	○	●	○	○
	L	●	○	●	●	●	●	●	○	●	●	○	●	●	●	○	●	●	●	○	●	○	○	●	●	○	●	○	○	○	●	○	○	●	●	●	●	○	●	●	○	●
	A	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	●	○	●	○	○	○	○	○	○	○	○	○	○	●	○
Target	K	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	●	●	○	○	○	○	●	○	●	○	○	○	○	○	○	●
	U	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
	M	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○	○	●	●	●	●	○	●	○	●	●	●	●	●	●	●
Deploy-ment	I	○	○	○	●	○	○	○	●	●	●	●	○	○	●	○	●	●	○	●	○	○	○	●	○	●	○	○	●	○	○	○	○	○	○	○	●	●	●	●	●	○
	E	●	●	●	○	●	●	●	○	○	○	○	●	●	○	●	○	○	●	○	●	●	●	○	●	○	●	●	○	●	●	●	●	●	●	●	○	○	○	○	○	●
Method-ology	T	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	P	○	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	●	●	○	○	●	●	●	●	●	○	○	●	●	●	●	●	●
	V	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	S	○	○	○	●	○	○	○	○	○	○	○	○	○	○	●	●	○	●	○	●	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	●
	O	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○
	H	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○
	J	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	●
	C	●	●	●	○	●	●	○	○	○	●	●	●	●	●	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	D	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Virtual-ization		●	○	●	○	●	●	○/●	○	●	○	○	●	●	○	●	●	○	●	○	●	○	●	○	●	○	●	○	○	○	○	●	●	●	●	●	○	○	○	○	○	○
Dataset Size	G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	56	30	0	123,453	0	0	12,214	2,081	467	NA†	0	20	0	0	NA†	180	NA‡	21,116	116,993	884
	K	25	311	8	NA*	37	3	23	100	8	8	10	23	10*	NA*	NA*	NA*	NA*	1	1	1	2,200	0	0	0	0	9	5	0	0	0	0	5	0	8	0	0	0	0	0	0	11
	U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	42	30	22	5,560	0	0	17,366	91	1,087	NA†	0	11	0	8	NA†	253	NA‡	23,033	12,735	1,135
Exp. w/ unseen mal.		○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●
Spatial-Bias		○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●
Acc. (%)	K	96	≈96.2	100	NA*	100	100	100	≈99.91	100	100	100	100	100*	NA*	NA*	NA*	NA*	100	100	100	98.15	-	-	-	-	100‡	100‡	-	-	-	-	≈99.9	-	100‡	-	-	-	-	-	-	100
	U	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	≈96.3
	M	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	≈97	93	100‡	94	-	-	96	≈95	≈92	99.5	-	≈96.9	-	100‡	100	95.2	99.8	≈93.2	78	≈99.7

X: x86_64 R: ARM W: Windows L: Linux A: Android K: Kernel-level Rootkits U: User-level Rootkits M: Malware I: Internal E: External T: Taint Analysis
P: Machine Learning V: Guest View Casting S: Integrity Verification O: Objective Oriented Association H: Threshold Based J: JTAG D: Data Invariant
C: Control Flow Checks/Symbolic Execution NA: Not applicable '$*$': Solution can be extended for rootkit protection '$\dagger$': Not mentioned in text
●: Parameter is studied or utilized '$\ddagger$': High accuracy due to limited test dataset size Blue Rows (3,8,18,24,26,27,29): Important contributions of this work

shown in Table VIII. Some methods isolate the protected device using virtualization, for instance, [67], leading to an overhead of about 1.5%, which is not desirable for PLC devices, limiting its applicability. Some approaches run their solution on the same device, which might enable a cunning adversary to design malware with evasion capabilities. Therefore ORRIS utilizes JTAG to extract data without adding significant overhead and moving the computation out of the device. Furthermore, using JTAG, an IEEE standard, makes the approach extensible.

