Efficient and Trusted Detection of Rootkit in IoT Devices via Offline Profiling and Online Monitoring

Xingbin Jiang, Michele Lora, Sudipta Chattopadhyay
{xingbin_jiang,michele_lora,sudipta_chattopadhyay}@sutd.edu.sg
Singapore University of Technology and Design
Information Systems Technology and Design (ISTD)

ABSTRACT

We present LKRDet: a framework based on a Trusted Execution Environment to detect kernel rootkits in IoT devices. LKRDet checks the consistency of hardware events, occurring in specific system call routines, to detect abnormalities caused by the kernel rootkits. LKRDet relies on Hardware Performance Counters to efficiently and safely count the hardware events occurring in the system.

We implement a prototype of LKRDet for the ARM TrustZone architecture, on top of the Open Portable Trusted Execution Environment and evaluate our prototype with four popular rootkits. Our evaluation reveals that LKRDet can accurately detect the presence of all the rootkits in the device.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Tamper-proof and tamper-resistant designs**; **Malware and its mitigation**;

KEYWORDS

Rootkit detection, Internet-of-Things security, ARM Architectures

ACM Reference Format:
Xingbin Jiang, Michele Lora, Sudipta Chattopadhyay. 2020. Efficient and Trusted Detection of Rootkit in IoT Devices via Offline Profiling and Online Monitoring . In *Proceedings of the Great Lakes Symposium on VLSI 2020 (GLSVLSI '20), September 7–9, 2020, Virtual Event, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3386263.3406939

1 INTRODUCTION

The rapid development of Internet of Things (IoT) technologies and their introduction into critical applications, makes IoT devices inviting targets for malwares and attacks [12]. *Rootkits* are a particular category of malware that may provide authentication rights to illegitimate users, or hide traces of other attacks. *Kernel rootkits* are a special kind of stealthy rootkits able to gain the same privileges as the operating system's kernel [17]. They achieve their malicious purposes by hijacking Linux's system calls. They can alter kernel control-flow by using apparently legitimate routines. System utilities (*e.g.*, `ls`, `ps`, `netstat`) can be spoofed making the system unaware of the existence of malicious software. For instance, Adore-ng redirects the execution of system calls to its malicious code by replacing the function pointers in the virtual file systems [10]. The Suterusu rootkit redirects the execution flow to malicious handlers by inline hooking the function pointers during a specific system call routine [6]. A large amount of IoT devices now relies on a common operating system (*e.g.*, Linux), making easier migrating rootkits to target ARM-based devices [11, 24], and IoT devices [3, 13].

We present *LKRDet: a rootkit detection framework for IoT devices* exploiting hardware features of modern architectures, and running on a Trusted Execution Environment (TEE). It exploits the information provided by the Hardware Performance Counters (HPCs) available in modern processors to check the kernel's control-flow consistency during a system call. The approach profiles the kernel space during the execution of system calls in a secured environment to gather data about each system call execution. The information is used to monitor the system after being deployed to detect rootkits. LKRDet is highly secure as its detection part is implemented in a TEE, thanks to the maturity reached by TEEs for IoT devices.

Non-deterministic approaches based on machine learning techniques are prominent in the state-of-the-art of rootkit detection [20, 26]. However, these techniques cannot reach complete accuracy. Furthermore, being them based on classification algorithms they must be trained by using both benign and malicious traces. These problems are mitigated by deterministic approaches monitoring hardware events [23]. However, this requires resources that are usually not available in IoT devices. We aim at reaching the accuracy of deterministic methods while using the resources provided by today's IoT devices. Moreover, the presented approach relies only on traces generated by non-infected systems, making it more effective for detecting still unknown rootkits. Moreover, LKRDet is the first kernel rootkit detection mechanism explicitly thought for ARM architectures and IoT devices in general. Since existing rootkit detection solutions cannot be directly ported for IoT devices, we show how to design and implement a light-weight rootkit detection framework for low-end IoT devices relying on a TEE.

2 BACKGROUND AND RELATED WORK

Rkhunter [2] and Kstat [1] are two fundamental in-the-box software-based detection approaches. They validate the kernel control-flow by comparing the kernel text or its hash to the contents of critical jump tables in a previously observed clean state. However, they may be spoofed by advanced rootkits because kernel rootkits have the same level of authority as the Linux kernel. Virtual Machine

This work is supported by the grant number *MOE2018-T2-01-098* from the Singapore Ministry of Education.

