Design Automation for Embedded Systems
https://doi.org/10.1007/s10617-024-09283-1

Profiling with trust: system monitoring from trusted execution environments

Christian Eichler[1] · Jonas Röckl[2] · Benedikt Jung[3] · Ralph Schlenk[3] · Tilo Müller[4] · Timo Hönig[1]

Received: 14 July 2023 / Accepted: 24 January 2024

Abstract
Large-scale attacks on IoT and edge computing devices pose a significant threat. As a prominent example, Mirai is an IoT botnet with 600,000 infected devices around the globe, capable of conducting effective and targeted DDoS attacks on (critical) infrastructure. Driven by the substantial impacts of attacks, manufacturers and system integrators propose Trusted Execution Environments (TEEs) that have gained significant importance recently. TEEs offer an execution environment to run small portions of code isolated from the rest of the system, even if the operating system is compromised. In this publication, we examine TEEs in the context of system monitoring and introduce the Trusted Monitor (TM), a novel anomaly detection system that runs within a TEE. The TM continuously profiles the system using hardware performance counters and utilizes an application-specific machine-learning model for anomaly detection. In our evaluation, we demonstrate that the TM accurately classifies 86% of 183 tested workloads, with an overhead of less than 2%. Notably, we show that a real-world kernel-level rootkit has observable effects on performance counters, allowing the

This manuscript is an extended version of the conference paper that appeared in https://doi.org/10.1109/SBESC56799.2022.9964869.

✉ Christian Eichler
christian.eichler@rub.de

✉ Jonas Röckl
jonas.roeckl@fau.de

Ralph Schlenk
ralph.schlenk@nokia.com

Tilo Müller
tilo.mueller@hof-university.de

Timo Hönig
timo.hoenig@rub.de

1 Ruhr University Bochum, Bochum, Germany

2 FAU Erlangen-Nürnberg, Erlangen, Germany

3 Nokia Solutions and Networks GmbH & Co. KG, Nürnberg, Germany

4 Hof University of Applied Sciences, Hof, Germany

Published online: 16 February 2024

TM to detect it. Major parts of the TM are implemented in the Rust programming language, eliminating common security-critical programming errors.

Keywords Trusted execution environment · Hardware performance counter · Machine learning · Anomaly detection · Malware detection · Rust

1 Introduction

Recent forecasts estimate the number of 3.6 Internet of Things (IoT) devices per capita by 2023 [1]. Moreover, by the end of 2030, a total number of 29.4 billion connected IoT devices is expected [2]. At the same time, researchers observe an increasing threat from vulnerabilities and malware [3]. Despite steady advances in security research and formal verification, it is still unrealistic to build large software systems that are completely secure. Attackers are still able to change the state of the running system by exploiting vulnerabilities. For example, a privilege escalation vulnerability in the `sudo` binary has been unreported for more than ten years [4]. Additionally, the Mirai [5] and the Hajime [6] botnets have hijacked a vast amount of IoT devices.

Since it is unlikely for complex systems to parry every attack, the question arises if tampering with the system state can be detected. A promising instrument for detecting state anomalies is system monitoring. Once an anomaly is detected, it is possible to react accordingly. However, system monitoring can only be beneficial as long as an adversary is not able to disable or subvert the monitoring techniques. As a matter of principle, a monitoring tool at the level of the operating system is susceptible to kernel-level attackers. This is because the adversary is always able to disable the monitoring techniques (e.g., by removing the corresponding code from the running operating system).

As a countermeasure, several hypervisor-based monitoring systems have been proposed [7–9]. By leveraging Virtual Machine Introspection (VMI) [10], it becomes possible to retrieve insights about the dynamic state of the virtual machine that potentially uncovers intruders. In contrast to a host-based system, a hypervisor-based monitoring system has a smaller Trusted Computing Base (TCB) [11] and can withstand a kernel-level attacker by design.

In recent times, TEEs have seen widespread use in modern System on Chips (SoCs). The ARM TrustZone, one of the most prevalent TEE implementations, introduces hardware-based isolation between two system partitions, the normal world and the secure world. The normal world hosts a feature-rich Operating System (OS), whereas the secure world runs a stripped-down special-purpose OS focusing on security-related tasks. The secure OS runs Trusted Application (TAs). Even if an attacker gains the highest privilege in the normal world, the secure world is still not compromised. This design can provide protection from strong system-wide adversaries: Even a kernel-level or hypervisor-level attacker is not able to influence the code running in the secure world arbitrarily, and communication is only possible via well-defined, lightweight interfaces.

The strong isolation properties of modern TEEs enable TEE-based system monitoring while keeping the TCB small. In this paper, we present the Trusted Monitor (TM), a novel approach for intrusion detection based on low-level system monitoring and artificial intelligence. From within the ARM TrustZone, we continuously monitor the system's state by relying on hardware performance counters. The performance metrics are utilized to infer the running process. We show that different applications have different footprints with respect

to their performance metrics and demonstrate that the insertion of a kernel rootkit changes the footprint of the running application.

Problem statement. This paper makes the case that high-end IoT devices are widespread and connected to the internet but still lack the level of security needed for reliable online operation. Existing IoT devices are insecure by (software) design (e.g., by using programming languages susceptible to buffer overflows), and also miss the ability to detect a successful attack on the system. The use of insecure languages, such as C, causes a variety of vulnerabilities (e.g., Ripple20 [12], AMNESIA:33 [13]) that cannot be avoided systematically without a rewrite of the whole system.

Our TM tackles this challenge by providing a novel intrusion detection system that is suitable for existing applications and will be presented in the following.

Contributions In summary, this paper makes the following contributions:

(1) To the best of our knowledge, we are the first to evaluate intrusion and anomaly detection based on hardware performance counters and artificial intelligence directly out of the ARM TrustZone TEE.
(2) We present a functional prototype as an extension to the Linux operating system. Major parts of the TM are implemented in the Rust programming language, eliminating common security-critical programming errors.
(3) We evaluate the monitoring capabilities based on hardware performance counters and show that the monitor correctly classifies 86% of the 183 evaluated workloads.
(4) We show the feasibility of our approach by analyzing the performance penalty in practice. We measure an overhead below 2% with an interval of one second or more between the system checks.
(5) We demonstrate that our trained model detects deviations in the hardware performance counters when a real-world kernel-level rootkit is activated.

2 Related work

There are several streams of work related to our proposed system architecture.

Hypervisor-based system monitoring Hypervisor-based monitoring systems are a well-researched field, and multiple security-focused thin hypervisors have been proposed [14]. For example, *HIMA* provides hypervisor-based integrity measurements for a Virtual Machine (VM) as an extension of a general-purpose hypervisor [7]. *SecVisor* is a tiny hypervisor that ensures that only trusted code can execute at the kernel level [8]. Sharif et al. focus on high performance and design an in-VM monitoring system [9]. Chen et al. propose *Overshadow*, a system to protect the integrity of application data even if the OS is compromised [15]. *NumChecker* is a hypervisor-based system for detecting control-flow deviations in the kernel [16]. All of the systems rely on virtualization techniques and either require a hypervisor or act as a hypervisor.

TEE-based system monitoring. In contrast, a TEE-based approach like the TM does not block or occupy virtualization layers for system monitoring purposes. Further, a TEE-based design allows for reducing the TCB by offloading software isolation directly to the hardware. Several novel systems based on the ARM TrustZone have been proposed. For example, the ARM TrustZone can be used to protect the kernel's memory [17, 18]. Moreover, *TrustDump* allows for dumping the RAM of the OS even after a crash or compromise [19]. Busch et al. designed a TEE-based system that can monitor the configuration of peripherals for optical networks [20]. These approaches selectively utilize TEE-based isolation to achieve security properties.