Advantages of using semantic and microarchitectural information. Malware detection approaches relying solely on semantic (high-level) data can be evaded by targeted malware [72]. For instance, solutions that rely only on static analysis of malware binary are circumvented by inserting dead-code, subroutine reordering, code encryption, and code compression. To prevent such evasion techniques, researchers employ microarchitectural event counts. While malware can evade the solutions that rely on semantic information, it is difficult to erase or modify its footprint on the microarchitectural level. Unfortunately, solutions that only rely on microarchitectural features disregard context-aware information that is available with higher abstraction. As evident in Table V, microarchitectural and semantic information together gives an accuracy of 99.5%, more than when considered individually.

Shortcomings of a broader sampling window. Real-time implementation of ORRIS utilizes a sampling rate of 2.5 seconds, implying that the microarchitectural features are collected every 2.5 second. A longer sampling window enables an adversary to perform malicious actions during the sampling window and pause its activities before acquiring microarchitectural event counts. Such activities might not significantly impact the feature values, limiting ORRIS from identifying malware that executes and exits quickly.

Using PMU instead of ETM. Using the performance counter allows ORRIS to use four features instead of two features available with ETM, resulting in better performance. However, using PMU instead of ETM requires ORRIS to halt the CPU of the protected device to extract microarchitectural features, resulting in intrusive behaviors. Such an approach might perform better but not be suitable for application in PLC devices due to their real-time processing requirements. So, for malware detection in ORRIS, we decided to go with ETM instead of the PMU.

Protection against mimicry attacks. In literature, the introduction of secret randomization has been used in ML models to prevent mimicry attacks [73]. In the context of our real-time malware detection approach, multiple models are created with a combination of features (microarchitectural and semantic), arbitrary sampling rates. Due to randomization, the adversary is unaware of the specific model employed to identify the protected device's malware activities. Evading such a situation would require an adversary to know the ML model currently used. Alternatively, change the malware source code every time the model changes, which is not scalable for widespread attacks.

Correct malware labels. Authors in [74] found out that malware labels from VirusTotal stabilize in about a year and can lead to misclassification. To avoid such a situation, we only consider malware samples observed till 2019, goodware applications preinstalled in various Linux variants, such as `git`, `nano`, `vim`, and some system binaries.

VIII. Related Work

A comprehensive summary of all the proposed approaches in literature with specifics on supported OS, architecture, technique, dataset size, performance result is presented in Table VIII.

A. *Kernel-level Rootkit Protection*

A wealth of literature exists for protection against kernel rootkits. Petroni Jr et al. propose a state-based control flow integrity (SBCFI) technique in which a state-based monitor periodically audits the state of a system, compares with a golden image, and successfully detected 24 out of the 25 test rootkits (96%) [40]. Whereas, [46] introduces Gibraltar, which utilizes data structure invariants (properties that must remain unchanged during their lifetime) to automatically detect undesirable changes in the state of the kernel, indicating a presence of a kernel-level rootkit. It detected all the tested 23 rootkits, with 20 seconds latency and a 0.49% performance overhead. [47] and [48] utilize HPC observations, [63] proposes manual integrity verification of critical memory regions and [50] with PoKeR routing all the kernel instruction fetches to shadow memory. In contrast, all the other memory access is performed via standard memory. Finally, [58], and [41] which use binary analysis and control flow graph-based techniques, respectively.

B. *User-level Rootkit Protection*

To the best of our knowledge, there has only been limited work on detecting user-level rootkits. A patent by Douglas et al. [75] compares command outputs from user space with corresponding low-level functions in kernel space. Discrepancies between these results indicate the presence of a hidden user-level rootkit. This approach assumes that the adversary does not hold kernel-privilege, an invalid assumption, effectively enabling an adversary to manipulate output from kernel space and render the solution ineffective.

Another approach put forth in [76] targets user-level rootkits in Windows, relying on API interception for achieving stealth. This work utilizes a diff-based approach by comparing the output gathered while enumerating files and registry entries through infected APIs and a clean system. This work does not focus on user-level rootkits utilizing preloading techniques in Linux. These limitations make ORRIS the only solution that can protect against *both* user-level and kernel-level rootkits in Linux.