Monitor (VMM)-based out-of-the-box rootkit detection methods are isolated from the target environment target, and they monitor the static or dynamic kernel objects of a guest Virtual Machine (VM) at the VMM level. However, the semantic gap between the external and internal observation makes complicated acquiring the semantics of the tasks running in the guest VM [5]. Also, attacks in the guest operating system may modify the layout of the guest kernel data structures, compromising the detection.

Hardware events (*e.g.*, *instructions* and *branches*) occurring while executing a kernel routine can be exploited to detect abnormalities in the kernel control-flow. HPCs provided by modern processors allow counting such events. HPC-based detection methods can be divided into two categories [26]: *data-centric* approaches checking the kernel work-flow integrity by using statistical techniques; *program-centric* approaches rely on machine learning methods to analyze the dynamic features of a specific workload.

A recognized data-centric solution [23] verifies the kernel Control-Flow Integrity (CFI) by counting the CPU hardware features during the execution of a system call. This solution is implemented in a VMM environment for enhancing security, and it has proved its effectiveness when detecting a variety of kernel rootkits. However, it increases design complexity and causes a waste of resources. As such, the solution is not suitable for IoT devices that are usually constrained by reduced resources. Furthermore, the approach suffers the possibility of the VMM itself being targeted by the attack [16].

Program-centric techniques use HPCs to model the dynamic behavior of a specific workload, identifying occurrences. Kernel rootkits can be detected by exploiting various machine learning methods [9, 19]. Most machine learning methods reaches more than 80% accuracy (true-positive). However, 100% accurate detection of the kernel rootkit cannot be achieved [20].

Most of the previous approaches target x86-based systems, whereas IoT devices usually adopt resource-constrained processors, often based on ARM architectures. Only [22] dealt with resource constrained embedded systems by embedding an additional software module in the system bootloader. Such a module is stored in the read-only memory of the device to not being tempered. As such, it is more invasive with respect to the approach we propose.

3 THE LKRDET DETECTION FRAMEWORK

LKRDet monitors the control-flow of multiple system calls while monitoring the HPCs. The HPC values must be comparable in different operating states and time points in the life cycle of the system. Therefore, a *test application* invokes a set of system calls with constant inputs. Each system call impacts differently on the monitored events. Thus, some event may be more effective than others for detecting rootkits, as discussed in sections 4.1.

A kernel rootkit may tamper HPCs values to hide its effects. Detection mechanisms for general-purpose computing systems mitigating such issue by relying on virtualization technologies and out-of-the-box approaches. However, virtualization solutions for IoT devices are still in their infancy. Although they can achieve isolation between VM and VMM to improve security, the security issues of VMM itself still exist as they do not provide standard protection mechanisms [14, 18]. Furthermore, VMM-based solutions require an amount of resources usually unavailable in IoT devices. These limitations make VMM-based solution unviable for IoT devices. For these reasons, LKRDet relies on a TEE, rather than a VMM, to implement the application monitoring the HPCs.

TEE is a technology that guarantees confidentiality and integrity of data and code running in a processor [15]. It is designed to be isolated from the traditional system, called Rich Execution Environment (REE), establishing a secure and trusted execution area in the processor. Several TEE solutions are available in modern processors, ARM TrustZone [15] is considered one of the most promising technology for implementing TEE in mobile and IoT devices. TrustZone is provided by ARM architectures, ranging from the Cortex-A family of CPUs for high-end devices, to the Cortex-M family of micro-controllers intended to be used in low cost IoT systems. A valid open-source TEE software solution is Open Portable Trusted Execution Environment (OP-TEE) [4]. It provides all the components needed to implement a completely secure operating environment. It includes a secure privileged layer, a set of secure user-space libraries, a REE based on the Linux kernel and drivers, a Linux user-space library, a Linux userspace supplicant daemon, *etc.* Moreover, OP-TEE supports many different hardware platforms.

LKRDet targets kernel rootkits which have the same privileges as the Linux kernel. Thus, rootkits able to modify the control-flow of the kernel through its read and write permissions in the kernel space. We assume that the REE is not trustworthy, while the TEE is trustworthy, that is a reasonable assumption due to TEEs nature [15]. As such, the rootkit can only compromise the REE but it is isolated from the TEE. Indeed, attacks able to compromise also the TEE have been proposed [7]. This is an orthogonal issue with respect to the addressed problem, that is out of scope of the paper.