We align ourselves with these approaches and propose the TM, a TEE-based approach for intrusion and anomaly detection. In this publication, we focus on the ARMv8-A architecture, which is already in use for the magnitude of IoT and edge-computing devices as of today. However, there are approaches fitted to Intel-based processors with Intel SGX enclaves as well. For example, Nakano et al. propose a system that is capable of inspecting the memory and storage of a target VM from within an SGX enclave [21]. Moreover, some works move Snort, an open-source network-based intrusion detection system, to an SGX enclave [22, 23]. In contrast to these, we do not inspect storage, memory, or network traffic but utilize hardware performance counters to profile the applications in the normal world. While the Intel SGX TEE has been deprecated in the consumer sector [24, p. 57], the ARM TrustZone TEE is continued with the upcoming ARMv9-A architecture [25], and we believe that the overall architecture of the TM is also compatible with Keystone [26], an open TEE framework for the RISC-V architecture.

Multiple recent works propose to use general machine-learning techniques (or at least inference) in a TEE [27–32]. In particular, Bayerl et al. protect machine-learning models from adversaries with a TEE [33]. The decryption of a model is bound to a TEE with the corresponding key. Similar to us, they use TensorFlow Lite in the TrustZone to conduct secure inference. However, we do not focus on protecting the model from adversaries but on detecting system anomalies.

Monitoring based on hardware performance counters As input for our anomaly detection framework, we rely on hardware performance counters, as provided by the Performance Monitoring Unit (PMU) of recent ARMv8-A CPUs. Given the significant amount of related work in this area, hardware performance counters are well-established for security purposes. For example, Xia et al. detect control-flow discrepancies with performance counters [34], Yuan et al. implement *Eunomia*, a detection framework for return-oriented programming [35], and Aweke et al. design a system to detect Rowhammer attacks [36]. Similar to us, Demme et al. use performance counters for the detection of running malware and identify that applications have distinguishable footprints [37]. Tang et al. rely on hardware performance counters and unsupervised machine learning for anomaly detection [38], while Bahador et al. and Singh et al. use support vector machines and decision trees to classify applications [39, 40]. Kuruvila et al. focus on an explainable machine-learning model for intrusion detection via hardware performance counters [41]. There are several approaches to detect CPU side-channel attacks based on machine learning [42–44]. Neither of these approaches relies on a TEE.

There are also limitations when using hardware performance counters for anomaly detection [45]. In particular, improper configuration can lead to indeterministic results, and depending on the architecture, hardware performance counters are subject to overcounting errors [46]. We share the opinion of Das et al. and Zhou et al. that hardware performance counters are not eligible for security monitoring *in every environment*, i.e., an arbitrary general-purpose device. We, however, focus on specialized devices executing monotonous but CPU-intensive tasks instead of changing interactive user sessions. For example, we consider high-end IoT or edge computing devices for video preprocessing or volumetric network stream processing and encoding (see Sect. 4). In this context, we can address the indeterminism and overcounting issues. One cause of overcounting lies in configuring the PMU in a way that generates hardware interrupts to halt the running program repeatedly after a certain number of recorded events. Both handling the interrupt and the interrupt skid cause overcounting. The latter describes the fact that the PMU interrupt does not immediately halt the CPU execution. Instead, an indeterministic amount of instructions is executed before the asynchronous interrupt arrives at the CPU. To mitigate this, our system does not rely on

PMU interrupts. Instead, we regularly sample the hardware performance counters from the secure world. While we deliberately do not offer per-process monitoring, we assume a monotonic CPU-intensive workload as the normal state and focus on detecting deviations from it. Moreover, we prevent the normal world from accessing the performance counter registers, impeding direct counter manipulations by an attacker. To the best of our knowledge, we are the first to propose a system monitoring framework that relies on hardware performance counters and uses machine learning techniques directly within the ARM TrustZone TEE.

Memory-safe and type-safe TEEs/TAs The memory-safe and type-safe Rust programming language has gained popularity. Thus, implementing system software in Rust is evident [47–49]. Cerdeira et al. show that contemporary TEEs/TAs are typically written in C and contain vulnerabilities [50]. RustZone [51] demonstrates the practical feasibility of developing TAs using memory-safe Rust code for the first time. With RusTEE [52], a Software Development Kit (SDK) for building TAs for ARM TrustZone is now publicly available. Using Rust for building TAs can be likewise found for other TEEs than the TrustZone. For example, Rust-SGX [53] and Fortanix Rust EDP [54] enable the use of Rust for the Intel SGX TEE.

Contrary to existing approaches, we combine and consolidate multiple ideas into one system architecture: Running within the ARM TrustZone TEE, our TM continuously collects performance metrics of the normal world by means of hardware performance counters. We then use a machine-learning model directly in the TEE to infer the running process. To prevent critical vulnerabilities (e.g., memory corruptions and type confusions) in the TCB, we implement major parts of our system in Rust.

3 Threat model

We consider an attacker who leverages remote communication channels to gain unauthorized access to the device. The adversary's goal is to gain control over the device to run malicious workloads. For example, the attacker might utilize the device for Distributed Denial of Service (DDoS) attacks [5, 6] or crypto mining [55]. We assume that the hardware works according to the specification of the manufacturer. Side-channel attacks on the secure world [56] are out of scope, as well as all forms of physical attacks on the device. Based on a hardware-protected Root of Trust (RoT), the integrity of the software components is ensured during boot time (secure boot), and a Chain of Trust (CoT) is formed, i.e., each software component ensures the integrity of the next component in the boot order. However, once the system is completely booted, no dynamic integrity measurement takes place. While initially benign, the normal world and its software are assumed to be vulnerable to attacks. For example, the attacker might exploit vulnerabilities in the system's network stack to take over the system [12, 13].

Our TCB consists of all software components in the secure world (see Sect. 4). Any other software (e.g., the operating system in the normal world) is untrusted and potentially compromised. This is a common threat model for TEE-based architectures.

To deceive a detection system based on hardware performance counters (like our TM), an attacker can attempt to mimic the performance metrics of a legitimate application (see Sect. 2). However, for this attack vector to be successful, the attacker requires extensive knowledge of the system, which we believe is challenging to acquire without affecting the system's behavior in a manner that would trigger TM's detection mechanisms in the first place. Furthermore, we expect the development of malware that matches the performance profile of a legal application to be highly complex, unstable, and error-prone.

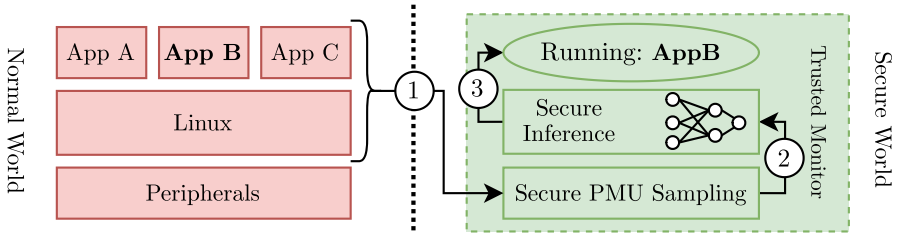


Fig. 1 Overview of the functionality of Trusted Monitor

4 Trusted Monitor

We first provide an overview of our proposed system architecture in Sect. 4.1. Subsequently, we outline a device model and usage profile in which TM can be deployed (see Sect. 4.2). We address the neural network architecture in Sect. 4.3 and explain the system components in-detail in a bottom-up approach (see Sect. 4.4).