C. *Malware Detection*

Semantic Information. Panorama, as presented in [59] performs taint analysis with fine-grained information flow tracking, [30] utilizes mined information from PCB, VMwatcher [60] uses guest view casting to systematically reconstruct internal aspects of a virtual machine (VM) and check for irregularities. [61] proposes the static analysis of dex code, Fluorescence [62] relies on a signature-based approach, [77] with negative-day malware detection and [78] which detects attack from log analysis.

Microarchitectural Information. Ozsoy et al. proposed a Malware Aware Processor augmented with a hardware-based online malware detector [64]. HPCMalHunter [66] performs real-time behavioral detection at the hardware level. [65] propose an anomaly-based hardware malware detector utilizing signatures from low-level events. [47], [79], [80] and [81] where the authors study the feasibility of using performance counters, detection of kernel rootkits in [48], [82], malware detection in [67], [68], and [69] and DDoS attack detection in [70].

D. *Comparison*

As evident in Table VIII, x86_64-based Windows and Linux platforms dominate malware detection research. These techniques require virtualization, complex data collection techniques, localized presence in the protected host, and much more. ORRIS, on the other hand, only requires an accessible JTAG port to gather all the information, isolate itself externally, and do not use virtualization, making it a perfect fit for less powerful PLC-type devices while achieving high accuracy. It offloads the malware detection mechanism to an external host, only minimally impacting the PLC device while employing the JTAG interface features.

PLCs need specialized malware detection solutions that have only been touched upon lightly in literature. Konstantinou et al. proposed a primitive method for detecting malicious modification in the firmware of bare-metal embedded devices with JTAG [83]. Zonouz et al. utilized symbolic execution and model checking to recognize violations in process code running on a PLC [84]. Nevertheless, both approaches failed to create a comprehensive malware detection solution for PLC devices.

IX. Conclusion

In this research, we put forth ORRIS, a malware detection framework utilizing static data structure integrity verification, static binary analysis, semantic and microarchitectural features in Linux-based PLC. ORRIS uses only the JTAG connection to collect all the required information from the kernel and hardware level to detect malicious activities on a system. Our framework combines a software implementation and commodity hardware to form a chain of trust that proactively responds to malware targetting PLC devices. It keeps processing out of the device with minimal performance overhead on the protected device. We tested ORRIS against in-the-wild user-level rootkits and malware samples on a BBB as a test PLC device while achieving the highest accuracy of 96.3% and 99.75%, respectively. We also analyze our ML model against unseen malware and spatial experimental bias, confirming its resilience.

Acknowledgment

This research was supported by the NYU Abu Dhabi Global Ph.D. Fellowship. ORRIS can be found at github.com/momalab/orris.

References

[1] I-Scoop. (2020) Industry 4.0: the fourth industrial revolution – guide to industrie 4.0. [Online]. Available: https://www.i-scoop.eu/industry-4-0/

[2] L. Columbus. (2016) Industry 4.0 is enabling a new era of manufacturing intelligence and analytics. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2016/08/07/industry-4-0-is-enabling-a-new-era-of-manufacturing-intelligence-and-analytics/#11b2a3257ad9

[3] Markets and Markets. (2019) Industrial control systems (ics) security market by solution - global forecast to 2023. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/industrial-control-systems-security-ics-market-1273.html

[4] G. Murray, M. N. Johnstone, and C. Valli, "The convergence of it and ot in critical infrastructure," 2017.

[5] *Cybersecurity in Operational Technology: 7 Insights You Need to Know*, Ponemon Institute, March 2019.

[6] S. McLaughlin, C. Konstantinou, X. Wang, L. Davi, A.-R. Sadeghi, M. Maniatakos, and R. Karri, "The cybersecurity landscape in industrial control systems," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 1039–1057, 2016.