LKRDet detects kernel rootkits by verifying the consistency of each system call execution process, and it is structured into two steps: the *offline profiling* step retrieves the information about the system calls used by a *test application* running on a clean system; the *online monitoring* step monitors the execution of the test application and compares the values of the HPCs with the value stored at the profiling step to determine whether there are abnormalities in the system behavior. Whenever the values of the HPCs observed during the monitoring phase differ from the values observed and stored during the profiling phase, then different hardware primitives have been used, and the system has been compromised. Recently, some studies showed that HPCs values may be not trustworthy in certain circumstances [25]. [8] identifies a set of guidelines to properly use HPCs for security applications: LKRDet follows all the suggested guidelines: it profiles small portions of software (*i.e.*, individual system calls), it considers the noise induced by other processes, and it manages context switches that may eventually occurs.

We present our TEE-based implementation, which addresses the challenges due to the scarce resources available in IoT devices. Communication between the TEE and operating system must allow efficient monitoring of the profiled primitives, while data storage must be efficient due to the limited memory available in IoT devices.

3.1 TEE–based implementation

During the offline profiling, a Client Application (CA) and the *test application* execute in an uncontaminated REE environment running a clean Linux operating system installation. Meanwhile, a

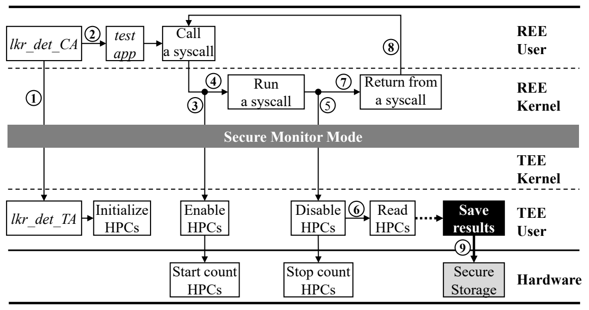

Figure 1: Execution flow of the offline profiling step.

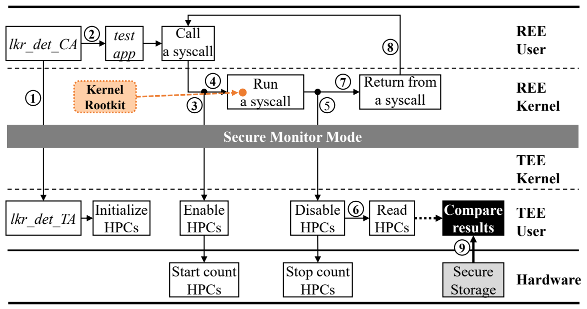

Figure 2: Execution flow of the online monitoring step.

Trusted Application (TA) is being executed in an isolated TEE environment based on OP-TEE. The *test application* invokes the different kinds of monitored system calls with constant inputs. The role of the CA is twofold: it intercepts the monitored system calls at their entry and return points, and it transfers the gathered information to the TA. The TA is guided by the commands and information received from the CA. The TA initializes the HPCs, starts and stops counting saving the final results to the secure storage. Once the HPCs are enabled, they automatically count the monitored hardware events in the CPU without needing any software intervention.

In multicore systems a system call's execution moves among multiple cores, possibly making the HPCs inconsistent. To solve this issue [23], whenever a system call execution "moves" from a core to another, LKRDet stops the HPCs of the first from counting the events, and starts the ones of the second. LKRDet keeps the sum obtained by the values in all cores. Moreover, the *test* invokes each system call multiple times to mitigate the effects of the noise due to the operating systems running in the REE. The TA will consider the average values of the HPCs as the final results to be stored.

Figure 1 illustrates the complete flow of the offline profiling phase. The arrows and circled numbers outline the order of the operations. The CA (*i.e.*, `lkr_det_ca`) starts executing in the REE. It sends to the TA (*i.e.*, `lkr_det_ta`) the command to initialize the HPCs (step ①), and the TA initializes the HPCs setting which hardware events must be counted. Then, it sets the relevant configuration registers in the Performance Monitoring Unit (PMU). In our case, six HPCs are used to record the six types of hardware events being monitored simultaneously. The HPCs are configured to log only the events happening in the REE kernel (*i.e.*, the Linux kernel), thus avoiding the noise generated by other user processes or switching tasks. Then, the runtime switches back to REE continuing the execution of the CA. The API interacts with the kernel drivers in the REE and the TEE to switch to the Monitor Mode: a secure state that provides to the CPU the ability to switch between REE and TEE.