4.1 Overview

Figure 1 gives an overview of the functionality of the TM. From a high-level perspective, the TM configures the PMU to continuously collect performance metrics of the normal world (①). We refer to this process as *secure* PMU sampling since the TM configures the PMU in a way that the normal world cannot interfere with or access the PMU (see Sect. 4.4). The collected performance metrics are fed into a machine-learning model (②). Like the PMU sampling, the machine learning model is executed directly in the secure world to prevent any influence from the normal world. We assume that it is possible to subdivide an application into a finite set of behavior phases (e.g., initialization phase and operative phase) [57, 58]. Thus, we train the model to detect the current application phase and infer the running process (③). If an unexpected system state is detected, we assume tampering with the system and raise an alarm. For example, if the TM detects the execution of an unknown application (e.g., malware), an administrator might get notified, or the system might be reset. The TM employs a machine-learning model to detect the current application phase and infer the running process. If an unexpected system state is detected, we assume tampering with the system and raise the alarm.

4.2 Device model

Because of indeterminism and overcounting issues, hardware performance counters are not applicable for security purposes in every environment (see Sect. 2). For this reason, we do *not* consider general-purpose computing systems. Instead, the main focus of our work is high-end IoT and edge computing devices with a comparably powerful CPU yet simple system behavior, which can be described as executing monotonous but CPU-intensive tasks instead of changing interactive user sessions on the device. Thus, we expect a weakly utilized scheduler, that is, a system with a comparably low number of processes and a prioritized, dominant process on each core. As a practical use case, we consider IoT devices that pre-process video streams. For example, modern surveillance equipment relies on visual computing and machine learning to automatically segment video streams [59, 60]. Another example is

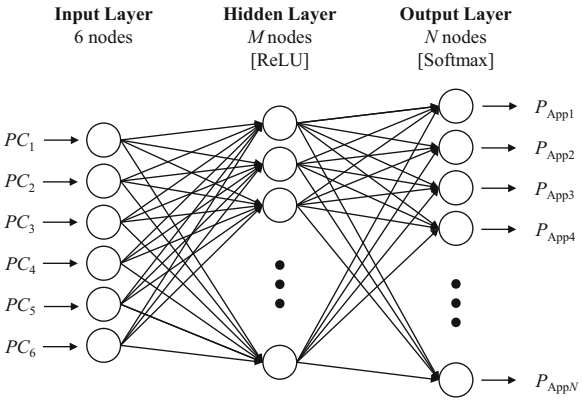


Fig. 2 Structure of the neural network model

network appliances and edge computing devices that continuously handle volumetric network streams can provide a suitable environment for our system. In particular, the widely-used DPDK offers a collection of data plane libraries and network interface controller polling-mode drivers that allow for high-speed packet processing of network packets directly in user-space processes. The toolkit enables telecommunication network providers to migrate performance-critical services, such as the backbone for mobile networks, to the cloud [61]. While the active polling for received packages reduces network latencies, it significantly increases CPU utilization, thereby facilitating an environment that is suited for the TM.

4.3 Neural network model

The TM conducts secure inference based on an Artificial Neural Network (ANN) directly from within the secure world. The conceptual process is as follows: We first train the model in a trusted environment. To do so, we collect and store performance metrics from the applications with their respective behavior phases that are meant to run on the device [57, 58]. With the collected performance metrics, we then use the machine-learning framework TensorFlow [62] for offline training of an ANN.

Neural-network structure Figure 2 shows the structure of the ANN used by the TM. On our evaluation platform (see Sect. 4.4), the PMU can monitor a maximum of six counters at a time. Thus, the input to the model is a <1x6> tensor consisting of the (normalized) performance counters PC_1 to PC_6. As our input is one-dimensional, we choose a multi-layer perceptron with a single fully connected hidden layer (<6xM>). The number of hidden neurons M is selected to be between the size of the input layer (6) and the size of the output layer (N). In the hidden layer, we use a Rectified Linear Unit (ReLU) activation function to mitigate vanishing gradients in the network. By contrast, we apply a Softmax activation function in the fully connected output layer (<MxN>). It converts the output to values that can be interpreted as probabilities. After applying the Softmax function, the elements of the output tensor (<1xN>) are in the range from zero to one and sum up to one. Each entry P_{App1} to $P_{\text{App}N}$ can be interpreted as the probability that one of the N trained application phases or processes is running in the normal world. We assume that the application with the highest probability runs in the normal world and do not use an unknown class that catches previously unseen applications. To evaluate TM (see Sect. 5), we rely on a training set with 183 application phases. Thus, we choose $N = 183$ and $M = 128$.

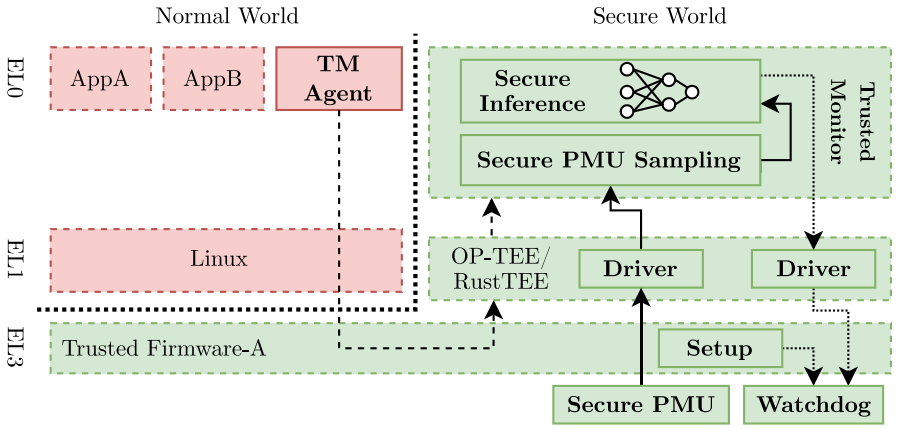


Fig. 3 System components of the Trusted Monitor

Training process The TM's ANN is trained using a supervised-learning approach that requires a large set of input data for the training process. To obtain the required data for training, the applications' behaviors and their performance metrics are recorded using the TM mechanisms while the system is still in a trustworthy environment. Each application that may legally run on the system is recorded for 100 s, resulting in the required amount of data that is used for the training of the model using TensorFlow running on a host machine with a powerful processor. Based on the recorded data, the model is trained for several epochs until the epoch accuracy is sufficiently high and the epoch loss is acceptably low enough to avoid overfitting to the data.

To conduct secure machine learning inference from within the secure world, we use an adjusted version of TensorFlow Lite for Microcontrollers [63] and include it as a building block in our system architecture.

4.4 System components

We implement the TM on a Marvell OCTEON TX2 CN913X development board, which has four Cortex-A72 ARMv8-A Central Processing Unit (CPUs), native support for the ARM TrustZone TEE, secure boot, and a PMU. Each of the ARM Cortex-A72 CPUs includes a PMU. The normal world is untrusted, whereas the secure world is part of the TCB. We extend standard software for the ARMv8-A architecture with custom components for the TM. In this Section, we refine the previously given overview of the TM. Figure 3 visualizes the components. Components in bold font extend the default software stack to implement the TM. We explain the system architecture in detail in a bottom-up approach.

An ARMv8-A CPU supports multiple hardware privilege layers, which are referred to as Exception Levels (ELs). EL3 is the most-privileged EL, and EL0 has the least privileges. Higher ELs can preempt the execution and access the data of less-privileged ELs. EL3 hosts the so-called secure monitor, which manages the context switches from and to the secure world OS. We deploy Trusted Firmware-A (TF-A), an open-source reference implementation for the secure monitor, to EL3 [64]. In the normal world, we use Linux as an OS. As a secure world OS, we use Open Source TEE (OP-TEE) [65] together with the RusTEE [52] extensions. OP-TEE allows implementing TAs which expose a function-based interface to the normal

world. Via the EL3 firmware, normal-world applications can issue a context switch to execute the exposed TA functions. In addition to that, RusTEE enables developers to implement TAs using the memory-safe and type-safe Rust programming language.