[7] M. B. Line, A. Zand, G. Stringhini, and R. Kemmerer, "Targeted attacks against industrial control systems: Is the power industry prepared?" in *Proceedings of the 2nd Workshop on Smart Energy Grid Security*, 2014, pp. 13–22.

[8] Gartner. (2018) Gartner forecasts worldwide information security spending to exceed $124 billion in 2019. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2018-08-15-gartner-forecasts-worldwide-information-security-spending-to-exceed-124-billion-in-2019

[9] P. Networks. (2015) How to break the cyber attack lifecycle. [Online]. Available: https://www.paloaltonetworks.com/cyberpedia/how-to-break-the-cyber-attack-lifecycle

[10] Y. Ye, D. Wang, T. Li, and D. Ye, "Imds: Intelligent malware detection system," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 1043–1047.

[11] T.-F. Yen and M. K. Reiter, "Traffic aggregation for malware detection," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 207–227.

[12] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *2012 European Intelligence and Security Informatics Conference*. IEEE, 2012, pp. 141–147.

[13] *WAGO-I/O-System 750 750-8202 PFC200 RS Manual*, WAGO, 2012.

[14] D. Greenfield. (2018) Why is linux trending? [Online]. Available: https://www.automationworld.com/products/control/blog/13318571/why-is-linux-trending

[15] B. T. Affair, "Hiding out under unix," *Phrack Magazine*, vol. 3, p. 25, 1989.

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

[17] N. Murilo and K. Steding-Jessen. (2007) Chkrootkit v. 0.43. [Online]. Available: http://www.chkrootkit.org/

[18] F. Merces. (2016) Trend micro. pokemon-themed umbreon linux rootkit hits x86, arm systems. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/pokemon-themed-umbreon-linux-rootkit-hits-x86-arm-systems/

[19] P. H. N. Rajput and M. Maniatakos, "Jtag: A multifaceted tool for cyber security," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2019, pp. 155–158.

[20] A. Infocenter. (2011) About embedded trace macrocells. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/I83164.html

[21] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 729–746. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury

[22] D. U. Case, "Analysis of the cyber attack on the ukrainian power grid," *Electricity Information Sharing and Analysis Center (E-ISAC)*, vol. 388, 2016.

[23] *ARM-ETM Training*, Lauterbach, 2020.

[24] Z. Ning and F. Zhang, "Understanding the security of arm debugging features," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 602–619.

[25] A. Developer. (2019) Cortex-a8 technical reference manual: Apb interface access permissions. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344h/Babdgjfh.html

[26] A. Developer. (2019) Cortex-a8 technical reference manual: Memory-mapped registers. [Online]. Available: https://developer.arm.com/docs/ddi0344/b/debug/debug-register-interface/memory-mapped-registers

[27] VirusTotal, "Virustotal," 2020. [Online]. Available: https://www.virustotal.com/gui/home

[28] VirusShare, "Virusshare.com," 2020. [Online]. Available: https://virusshare.com/

[29] D. Tychalas and M. Maniatakos, "Iffset: in-field fuzzing of industrial control systems using system emulation," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 662–665.

[30] F. Shahzad, S. Bhatti, M. Shahzad, and M. Farooq, "In-execution malware detection using task structures of linux processes," in *2011 IEEE International Conference on Communications (ICC)*. IEEE, 2011, pp. 1–6.

[31] E. Burnaev and D. Smolyakov, "One-class svm with privileged information and its application to malware detection," in *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2016, pp. 273–280.

[32] P. H. N. Rajput, P. Rajput, M. Sazos, and M. Maniatakos, "Process-aware cyberattacks for thermal desalination plants," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 441–452.

[33] A. Comparatives. (2019) False alarm tests. [Online]. Available: https://www.av-comparatives.org/testmethod/false-alarm-tests/

[34] F. A. Pinage, E. M. dos Santos, and J. M. P. da Gama, "Classification systems in dynamic environments: an overview," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 6, no. 5, pp. 156–166, 2016.