The CA executes the *test application* as a child process (step ②). Thus, the CA can use the Linux `ptrace()` function to trace and control the *test application*. Whenever the *test application* starts calling a monitored system call, it will be intercepted by the CA at the entry point (step ③). The execution switches from the CA to the TA, and the HPCs start counting the hardware events. This also requires to configure the relevant registers in the PMU. The profiling targets the kernel space and is executed for each system call, rather than for the entire program. This allows reducing as much as possible the noise due to other operations that can be executed by the CPU running the operating system on the REE. The intercepted system call restores immediately after, the runtime switches back to the REE and starts executing in the REE kernel (step ④). The HPCs will automatically count the hardware events generated in the CPU during the control-flow of the system call. Similarly, the system call is intercepted at its return point by the CA (step ⑤). The runtime switches to the TEE and the TA disables the HPCs to stop counting the hardware events. The values stored in the HPCs are read and recorded as reference values for the executed system call (step ⑥). The *test application* returns from the system call routine after the runtime switches back to the REE (step ⑦).

During the lifetime of the *test application*, the loop between steps ③ to ⑧ is run multiple times to obtain the HPCs values of all the types of monitored system call. Each system call is invoked multiple times, and the average values of the HPCs will be stored. This is necessary to eliminate random noise and enhance accuracy. Finally, the TA saves the results to secure storage (step ⑨). The values are stored as a two-dimensional array having as many rows as the monitored system calls, and as many columns as the traced events.

Figure 2 illustrates the work flow of the online monitoring phase. The CA and the TA work almost the same as in the offline profiling phase. During the monitoring phase, the TA gather the the HPCs values from the CPU executing the system call. It compares the HPCs values with those retrieved during the profiling (step ⑨ of Figure 2). The *compare results* step calculates the deviation between each *HPC event* value monitored and the one in the secure storage. The monitored value is considered to be in a normal range if its deviation is within a predefined threshold. A larger deviation is an evidence of the existence of a kernel rootkit.

4 EXPERIMENTAL RESULTS

We implemented the framework[1] to protect a device based on an ARM Cortex-A53 CPU, running Linux (kernel version 4.6.3) and OP-TEE (version 3.2.0). We tested the detection system against four kernel rootkits (*i.e.*, *Adore-ng*, *Diamorphine*, *Kbeast*, and *Suterusu*) both injected individually and collectively in the system, and by checking the consistency of five system calls, *i.e.*, *sys_getdents64*, *sys_read*, *sys_write*, *sys_openat*, and *sys_close*. Six kinds of HPC events are counted simultaneously for the execution of each system call: *Instructions (IN)*, *Cache References (CR)*, *Cache Misses (CM)*, *Branch Instructions (BI)*, *Branch Misses (BM)*, and *Bus Cycles (BC)*.

[1] The experiments' code is available at https://gitlab.com/asset-sutd/public/lkrdet

Table 1: Counts of HPC events for each system call in different kernel modes. For each monitored event, it reports the value (Val) stored during the monitoring phase, and the deviation (Dev), in percentage, with respect the reference values.