To reduce the risk caused by memory corruptions, null-pointer dereferences, and use-after-free vulnerabilities, we implement the TM as a Rust TA running on OP-TEE with the RusTEE extensions. The TM TA consists of 300 lines of custom Rust code, the Rust standard library, the Rust `libc` wrapper, and the RusTEE wrapper for the system call API towards OP-TEE. To allow for secure inference, we also build TensorFlow Lite for Microcontrollers, including our model as a static library, which we link against the Rust TA. Finally, we implement a small wrapper that allows us to call our ANN model from within the Rust code of the TA. Overall, the TM continuously reads out the PMU hardware performance counters, profiling the normal world. The retrieved values are fed into our ANN model to predict the application phase that is currently executing. If we detect a deviation in the workload over a user-configurable time, we raise an alarm. To implement the TM, we had to face several challenges, which are described in the following.

Scheduling the TM TA Traditionally, the secure world operating system does not contain a scheduler. Instead, the secure world is passive and only activated if the normal world issues the execution of a function that is exposed by a TA. We, however, depend on continuously executing the TM TA to sample the PMU hardware performance counters. To do so, we implement the TM agent, a Linux application, that regularly calls the TM TA (dashed arrow, Fig. 3). Since the TM agent is part of the normal world, it may be subject to compromise. Therefore, we need to assume that an adversary can influence the TM agent arbitrarily. To keep the attack surface small, we thus limit the interface between the TM agent and the TM to a single exposed function. Still, the attacker might try to prevent the execution of the TM completely (e.g., by killing the TM agent). For this reason, we rely on a mechanism based on hardware watchdogs.

Hardware watchdog Hardware watchdogs are widespread on IoT devices and are an established mechanism to react to a critical event. We force the normal world to schedule the TM TA indirectly [66–69]. Whenever the TM TA is called, the watchdog is reset, i.e., the pending system restart is deferred. However, if an attacker in the normal world prevents the scheduling of the TM, the watchdog expires and restarts the system. This requires that the normal world has no access to the watchdog. Therefore, we assign a hardware watchdog to the secure world, limiting access to software in the secure world only. Moreover, access to peripherals is, by design, not possible from a TA in OP-TEE. Thus, we extend OP-TEE with a driver for the hardware watchdog and extend OP-TEE with a new system call to expose the watchdog to the TM TA (dotted arrows, Fig. 3). Finally, the watchdog needs to be initially set up. The TM agent calls the TM TA for the first time when the normal world is fully booted. However, this also means that an attacker could try to prevent the start of the TM TA in the first place. Since the watchdog might not yet be enabled, the attacker would be able to take over the device indefinitely. To parry such attacks, the watchdog is initialized directly after a device reset. To do so, we add a watchdog setup routine to TF-A, which is executed before any normal-world system software.

PMU driver Similar to the hardware watchdog, directly accessing the PMU from an OP-TEE TA is not possible. Thus, we extend OP-TEE with a PMU driver and implement a system call that allows the TM TA to retrieve the PMU performance counter values (solid arrows, Fig. 3). In total, we add around 400 lines of PMU driver code to OP-TEE. By default, TF-A at EL3 saves and restores the PMU registers during a context switch from the normal world to the secure world and vice versa. This, however, hinders our approach that relies on extracting the

hardware performance counters of the normal world. Therefore, we also adjust TF-A such that the PMU registers are left untouched during a context switch.

Secure PMU We design our system with the idea that the PMU cannot be accessed from the normal world. However, restricting access to the PMU has proven to be challenging. There are two interfaces to the PMU. The first option is to directly access the PMU configuration registers (e.g., `PMUSERENR_EL0` [70]). Alternatively, the PMU is mapped in the physical address space and can be accessed with standard memory operations [71]. To implement the TM, the normal world must not be able to access the PMU configuration, neither via system registers nor the memory-mapped interface.

In the first step, we configure the system in a way that any normal world access to the PMU system *registers* raises an exception that is taken to EL3, which is part of the secure world (`MDCR_EL3.TDA` [72]). In the corresponding secure world handler, we avert the register access. Normal world software is neither able to prevent the trap nor overwrite the secure world handler. This is because it runs on secure world memory that is inaccessible from the normal world.

Restricting normal world accesses to the *memory-mapped* PMU interface is more challenging. This is because the operating system in the normal world can, in the first instance, map any address in the physical address space, depending on its own MMU configuration. Whether the normal world access to a physical address is successful depends on whether the address resolves to RAM or a memory-mapped peripheral. The TrustZone Address Space Controller (TZASC) enables partitioning RAM into secure and normal regions, preventing normal-world access to secure regions. Similarly, the TrustZone Protection Controller (TZPC) allows assigning peripherals to one of the worlds [73]. Since the memory-mapped PMU interface is not located in RAM, we cannot rely on the TZASC to protect them [71]. Despite not being supported by all hardware platforms, the memory-mapped PMU interface can be configured to be only accessible from the secure world through the TZPC. If the hardware platform supports it, we propose to configure the memory-mapped PMU interface as secure-world only. In combination with trapping normal world PMU register access, the PMU is now exclusive to the secure world, which is an important building block for our system architecture.

If the TZPC on a platform does not allow to restrict memory-mapped PMU access, there is an additional protection primitive we can rely on. The Platform Partition Controller (PPC), available on most off-the-shelf hardware of big manufacturers (e.g., NXP, Xilinx, Nvidia, Qualcomm, Broadcom, and Samsung), allows intercepting all requests on the system bus for configurable regions, regardless of the world the request originates from [74]. We then propose to dynamically configure the PPC to disallow any access to the memory-mapped PMU interface before entering the normal world and, respectively, allow access upon switching to the secure world. For a secure TM implementation, we require a platform that allows exclusively assigning the PMU to the secure world.

Our evaluation platform, the Marvell OCTEON TX2, does not support restricting access to the memory-mapped TM interface via the TZPC or the PPC and, thus, is susceptible to an attacker with knowledge about the existence and the internals of the TM to manipulate the performance counters and the PMU configuration systematically and, thus, trick the TM. This hardware limitation, however, does neither invalidate the concept of TM nor the platform's use as an evaluation platform. If suitable hardware is available (e.g., Boundary Device's Nitrogen8M board [75]), a secure implementation is possible.

5 Evaluation

In this Section, we evaluate the capabilities and the performance of the TM. At first, we compare the performance-counter values obtained from Linux's perf tool against the values from the TM to underline the validity of system-wide performance monitoring compared to process-level monitoring. Subsequently, we discuss the selection of performance counters, as hardware-based performance-monitoring units (such as the ARM PMU) can only monitor a limited number of performance counters simultaneously. Based on the selected performance counters, we evaluate the TM's ability to detect a real-world Linux rootkit. Finally, we conclude with an evaluation of the runtime overhead of the TM.