[35] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *20th Annual Computer Security Applications Conference*. IEEE, 2004, pp. 91–100.

[36] H. Yin, P. Poosankam, S. Hanna, and D. Song, "Hookscout: Proactive binary-centric hook detection," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2010, pp. 1–20.

[37] B. Neuburger. (2017) Implementing kaslr detection in volatility. [Online]. Available: https://bneuburg.github.io/volatility/kaslr/2017/05/16/KASLR3.html

[38] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics! attacking plcs with physical model aware rootkit." in *NDSS*, 2017.

[39] M. D. Schwartz, J. Mulder, J. Trent, and W. D. Atkins, "Control system devices: Architectures and supply channels overview," *Sandia Report SAND2010-5183, Sandia National Laboratories, Albuquerque, New Mexico*, vol. 102, p. 103, 2010.

[40] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 103–115.

[41] J. Wilhelm and T.-c. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 219–235.

[42] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008, pp. 21–38.

[43] M. Schmidt, L. Baumgartner, P. Graubner, D. Bock, and B. Freisleben, "Malware detection and kernel rootkit prevention in cloud computing environments," in *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2011, pp. 603–610.

[44] A. Baliga, X. Chen, and L. Iftode, "Paladin: Automated detection and containment of rootkit attacks," *Department of Computer Science, Rutgers University*, 2006.

[45] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring," in *2009 international conference on availability, reliability and security*. IEEE, 2009, pp. 74–81.

[46] A. Baliga, V. Ganapathy, and L. Iftode, “Detecting kernel-level rootkits using data structure invariants,” *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 670–684, 2010.

[47] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, “On the detection of kernel-level rootkits using hardware performance counters,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 483–493.

[48] X. Wang and R. Karri, “Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters,” in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2013, pp. 1–7.

[49] R. Riley, X. Jiang, and D. Xu, “Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008, pp. 1–20.

[50] R. Riley, X. Jiang, and D. Xu, “Multi-aspect profiling of kernel rootkit behavior,” in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 47–60.

[51] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, “Nested kernel: An operating system architecture for intra-kernel privilege separation,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 191–206.

[52] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, “Secure in-vm monitoring using hardware virtualization,” in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 477–487.

[53] A. Seshadri, M. Luk, N. Qu, and A. Perrig, “Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses,” in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 335–350.

[54] X. Xiong and P. Liu, “Silver: Fine-grained and transparent protection domain primitives in commodity os kernel,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2013, pp. 103–122.

[55] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, “Secpod: a framework for virtualization-based security systems,” in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 347–360. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/wang-xiaoguang

[56] S. Sparks and J. Butler, “Shadow walker: Raising the bar for rootkit detection,” *Black Hat Japan*, vol. 11, no. 63, pp. 504–533, 2005.

[57] X. Xie and W. Wang, “Rootkit detection on virtual machines through deep information extraction at hypervisor-level,” in *2013 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2013, pp. 498–503.

[58] S. A. Musavi and M. Kharrazi, “Back to static analysis for kernel-level rootkit detection,” *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 9, pp. 1465–1476, 2014.

[59] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 116–127.

[60] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection through vmm-based” out-of-the-box” semantic view reconstruction,” in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 128–138.

[61] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, “Drebin: Effective and explainable detection of android malware in your pocket.” in *Ndss*, vol. 14, 2014, pp. 23–26.

[62] R. Li, M. Du, D. Johnson, R. Ricci, J. Van der Merwe, and E. Eide, “Fluorescence: Detecting kernel-resident malware in clouds,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 367–382.

[63] M. Guri, Y. Poliak, B. Shapira, and Y. Elovici, “Joker: Trusted detection of kernel rootkits in android devices via jtag interface,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 65–73.

[64] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, “Hardware-based malware detection using low-level architectural features,” *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, 2016.

[65] A. Tang, S. Sethumadhavan, and S. J. Stolfo, “Unsupervised anomaly-based malware detection using hardware features,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 109–129.