System call	Mode	HPC event												Status
		IN		CR		CM		BI		BM		BC		
		Val	Dev	Val	Dev	Val	Dev	Value	Dev	Val	Dev	Val	Dev	
getdents64	*Reference*	*13220*	-	*5184*	-	*87*	-	*1452*	-	*241*	-	*10918*	-	-
	Clean	13205	0	5174	0	81	-6	1449	0	243	0	10892	0	Normal
	Adore-ng	14748	**11**	5783	**11**	109	**25**	1626	**11**	283	**17**	12421	**13**	**Abnornal**
	Diamorphine	13764	4	5370	3	86	-1	1529	**5**	281	**16**	11630	**6**	**Abnornal**
	Kbeast	14036	**6**	5411	4	85	-2	1580	**8**	284	**17**	11788	**7**	**Abnornal**
	Suterusu	13796	4	5415	4	100	**15**	1524	4	272	**12**	11739	**7**	**Abnornal**
	Mixed	15884	**20**	6071	**17**	127	**45**	1785	**22**	345	**43**	13716	**25**	**Abnornal**
read	*Reference*	*6762*	-	*3149*	-	*68*	-	*866*	-	*112*	-	*6678*	-	-
	Clean	6803	0	3167	0	66	-2	872	0	112	0	6759	1	Normal
	Adore-ng	6789	0	3160	0	70	2	870	0	114	1	6717	0	Normal
	Diamorphine	6835	1	3182	1	61	-10	876	1	115	2	6781	1	Normal
	Kbeast	8500	**25**	3447	**9**	81	**19**	1139	**31**	168	**50**	8017	**20**	**Abnornal**
	Suterusu	6811	0	3169	0	68	0	873	0	115	2	6755	1	Normal
	Mixed	8437	**24**	3418	**8**	80	**17**	1130	**30**	170	**51**	7983	**19**	**Abnornal**
write	*Reference*	*8779*	-	*4004*	-	*132*	-	*1083*	-	*183*	-	*9627*	-	-
	Clean	8811	0	4017	0	130	-1	1086	0	185	1	9694	0	Normal
	Adore-ng	8850	0	4035	0	139	5	1092	0	188	2	9736	1	Normal
	Diamorphine	8855	0	4037	0	131	0	1092	0	190	3	9759	1	Normal
	Kbeast	10251	**16**	4247	**6**	152	**15**	1317	**21**	231	**26**	10744	**11**	**Abnornal**
	Suterusu	8828	0	4023	0	146	10	1088	0	191	4	9773	1	Normal
	Mixed	10278	**17**	4260	**6**	154	**16**	1323	**22**	233	**27**	10809	**12**	**Abnornal**
openat	*Reference*	*9707*	-	*4445*	-	*109*	-	*1134*	-	*191*	-	*9937*	-	-
	Clean	9728	0	4454	0	107	-1	1136	0	193	1	9975	0	Normal
	Adore-ng	9718	0	4447	0	111	1	1134	0	195	2	9948	0	Normal
	Diamorphine	9704	0	4441	0	106	-2	1133	0	191	0	9965	0	Normal
	Kbeast	9755	0	4463	0	113	3	1138	0	194	1	10012	0	Normal
	Suterusu	9731	0	4456	0	124	13	1137	0	199	4	10074	1	Normal
	Mixed	9731	0	4454	0	118	8	1135	0	198	3	10034	0	Normal
close	*Reference*	*6941*	-	*3240*	-	*71*	-	*898*	-	*104*	-	*6902*	-	-
	Clean	6968	0	3252	0	70	-1	902	0	105	0	6950	0	Normal
	Adore-ng	6958	0	3248	0	72	1	900	0	105	0	6924	0	Normal
	Diamorphine	6936	0	3235	0	68	-4	897	0	103	0	6925	0	Normal
	Kbeast	6892	0	3222	0	67	-5	892	0	107	2	6840	0	Normal
	Suterusu	6943	0	3241	0	75	5	898	0	103	0	6928	0	Normal
	Mixed	6911	0	3230	0	70	-1	894	0	106	1	6897	0	Normal

Offline profiling is first performed in a clean kernel state. Online monitoring is performed in six kernel states: clean state, and compromised by different rootkits. The test application invokes each system call 2000 times to mitigate the noise introduced by the Linux Kernel, obtaining accurate thresholds to identify abnormalities.

Among the HPC events that we monitored, *Cache Misses (CM)* has the minimum counting value and maximum deviation. The observed noise for CM is less than 15% for all uninfected system calls; thus, 15% is an appropriate threshold to detect abnormalities in CM. *Branch Misses (BM)* also has a small counting value, and its observed noise is 10%. So a deviation greater than 10% in BM suggests a malicious modification. The noise for the other four HPC events, *i.e.*, Instructions (IN), Cache References (CR), Branch Instructions (BI), and Bus Cycles (BC), are all less than 5%. Therefore, the deviation thresholds for IN, CR, BI, and BC are set to 5%.

Table 1 reports the results. The first column lists the system calls being monitored; the second column lists one offline-reference mode the six online-test modes (*i.e.*, clean kernel, the four infected kernels by single rootkit, and the *Mixed* line referring to a kernel infected by all the rootkits). The next columns show, for each of the six HPC events, both the absolute values and their deviation from the reference value computed during the offline profiling. The last column reports whether the deviation identifies as normal (*i.e.*, no rootkit detected) or an abnormal (*i.e.*, a rootkit is detected) status.

Figure 3 depicts the distribution of branch misses during the different executions considered in our experiments. It allows noticing a factora useful at balancing sensitivity and accuracy when selecting the thresholds for each specific HPC event. The count values

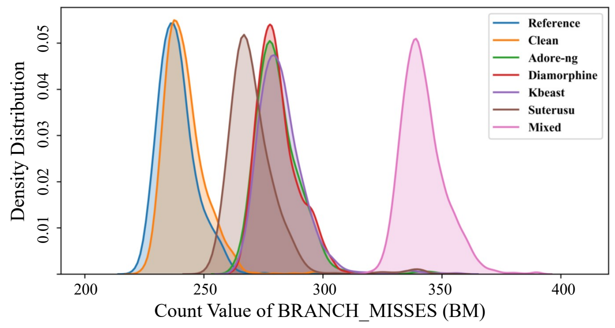

Figure 3: Density distribution of Branch Misses.

for the clean kernel state in Offline (Reference) and Online (Clean) modes are not exactly the same due to system noise: the value of the discrepancy is a good criterion for choosing the threshold.