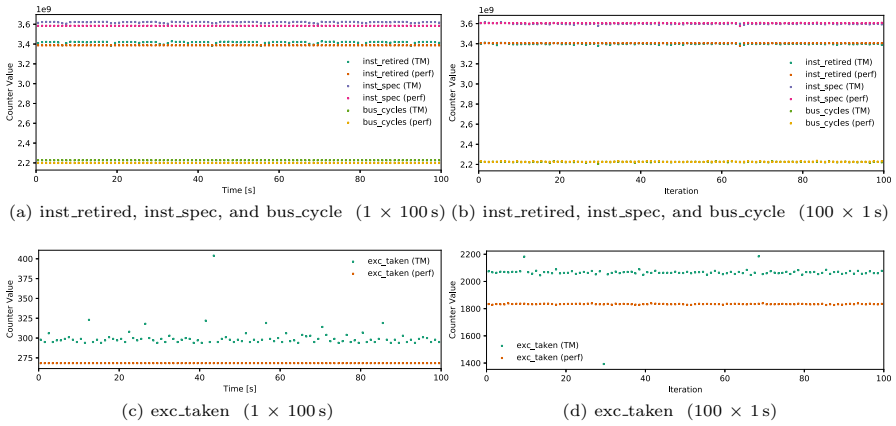
5.1 Evaluation setup

Training data To demonstrate the monitoring capabilities, we train our model to classify stress-ng stressors [76] that put stress on various system components (e.g., CPU or memory). In a production setup, the model would be trained to classify the applications running on the device instead. Several of the 200+ stress-ng stressors are not compatible with our platform (e.g., `bigheap` and `brk`), yielding 183 available stressors. To gain data for these, we execute the stressors individually while we collect the performance-counter values. For simplification, only one stressor runs simultaneously, with exactly one worker thread.
Performance counters In the following evaluations, we present measurements for several performance counters. The counter names used throughout the rest of the paper are taken from the ARM PMU documentation [77]. Different names used by, for instance, the perf tool have been mapped to the corresponding name from the PMU documentation.
Structure of the boxplots Unless stated otherwise, the structure of the boxplots is as follows: The box indicates the first and third quartiles; the pink line marks the median. The whiskers extend to the last performance-counter value greater/smaller than the first/third quartile minus/plus 1.5 times the interquartile range. The green markers show the individual outliers.

5.2 System and process monitoring

A system monitor like the TM observes all processes in the system, while a process monitor is scoped to a single process. Generally speaking, performance measurements from a system monitor have a higher mean variation since the OS and other running processes influence the measurements (see Sect. 2). Compared to a process monitor, a system monitor is less dependent on the operating system in the normal world and its internal data structures, which fosters stability during system updates. We claim that in an environment suitable for the TM, i.e., monotonous but CPU-intensive tasks (see Sect. 4), a process monitor yields results comparable to a system monitor. To assess this claim, we compare the measurement results from the TM and the perf tool, which is a process monitoring tool provided by the Linux kernel. We conduct two experiments to compare the performance-counter values for a comparably long-running application (100 s) and a short-running application (one second).

In the first experiment, the `cpu` stressor from stress-ng runs for 100 s while the TM collects the performance counters every second. Afterward, we again run the `cpu` stressor for 100 s while measuring the counters using the perf tool. We average the output per second. For better clarity, only four representative hardware performance counters are illustrated. The counter inst_retired describes the number of instructions architecturally executed, inst_spec



(a) inst_retired, inst_spec, and bus_cycle (1 × 100 s) (b) inst_retired, inst_spec, and bus_cycle (100 × 1 s)

(c) exc_taken (1 × 100 s) (d) exc_taken (100 × 1 s)

Fig. 4 Comparison of performance-counter values of an application that runs 100 s and one that runs 100 × 1 s. For each application, we obtain performance-counter values from Linux's perf tool (process monitoring) and our TM (system monitoring). Each marker represents a single measurement value

refers to the number of instructions speculatively executed, bus_cycles counts the number of bus cycles, and exc_taken is the number of interrupts. Figure 4 shows that the performance-counter values measured by the TM and the perf tool are comparable. Figure 4c shows the measurement for taken exceptions to accommodate for different value ranges and thus scales. The values differ slightly because the TM monitors the whole system and is, therefore, subject to variations caused by the OS. Thus, it is reasonable that the values from the TM are slightly higher. The values reported by perf are averaged, so they resemble a horizontal line.

In the second experiment, the `cpu` stressor runs 100 times for one second. The results of these measurements are shown in Figs. 4b and 4d. This time, the perf data is not averaged but sampled each second, like the TM data. We can see that the performance counters of the TM and perf differ slightly. However, the fluctuation of the TM values is less pronounced. Regarding the exc_taken counter, we recognize higher values when monitored with the TM and compared to the perf values. Interestingly, the counted exceptions differ from the number of exceptions when executing the long-running application. We assume that this is due to the higher amount of administrative tasks linked to repeatedly spawning the short-running process on the system: In case the application runs for 100 s, the code that interprets the arguments initializes the worker and exits the tool is executed only once. When executing the application 100 times for one second, all the surrounding tasks are also run through 100 times.

To sum up, the results from the two monitoring types are still very similar and differ only slightly. Thus, we use the user-friendly tool perf for a simplified data gathering for all performance counters to get an overview of the behavior of different performance counters for different stressors.

5.3 Selection of the performance counters

Given that our development board can monitor a maximum of six hardware performance counters at once, this Section focuses on the selection of a set of counters. Therefore, we compare three different sets of randomly chosen performance counters. To assess the different

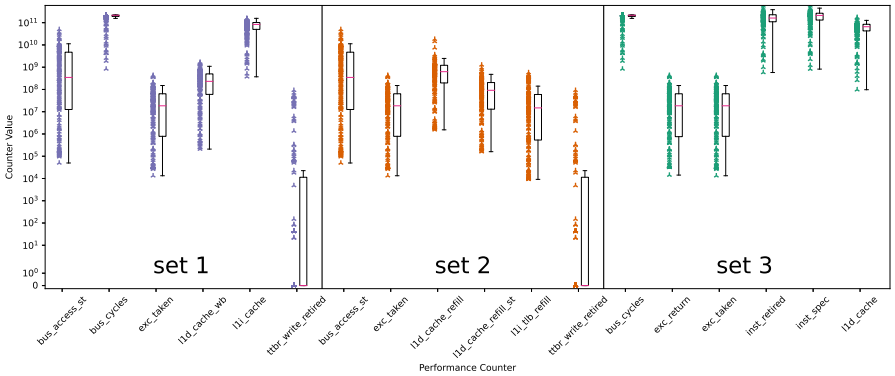


Fig. 5 Performance-counter values for all 183 stressors from stress-ng for all three sets. Each dot represents the performance-counter value for a single benchmark, the box indicates the first and third quartiles; the pink line marks the median. The whiskers extend to the last performance-counter value greater/smaller than the first/third quartile minus/plus 1.5 times the interquartile range

counters, all stressors are measured with the perf tool in time-based multiplexing mode. In multiplexing mode, perf reconfigures the event counters multiple times using the performance measuring framework of the Linux kernel. The kernel keeps track of measurement phases for each counter and finally interpolates the measurement values. For this evaluation, we sequentially execute all stress-ng stressors for 100 s.

Figure 5 illustrates the performance counters, grouped by the three sets, along with their observed values for all benchmarks from stress-ng. Each marker represents a single stressor; some remarkable stressors are discussed in the following.

The first set comprises the number of write accesses to the bus (bus_access_st), bus_cycles, exc_taken, the number of level 1 data-cache writebacks (l1d_cache_wb), the number of level 1 instruction-cache accesses (l1i_cache), and the number of architecturally executed writes the translation table base (ttbr_write_retired). The second set also includes the counters bus_access_st, exc_taken, and ttbr_write_retired. In addition, three other performance counters, the number of level 1 instruction TLB refills (l1i_tlb_refill), level 1 data-cache refills (l1d_cache_refill), and level 1 data-cache refills on writes (l1d_store_misses), are used. The third set contains the number of bus_cycles and the number of exc_taken already used in the first set. In addition, the number of exceptions returned (exc_return) is observed. Further, the third set includes inst_retired, inst_spec, and the number of level 1 data-cache accesses (l1d_cache).