[66] M. B. Bahador, M. Abadi, and A. Tajoddin, “Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition,” in *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2014, pp. 703–708.

[67] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, “Hardware performance counter-based malware identification and detection with adaptive compressive sensing,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–23, 2016.

[68] A. Garcia-Serrano, “Anomaly detection for malware identification using hardware performance counters,” *arXiv preprint arXiv:1508.07482*, 2015.

[69] H. Peng, J. Wei, and W. Guo, “Micro-architectural features for malware detection,” in *Conference on Advanced Computer Architecture*. Springer, 2016, pp. 48–60.

[70] V. Jyothi, X. Wang, S. K. Addepalli, and R. Karri, “Brain: Behavior based adaptive intrusion detection in networks: Using hardware performance counters to detect ddos attacks,” in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*. IEEE, 2016, pp. 587–588.

[71] F. Ceschin, F. Pinage, M. Castilho, D. Menotti, L. S. Oliveira, and A. Gregio, “The need for speed: An analysis of brazilian malware classifiers,” *IEEE Security Privacy*, vol. 16, no. 6, pp. 31–41, 2018.

[72] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, “Evading machine learning malware detection,” *black Hat*, 2017.

[73] K. Wang, J. J. Parekh, and S. J. Stolfo, “Anagram: A content anomaly detector resistant to mimicry attack,” in *International workshop on recent advances in intrusion detection*. Springer, 2006, pp. 226–248.

[74] B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu *et al.*, “Reviewer integration and performance measurement for malware detection,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 122–141.

[75] D. R. Beck and Y.-M. Wang, “Detecting user-mode rootkits,” Jan. 18 2011, uS Patent 7,874,001.

[76] Y.-M. Wang and D. Beck, “Fast user-mode rootkit scanner for the enterprise.” in *LISA*, 2005, pp. 23–30.

[77] L.-P. Yuan, W. Hu, T. Yu, P. Liu, and S. Zhu, “Towards large-scale hunting for android negative-day malware,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 533–545.

[78] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, “Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis,” in *Network and Distributed System Security Symposium*, 2020.

[79] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, “On the feasibility of online malware detection with performance counters,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 559–570, 2013.

[80] M. Kazdagli, L. Huang, V. Reddi, and M. Tiwari, “Morpheus: Benchmarking computational diversity in mobile malware,” in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014, pp. 1–8.

[81] N. Patel, A. Sasan, and H. Homayoun, “Analyzing hardware based malware detectors,” in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[82] X. Wang and R. Karri, “Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 485–498, 2015.

[83] C. Konstantinou, E. Chielle, and M. Maniatakos, “Phylax: Snapshot-based profiling of real-time embedded devices via jtag interface,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 869–872.

[84] S. Zonouz, J. Rrushi, and S. McLaughlin, “Detecting industrial control malware using automated plc code analytics,” *IEEE Security & Privacy*, vol. 12, no. 6, pp. 40–47, 2014.

Appendix A
Summary of Features

TABLE IX: A summary of microarchitectural features.