Then, a second factor to be considered when choosing the thresholds is that larger absolute values are less affected by the system noise. Therefore, a smaller threshold is applicable for IN, CR, BI, and BC, but a larger threshold is needed for CM and BM.

The bold numbers in Table 1 indicate that the deviations exceed the set thresholds (15% for CM, 10% for BM, 5% for IN, CR, BI, and BC). In other words, the execution of these system calls (*sys_getdents64* for all the four rootkits and *sys_read*, *sys_write* for Kbeast only) allows identifying an abnormal state. This result is consistent with the working principle of the four rootkits. Adore-ng, Diamorphine and Suterusu just modify *sys_getdents64* for hiding malicious information, but Kbeast also modifies *sys_read* and *sys_write*. Finally, the kernel infected by multiple rootkits (*i.e.*,

Table 2: Influence of selected HPC events on the detection accuracy: *Y* indicates that the rootkit (rows) has been identified by monitoring the event (columns).

	IN	CR	CM	BI	BM	BC
Adore-ng	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*
Diamorphine			*Y*	*Y*	*Y*	*Y*
Kbeast	*Y*			*Y*	*Y*	*Y*
Suterusu			*Y*		*Y*	*Y*
Mixed	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*

Mixed) presents the largest deviations with respect to the stored values. Therefore, it is reasonable saying that LKRDet can accurately distinguish whether a rootkit infects the kernel.

4.1 HPCs events selection and accuracy

The sets of available HPC registers and hardware events vary among different processor. Some processors might have constrained number of HPC registers (*e.g.*, ARM11 only has two HPC registers). As such, we cannot monitor too many types of HPC events simultaneously. Meanwhile, not all kinds of hardware events are sensitive to the kernel control-flow modifications. Therefore, it is useful to select HPC events properly for improving detection effectiveness.

Table 2 shows which kinds of HPC events detected the rootkits accurately. We can indicate that the values of BM and BC are more susceptible to the kernel control-flow modifications. Thus, they provide more useful information when aiming at detecting rootkits that impacts on the kernel control-flow. CM and BI are less sensitive, IN and CR are worst for detecting the considered rootkits.

4.2 Accuracy

As discussed in section 2, there are mainly two kinds of HPC-based kernel rootkits detection methods. *Program-centric Machine Learning (ML)-based* methods classifying the dynamic behavior of specific workload patterns [9, 19], and *data-centric* solutions counting the HPC values during a system call execution of which the most representative approach is *NumChecker* [23]. We compare our detection framework against these state-of-the-art approaches.

For the sake of correctness, we implemented a ML-based mechanism in the same device used for the previous set of experiments. The features for training the ML models are captured by sampling the HPC events every 10 millisecond during the execution of a fixed input `ls` command. Then, we sample the same six HPC events as in our TEE-based LKRDet mechanism. Since `ls` command will call many system calls, including *sys_getdents64*, the malicious modification in the system call control-flow should also cause different features shown in the timeline characteristics. Figure 4 shows the samplings of branch misses during the execution of the `ls` command in different kernel modes (*i.e.*, lean kernel, the four infected kernels by different rootkits individually, and the Mixed line referring at the kernel infected by all the rootkits). Comparing the sampling values at points ①, ②, and ③, the existence of these rootkits is visible thanks to their impact on the number of branch misses. In particular, the rootkits Adore-ng and Kbeast highly modify the kernel control-flow thus they cause large deviation of the timeline HPC features, while Diamorphine and Suterusu show milder impacts. Finally, the trace generated kernel infected by multiple rootkits (*i.e.*, Mixed) is visibly different by any other trace. This is consistent with the deviations observed in Table 1.

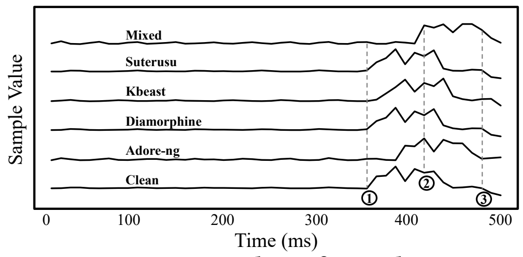

Figure 4: HPC sampling of Branch Misses.

Table 3: Detection accuracy (true positive/false positive) of the different mechanisms considered in this work.

	Adore-ng	Diamorphine	Kbeast	Suterusu	Mixed
NumChecker	1/0	-	1/0	1/0	-
ML-Based	1/0	0.67/0.15	1/0.08	0.76/0.18	1/0
LKRDet	1/0	1/0	1/0	1/0	1/0

Table 4: Time overhead required for each method.