Overall, all counters used show diverse values in comparatively large value ranges for different stressors. In some cases, however, the values for a single stressor or performance counter deviate for one or more stressors:

bus_access_st The number of bus write accesses exhibits high variations between $5 \cdot 10^4$ (`sigio`) and $4.9 \cdot 10^{10}$ (`numa`).

bus_cycles In comparison to the number of bus write accesses, the number of bus cycles is higher and visually denser, ranging from $8 \cdot 10^8$ (`sigio`) to $2.2 \cdot 10^{11}$ (`mmapfork`).

exc_taken The number of exceptions taken (i.e., the number of issued interrupts) is identical to the number of exceptions returned (assuming a normal system function) and ranges from $1.4 \cdot 10^4$ (`copy-file`) to $4.3 \cdot 10^8$ (`personality`).

inst_spec & inst_retired Both instruction-related performance counters exhibit high values for all stressors, ranging from $8 \cdot 10^8$ / $5 \cdot 10^8$ (`sigio`) to $5.6 \cdot 10^{11}$ (`shellsort`).

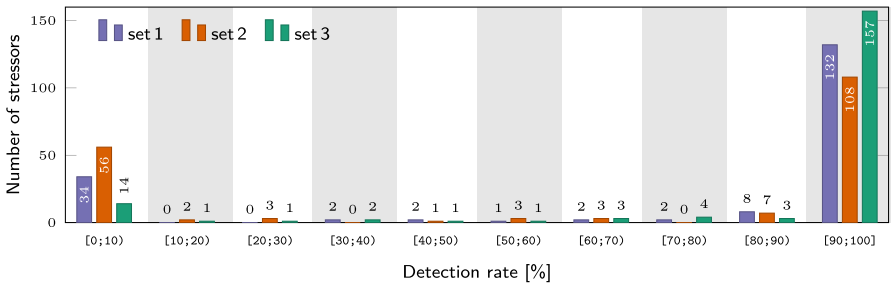


Fig. 6 Histogram over detection rates in 10% steps for all 183 stressors for all three sets. The detection rate is the ratio of correct predictions of a running stressor to the number of total predictions

l1d_cache Regarding the l1d_cache, the stressor `nop` has by far the lowest count of around $9.7 \cdot 10^7$.
l1d_cache_refill Some stressors are outstanding when looking at the number of level 1 data-cache refills, such as `matrix` and `lockf`, which both have very high counts of level 1 data-cache refills of around $18 \cdot 10^9$ and $14 \cdot 10^9$. `kill` has the third most refills, with a count of $7.3 \cdot 10^9$.
ttbr_write_retired Regarding the number of writes to the translation table base, the stressors can be subdivided into roughly three groups: For the stressors `sigpipe`, `sigpending`, `nice`, and others, the number of counted events is below 150, often 0. Stressors like `tlb-shootdown`, `lockf`, `lockofd`, and `zombie` are in the range between 2 000 and $3.5 \cdot 10^5$. Others like `sockmany`, `tee`, and `sigrt` exhibit high values in the range of $1.3 \cdot 10^6$ to $9.1 \cdot 10^7$.

In summary, the three presented sets of events are made up of performance counters with large value ranges for different stressors. For each performance counter, different stressors show particular behavior, indicating that different applications can be distinguished by their performance-counter values, as will be shown in the following.

5.4 Distinguishability of stressors

In this section, we evaluate the detection rates for all 183 stressors. Each stressor is executed for 100 s, and the TM conducts the inference process every second. After conducting the inference process, the TM logs the inferred stressor to the normal world via the serial interface. As a performance metric, we rely on a simple detection rate: We calculate the ratio of correct predictions to the number of total predictions. For instance, the detection rate of a stressor is 98.98% if it is recognized 98 times out of 99 triggers of the TM. On the other hand, if all decisions during a run of a specific stressor fail to log the correct application, the detection rate is 0.00%. Since there is no unknown class, the classic machine-learning metrics (precision and recall) cannot be determined meaningfully. Figure 6 shows a histogram of the detection rate for all stressors in 10% steps. From the three sets we analyzed, set 3 correctly (i.e., with an accuracy of above 90 %) classifies 157 out of our 183 evaluated stressors (86%), while the other sets have slightly lower accuracy (72 % for set 1, 59 % for set 2). The histogram illustrates that the detection rate for each stressor is either very high (in the bin [90; 100]) or very low (in the bin [90; 100], 0 % in most cases).

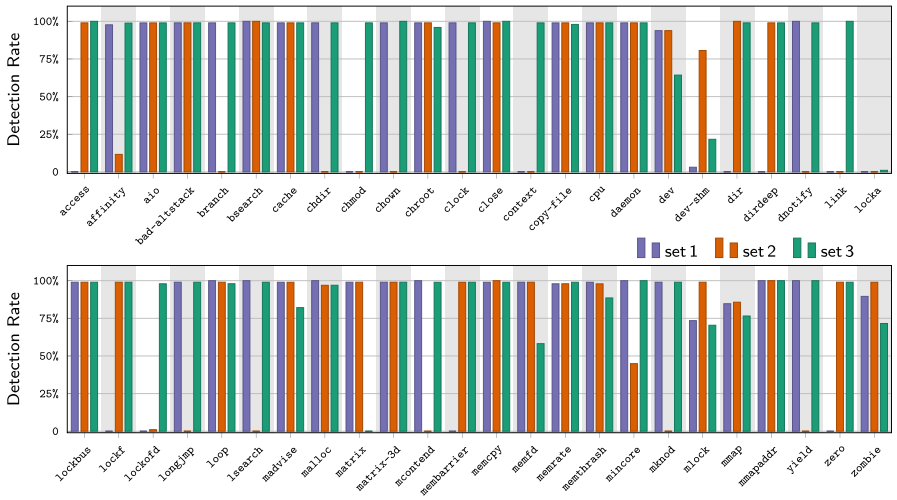


Fig. 7 Detection rates for a selection of 48 stressors and the three sets of performance counters (see Fig. 5). The detection rate is the ratio of correct predictions of a running stressor to the number of total predictions

In many of the instances with very low detection rates, this is caused by the high similarity between two stressors. Hence, the counted events for both tasks do not differ, and the TM has no means to distinguish them.

Figure 7 shows the individual detection rates for running a selection of 48 running stressors, including several noticeable stressors that will be detailed in the following. Subsequent to an introduction to our evaluation setup, we analyze the TM's behavior for two outstanding stressors `locka` and `mlock`:

locka The stressor `locka` has a detection rate of almost zero. Instead, the TM consistently classifies a running `locka` stressor as `lockofd`. `locka` and `lockofd` are very similar; both concurrently lock and unlock regions of a file using different locking mechanisms (POSIX advisory locking vs. Linux open file description locks, see fcntl(2)). Their similarity is also reflected in their runtime behavior: `locka` and `lockofd` exhibit similar performance-counter values with overlapping ranges. This makes reliable differentiation between `locka` and `lockofd` impossible.

While most of the stressors either are detected reliably (close to 100%) or not at all (close to 0%), the stressor `mmap` stands out with a detection rate of 86%. Sometimes, the TM classifies `mmap` as either `zero` or `urandom`; the detection rate for the stressors `urandom` and `zero` is 100%, excluding transition periods.

Figure 8 visualizes the performance counters while the `mmap` stressor is running: For every performance counter, the observed values for all three stressors are in a similar value range, and for most counters, the dense value ranges for `zero` and `urandom` overlap the relatively broad range for `mmap`. This relatively high fluctuation of counter values for `mmap` hinders reliable detection on the basis of a single measurement. The low variation of counter values for `zero` and `urandom`, on the other hand, facilitates a reliable detection of these stressors.