Type	Feature	Description
Microarchitectural	ERETURN	Exception Return Instructions
	DUNALIGNED	Unaligned Data Accesses
	NEONWORK	NEON and Integer Unit not idle
	UNALIGNEDREPLAY	Replay Events from Unaligned Access
	DCACCESSNEON	Data Cache accesses by NEON
	ETMEXTOUT1	ETM Signal ETMEXTOUT1
	L2STORE	L2 Cache Stores
	ITLBMISS	Instruction TLB Misses
	DCHITNEON	Data Cache Hits by NEON
	L2ACCESSNEON	L2 Cache accesses by NEON
	DTLBMISS	Data TLB Misses
	L2MERGE	L2 Cache Stores Merged
	DCHASHMISS	Data Cache Misses by Hash
	CONTEXT	Context Switch Instructions
	L2ACCESS	L2 Cache Accesses
	RETURN	Return Instructions
	ECALL	Exception Call Instructions
	DWRITE	Data Write Accesses
	DREAD	Data Read Accesses
	BPREDICTABLE	Predictable Branch Instructions
	RSTKMISS	Return Stack Misses
	L2MISS	L2 Cache Misses
	BINST	Immediate Branch Instructions
	L2HITNEON	L2 Cache Hit by NEON
	BPREDTAKEN	Branch Instructions predicted to be taken
	CLOCKCYCLES	Clockcycles
	BPEXECTAKEN	Predictable Branch Instructions taken
	DCACCESS	Data Cache Accesses
	IIDLE	Instruction Buffer Idles
	PCINST	PC Change Instructions
	REPLAY	Replay Events
	BPCONDMIS	Branch Instructions Condition Mispredicted
	AXIWRITE	AXI write active
	BPMIS	Branch Instructions Mispredicted or not Predicted
	DCALIAS	Data Cache page coloring aliases
	ICMISS	Instruction Cache Misses
	AXIREAD	AXI read active
	OPERATION	Operations Issued
	INST	Instructions
	IISSUE	Instructions Issued
	NEONSTALL	Stalls from NEON
	NEONWAITS	Waits for NEON access
	ICACCESS	Instruction Cache Accesses
	DCMISS	Data Cache Misses
	WBFULL	Write Buffer Full
	ICHASHMISS	Instruction Cache Misses by Hash
	SOFT	Software increment
	ETMEXTOUT0	ETM Signal ETMEXTOUT0
	ETMEXTOUT01	ETM Signal ETMEXTOUT0 + ETMEXTOUT1

TABLE X: A summary of semantic features.

Type	Feature	Description
Semantic	prio	Priority of a process used when scheduled
	vruntime	Runtime of a thread
	min_flt	Minor page faults
	wakee_flips	For task wake up
	maj_flt	Major page faults
	stime	System time
	pcount	Number of times a process ran on a CPU
	nvcsw	Context switch counts
	sum_exec_runtime	Total time spent on the CPU
	utime	Time spent in user mode
	syscr	Number of read syscalls
	prev_sum_exec_runtime	Previous execution time
	nr_failed_migrations_hot	Failed process migration statistics
	rchar	Number of bytes read
	wchar	Number of bytes written
	wakee_flip_decay_ts	Delay in process wake up
	acct_rss_mem1	Accumulated RSS usage
	priority	Real-time priority
	timeslice	Execution time slice
	acct_vm_mem1	Accumulated virtual memory usage
	block_max	Maximum block I/O size
	nivcsw	Context switch counts
	run_delay	Time spent waiting on a runqueue
	syscw	Number of write syscalls
	wakeups	Number of wake up
	wakeups_passive	Number of passive wake up
	nr_failed_migrations_running	Failed running process migration statistics
	usage	Reference count on the task structure of a process
	wakeups_idle	Number of idle wake ups
	iowait_count	I/O wait counter
	weight	Process weight for load balancing
	slice_max	Maximum time slice
	wait_max	Maximum wait time
	nr_forced_migrations	Number of forced migrations
	wakeups_local	Number of local wake ups
	wakeups_affine_attempts	Number of wake up affine attempts
	dl_bw	$\frac{dl\_runtime}{dl\_period}$
	sleep_start	Sleep start timestamp
	wait_count	Wait count
	wait_start	Waiting start timestamp
	wait_sum	Total wait time
	flags	Specifying scheduler behavior
	exec_max	Maximum execution time
	sum_sleep_runtime	Total sleep time during runtime
	iowait_sum	Total I/O wait time
	sleep_max	Maximum sleep time
	dl_density	$\frac{dl\_runtime}{dl\_deadline}$
	wakeups_affine	Total affine wake ups
	wakeups_remote	Total remote wake ups
	wakeups_migrate	Total migrate wake ups
	block_start	Block I/O start timestamp
	nr_failed_migrations_affine	Number of failed affine migrations
	nr_migrations_cold	Number of cold migrations
	wakeups_sync	Number of sync wake ups