	CPU	Checking Time
NumChecker	AMD Opteron 1356/2.3GHz	262.3 ms
ML-Based	Broadcom BCM2837/1.2GHz	825 s
LKRDet	Broadcom BCM2837/1.2GHz	2.91 s

We implement an online ML-based rootkit detection framework and trained six kinds of classification methods (*i.e.*, k-Nearest Neighbors (k-NN), Logistic Regression, Naive Bayes, Support Vector Machines (SVM), Decision Tree, and Random Forest), with each timeline feature of six HPC events. We collected 15000 entries for each HPC event in different kernel states. The dataset for training includes 90000 data for each HPC event. The online testing of the ML-based classification approach is conducted by testing 1000 data for each HPC event and calculating the probability that the rootkit exists. It is important noticing that the ML-based approaches required to be trained with both malicious and benign traces. On the contrary, LKRDet requires to be fed only benign traces, generated by non-violated system executions. As such, it is more suitable than ML-based approaches to identify previously unknown rootkits.

Table 3 reports data about the detection accuracy of different rootkit detection mechanisms. The results of NumChecker are collected from the referenced articles [23], as it is not suitable for running on IoT devices. The ML-based results refers to our implementation of such methods; for each rootkit, the table reports the highest accuracy and lowest false positive achieved using different classification methods: *Random Forest over IN for Adore-ng*, *k-NN over BM for Diamorphine* and *BS for Kbeast*, *Decision Tree over IN for Suterusu*, and *Random Forest over IN for Mixed mode.*

Both ML-based approaches and LKRDet reaches full accuracy for kernel infected by rootkits that strongly modify the kernel control-flow (*i.e.*, Adore-ng, Kbeast) and by multiple rootkits (*i.e.*, Mixed column): a configuration that has not been tested in [23]. Considering Suterusu, which causes slight modifications, the accuracy of ML-based mechanism decrease to 0.76, and the false-positive raise to 0.18. On the other hand, LKRDet and NumChecker preserve complete accuracy. LKRDet can still show 100% detection accuracy for the rootkit Diamorphine, that has not been considered by Numchecker, while the ML-based method has the worst accuracy.

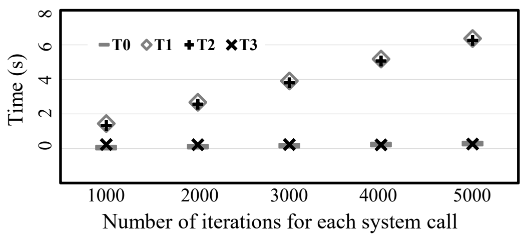

Figure 5: Components of the LKRDet overhead.

4.3 Efficiency and scalability

Table 4 compares the efficiency of LKRDet against the state-of-the-art approaches by considering the time required for the online checking. NumChecker [23] performs one test in 262.3 milliseconds, performing 500 iterations of the test application, each iteration includes 5 system calls and monitors 3 HPC events. However, it cannot be applied to IoT architectures due to lack of resources. Our LKRDet approach takes 2.91 seconds to run the test application, each iteration including 2000 iterations of the 5 system calls and monitoring 6 HPC events. The ML-based method requires 825 seconds for capturing and testing 1000 feature data, with each feature data including 6 HPC events. Furthermore, any ML-based method will require time for training for each rootkit it aims at identifying.

To analyze the scalability of LKRDet, its time consumption can be split into four parts: (1) time to run the test application without profiling T0, (2) offline profiling time T1, (3) online monitoring time T2, (4) time to compare T3. Figure 5 shows the values of these values varying the number of system calls in the test application from 1000 to 5000. T0 increases slightly with the number of iterations but remains negligible (less than 0.3s). T1 and T2 have very close values and increase reasonably. T3 remains identical when varying the number of iterations. Increasing the number of system calls reduces the system noise improving the precision, meanwhile only T1 and T2 increases. Since T1 is executed only once offline, only T2 impacts the detection overhead during the life of the system. Thus, LKRDet overhead scale well when increasing the precision.

5 DISCUSSION AND CONCLUDING REMARKS

Among the HPC-based detection methods, ML-based solutions can continuously improve their effectiveness by adjusting their models and obtaining more data. However, they struggle to deal with new rootkits having unknown features. This is because ML-based methods require to be trained by using traces produced by both secured and infected systems. Otherwise, most of ML algorithms for classification will not be able to recognize malicious executions.

On the other hand, deterministic solutions provide complete accuracy, but they require complex architectures, providing virtualization features usually unavailable to IoT devices. Also, the VMM itself can be the attacking target to damage detection mechanisms.