Overall, we observe a very high detection rate of the currently running application that varies depending on the chosen set of performance counters. For the best-performing set 3, 157 out of 183 (86 %) stressors have been detected with an accuracy of almost 100 %, while

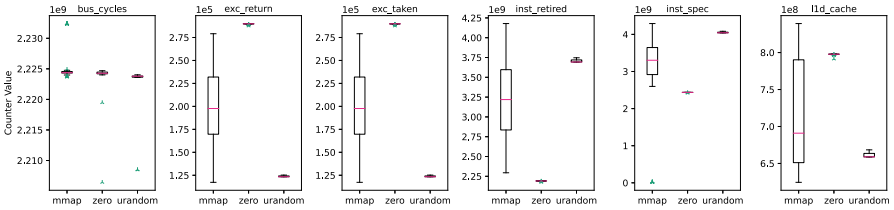


Fig. 8 Performance-counter values for the stressors `mmap`, `zero`, and `urandom`. Each dot represents the performance-counter values for a single benchmark, the box indicates the first and third quartiles; the pink line marks the median. The whiskers extend to the last performance-counter value greater/smaller than the first/third quartile minus/plus 1.5 times the interquartile range

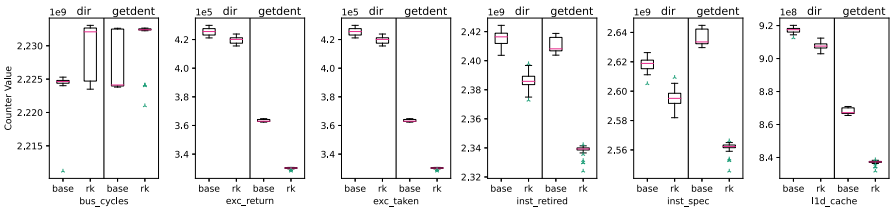


Fig. 9 Performance-counter values for stressor `dir` and stressor `getdent` without active rootkit (base) and with the Diamorphine rootkit activated (rk). The activation of the rootkit significantly influences the observed performance-counter values or variances

many of the non-detected stressors have an almost identical twin that hardly differs in terms of performance-counter values.

5.5 Rootkit detection

We use TM to indirectly detect the presence of Diamorphine [78], a Linux-kernel–module rootkit, based on deviations in the performance-counter values. For this evaluation, we use the performance counters from set 3 (due to their high rate of distinguished stressors). The Diamorphine rootkit finds and modifies the OS's syscall table and can become invisible (to the user), hide or unhide any process, grant any user root rights, and hide files. We analyze the rootkit's effect on the performance-counter values. Once no stressor or a non-expected stressor is detected, the TM classifies the system as being compromised.

dir and getdent The stressors `dir` and `getdent` both use the file system API that is modified by Diamorphine to hide files and directories. `dir` creates and removes directories using mkdir and rmdir, `getdent` recursively reads the directories /proc, /dev, /tmp, /sys, and /run using getdents and getdents64 [76]. `dir`, without an active rootkit, is recognized with a rate of 99%. With Diamorphine running, `dir` behaves differently w.r.t. the performance counters and thus has a detection rate of 0%. We observe similar behavior for `getdent`, whose detection rate declines from 99% (96/97) to 0% (0/97).

To reason the detection rates dropping to zero, Fig. 9 shows the measurement results for the baseline `dir` and `getdent` applications, as well as the applications with the rootkit running: When comparing the non-rootkit baselines with their values while the rootkit is running, all monitored performance counters show noticeable changes in the observed values or variances.

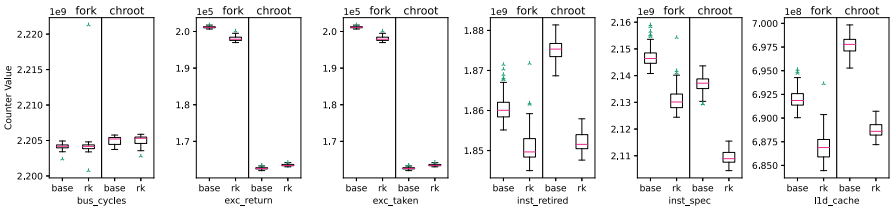


Fig. 10 Performance-counter values for stressor `fork` and stressor `chroot` without active rootkit (base) and with the Diamorphine rootkit activated (rk). `fork` and `chroot` overlap in the number of retired and speculated instructions

fork and chroot The `fork` stressor stands out because the detection rate of the TM goes down from almost 100% without the rootkit to nearly 0% with the rootkit enabled. When the rootkit is enabled, the TM mixes the `fork` stressor up with the `chroot` stressor. Figure 10 shows a comparison of the performance-counter values of `fork` and `chroot`: The number of bus cycles is similar for all four cases (`fork` and `chroot` with and without rootkit) and barely allows any conclusion regarding the running application. The number of exceptions taken and returned is slightly influenced by the rootkit for both stressors, but the value ranges for both stressors are still easily distinguishable. The number of retired and speculated instructions each are in a similar range, both with and without rootkit for both stressors, while the rootkit consistently reduces the observed values. The same applies to the number of level 1–data-cache accesses. These overlaps in the value range eventually lead to the model classifying `fork` as `chroot`.

The weights of the trained model seem to give higher precedence to the very good fit of the speculated instructions curve of the `chroot` stressor and the `fork` stressor with rootkit enabled. This is the case compared to the less good fits for the cache references and retired instructions.

To sum up, the enabled Diamorphine rootkit impacts multiple of the counted events. Depending on the running stressor, the rootkit can be detected with higher or lower accuracy. Stressors that do not use syscalls that have been modified by the rootkit are not impacted at all. This makes it challenging to predict how an attack, such as a rootkit, influences the system behavior and the performance-counter values in general. Still, we demonstrate that the TM is able to detect deviations from the normal application behavior and, in an indirect manner, the presence of a kernel-level rootkit.

5.6 Overhead

As the last evaluation of our TM, we evaluate the overhead of the periodic execution of the performance-counter collection and process inference. For the evaluation of the rootkit detection, the TM was configured to use a measurement interval of 1 s. However, the classification benefits from a higher measurement frequency, both due to the higher resolution and the reduced impact of task switching during a sampling period. A higher execution frequency, on the other hand, leads to an increased number of interruptions of the workload and thus increases the overhead.

We evaluate the overhead of multiple measurement frequencies by measuring the reduction in CPU performance for a 10 s workload. We determine the CPU performance by running the `CPU` stressor of stress-ng. The result is given in so-called Bogus Operations per Second (bogo), i.e., executed iterations of a CPU-intensive workload. Our baseline is a system without

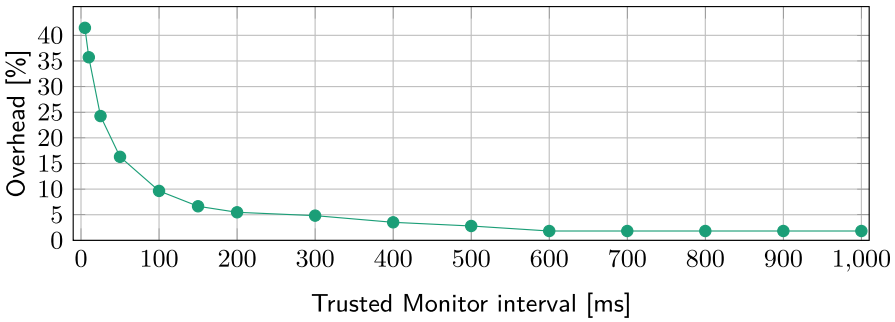


Fig. 11 CPU overhead of the TM for different measurement intervals in the range [5 ms; 1 s]. A lower interval leads to an increased number of context switches to the secure world, which reduces the CPU time for other tasks

the TM. Figure 11 illustrates the overhead for TM intervals between 5 ms and 1000 ms: For the interval of 1000 ms, the overhead is only 2% compared to the system without the TM being initialized. stress-ng reports 1534 bogo without TM and 1506 bogo with TM when running the CPU stressor for ten seconds. For an interval of 200 ms, the CPU performance is reduced by 5.5%, which might still be acceptable for some systems. A further reduction of the interval results in a noticeable performance reduction, reaching an overhead of 42% at an interval of 5 ms.