We presented LKRDet, a TEE-based kernel rootkit detection framework relying on HPCs to monitor the execution flow of system calls. LKRDet is secured by running in a trusted execution environment, and lightweight enough to be used in any ARM-based IoT devices. It is not prone to application updates and varying system load, as the test application remains the same, and the profile data remains almost unchanged due to the kernel space profiling. As such, it requires to execute the profiling phase only once for each kernel update and to redesign the test application only in case of significant changes in the kernel. Meanwhile, the monitoring phase may be executed periodically during the life of the system, as a low-priority task by impacting only minimally on eventual real-time constraints of the system. Furthermore, by monitoring the smallest piece of software to be monitored, it overcomes the drawbacks recently highlighted [8, 25] in the use of HPCs.

As a major limitation, LKRDet cannot detect kernel rootkits tampering with the HPC values [21]. However, this limitation is common to any methods relying on HPCs to detect kernel rootkits.

REFERENCES

[1] 2003. Kernel Security Therapy Anti-Trolls. (2003). http://freshmeat.sourceforge.net/projects/kstat [accessed 13-May-2019].
[2] 2018. The Rootkit Hunter project. (2018). http://rkhunter.sourceforge.net/
[3] 2019. McAfee Labs 2019 Threats Predictions Report. (2019). https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/mcafee-labs-2019-threats-predictions/ [Online; accessed 8-June-2019].
[4] 2019. Open Portable Trusted Execution Environment. (2019). https://www.op-tee.org/ [Online; accessed 13-May-2019].
[5] Sina Bahram et al. 2010. Dksm: Subverting virtual machine introspection for fun and profit. In *2010 29th IEEE symposium on reliable distributed systems*. 82–91.
[6] Robert Buhren et al. 2016. The threat of virtualization: Hypervisor-based rootkits on the ARM architecture. In *Proc. of ICICS*. 376–391.
[7] Haehyun Cho et al. 2018. Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone. In *Proc. of the 34th ACSAC*.
[8] Sanjeev Das et al. 2019. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proc. of IEEE S&P 2019*.
[9] John Demme et al. 2013. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, Vol. 41. 559–570.
[10] Jun Gu et al. 2016. A Linux rootkit improvement based on inline hook. (2016).
[11] Mordechai Guri et al. 2015. JoKER: Trusted detection of kernel rootkits in android devices via JTAG interface. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. 65–73.
[12] Xingbin Jiang et al. 2020. An Experimental Analysis of Security Vulnerabilities in Industrial IoT Devices. *ACM Trans. Internet Technol.* 20, 2, Article 16 (2020).
[13] Arun Kanuparthi et al. 2013. Hardware and embedded security in the context of internet of things. In *Proc. of ACM CyCAR*. 61–64.
[14] M. Mounika and C. Chinnaswamy. 2016. A comprehensive review on embedded hypervisors. *computing* 5, 5 (2016).
[15] Bernard Ngabonziza et al. 2016. Trustzone explained: Architectural features and use cases. In *Proc. of IEEE CIC*. 445–451.
[16] Diego Perez-Botero et al. 2013. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proc. of ACM SCC*. 3–10.
[17] Ryan Riley et al. 2009. Multi-aspect profiling of kernel rootkit behavior. In *Proc. of ACM EuroSys*. 47–60.
[18] Kristian Sandström et al. 2013. Virtualization technologies in embedded real-time systems. In *Proc. of IEEE ETFA*. IEEE, 1–8.
[19] Hossein Sayadi et al. 2018. Customized machine learning-based hardware-assisted malware detection in embedded devices. In *Proc. of TrustCom/BigDataSE*. 1685–1688.
[20] Baljit Singh et al. 2017. On the detection of kernel-level rootkits using hardware performance counters. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 483–493.
[21] Matt Spisak. 2016. Hardware-Assisted Rootkits: Abusing Performance Counters on the ARM and x86 Architectures. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*.
[22] Xueyang Wang et al. 2015. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *Proc. of IEEE/ACM ICCAD*. 544–551.
[23] Xueyang Wang and Ramesh Karri. 2015. Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 3 (2015), 485–498.
[24] Xianyi Zheng et al. 2016. TZ-KPM: Kernel Protection Mechanism on Embedded Devices on Hardware-Assisted Isolated Environment. In *Proc. of HPCC/SmartCity/DSS*. 663–670.
[25] Boyou Zhou et al. 2018. Hardware performance counters can detect malware: Myth or fact?. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 457–468.
[26] Liwei Zhou and Yiorgos Makris. 2018. Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution. In *Proc. of the IEEE/ACM DATE*. 1580–1585.