Summary Our evaluation shows that the TM correctly classifies 86% of the 183 stress-ng stressors, even though some of the stressors cannot be distinguished due to their similar runtime behavior (e.g., `locka` and `lockofd`). Because of its nature (see Sect. 4), we expect that an embedded application can be typically represented with considerably less than 183 stressors. During the evaluation, the TM was not only used to identify the currently running application but also to detect the kernel-level rootkit Diamorphine with high detection rates. The overhead induced by the system monitoring depends on the sampling frequency but stays below 2% for the configuration used during our evaluations.

6 Conclusion

With the widespread use of IoT devices, their security has become increasingly important. In this paper, we propose the TM, an anomaly-detection approach based on low-level system monitoring with hardware performance counters. Our approach relies on a TEE to be securely isolated from both applications and the operating system. This isolation prevents tampering with the TM, even if the untrusted operating system is compromised. Circumventing the TM is highly unlikely due to the potential attack's influence on the system behavior. Once deviations from the expected performance-counter values are detected, our TM reveals the attack and activates countermeasures (e.g., a system reset).

In our evaluation, we show that hardware performance counters characterize the system behavior and can be used to detect unwanted deviations. Using the example of the stress-ng stressors, we can infer the currently running application from within the TM. Our evaluation further demonstrates that our TM is able to detect the impact of the open-source rootkit Diamorphine on the system. The trained ANN model reliably detects unexpected deviations in the system performance counters when the rootkit is activated.

The overhead induced by our TM stays below 2% in the configuration used during our evaluation, yet the TM correctly classifies the current application with a high detection rate.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—project numbers 465958100 ("NEON"); 502228341 ("Memento") and by the Bundesministerium für Bildung und Forschung (BMBF) for the project AI-NET-ANTILLAS 16KIS1305, 16KIS1314, and 16KIS1315.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Cisco (2020) Annual internet report. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Accessed 16 June 2021
2. Transforma Insights (2022) Number of IoT connected devices worldwide 2019–2021, with forecasts to 2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. Accessed 16 June 2023
3. McAfee Labs threats report (2021). https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-apr-2021.pdf. Accessed 17 June 2021
4. CVE-2021-3156 (2021). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156. Accessed 16 June 2021
5. Antonakakis M et al (2017) Understanding the Mirai botnet. In: Proceedings of the 26th USENIX security symposium (USENIX Security '17), pp 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis
6. Edwards S, Profetis I (2016) Hajime: analysis of a decentralized internet worm for IoT devices. https://www.cs.umd.edu/class/spring2021/cmsc614/papers/hajime-rapidity.pdf. Accessed 4 Dec 2022
7. Azab A, Ning P, Sezer E, Zhang X (2009) HIMA: a hypervisor-based integrity measurement agent. In: Proceedings of the 2009 annual computer security applications conference (ACSAC '09), pp 461–470. https://doi.org/10.1109/ACSAC.2009.50
8. Seshadri A, Luk M, Qu N, Perrig A (2007) SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of the 21st ACM symposium on operating systems principles (SOSP '07), pp 335–350. https://doi.org/10.1145/1294261.1294294
9. Sharif MI, Lee W, Cui W, Lanzi A (2009) Secure in-VM monitoring using hardware virtualization. In: Proceedings of the 16th ACM conference on computer and communications security (CCS '09), pp 477–487. https://doi.org/10.1145/1653662.1653720
10. Garfinkel T, Rosenblum M (2003) A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the network and distributed system security symposium (NDSS '03). https://www.ndss-symposium.org/ndss2003/virtual-machine-introspection-based-architecture-intrusion-detection/
11. Dunlap GW, King ST, Cinar S, Basrai MA, Chen PM (2002) ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In: Proceedings of the 5th symposium on operating system design and implementation (OSDI '02). http://www.usenix.org/events/osdi02/tech/dunlap.html
12. Kol M, Oberman S (2020) Ripple20. https://www.jsof-tech.com/wp-content/uploads/2020/06/JSOF_Ripple20_Technical_Whitepaper_June20.pdf. Accessed 21 Apr 2023

13. Forescout Research Labs (2020) How TCP/IP Stacks Breed Critical Vulnerabilities in IoT, OT and IT Devices. https://www.forescout.com/company/resources/amnesia33-how-tcp-ip-stacks-breed-critical-vulnerabilities-in-iot-ot-and-it-devices/. Accessed 21 Apr 2023
14. Bauman E, Ayoade G, Lin Z (2015) A survey on hypervisor-based monitoring: approaches, applications, and evolutions. ACM Comput Surv CSUR 48:1–33. https://doi.org/10.1145/2775111
15. Chen X et al (2008) Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In: Proceedings of the 13th international conference on architectural support for programming languages and operating systems (ASPLOS '08), pp 2–13. https://doi.org/10.1145/1346281.1346284
16. Wang X, Karri R (2013) NumChecker: detecting kernel control-flow modifying rootkits by using hardware performance counters. In: Proceedings of the 50th annual design automation conference 2013 (DAC '13), pp 79:1–79:7. https://doi.org/10.1145/2463209.2488831
17. Guan L et al (2017) TrustShadow: secure execution of unmodified applications with ARM TrustZone. In: Proceedings of the 15th annual international conference on mobile systems, applications, and services (MobiSys '17), pp 488–501. https://doi.org/10.1145/3081333.3081349
18. Azab AM et al (2014) Hypervision across worlds: real-time kernel protection from the ARM TrustZone secure world. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security (CCS '14). https://doi.org/10.1145/2660267.2660350
19. Sun H, Sun K, Wang Y, Jing J, Jajodia S (2014) TrustDump: reliable memory acquisition on smartphones. In: Proceedings of the 19th European symposium on research in computer security (ESORICS '14). https://doi.org/10.1007/978-3-319-11203-9_12
20. Busch M, Schlenk R, Heckel H (2019) TEEMo: trusted peripheral monitoring for optical networks and beyond. In: Proceedings of the 4th workshop on system software for trusted execution (SysTEX'19), pp 7:1–7:6. https://doi.org/10.1145/3342559.3365339
21. Nakano T, Kourai K (2021) Secure offloading of intrusion detection systems from VMs with Intel SGX. In: Proceedings of the 14th IEEE international conference on cloud computing (CLOUD '21), pp 297–303. https://doi.org/10.1109/CLOUD53861.2021.00043
22. Kuvaiskii D, Chakrabarti S, Vij M (2018) Snort intrusion detection system with Intel Software Guard Extension (Intel SGX). arXiv:1802.00508
23. Shih M, Kumar M, Kim T, Gavrilovska A (2016) S-NFV: securing NFV states by using SGX. In: Proceedings of the ACM international workshop on security in software defined networks & network function virtualization (SDN-NFV '16), pp 45–48. https://doi.org/10.1145/2876019.2876032
24. Intel (2022) 12th generation Intel Core processors datasheet. https://www.intel.com/content/www/us/en/products/docs/processors/core/core-technical-resources.html. Accessed 22 June 2022
25. ARM Limited (2021) ARM's solution to the future needs of AI, security and specialized computing is v9. https://www.arm.com/company/news/2021/03/arms-answer-to-the-future-of-ai-armv9-architecture. Accessed 22 June 2022
26. Lee D, Kohlbrenner D, Shinde S, Asanović K, Song D (2020) Keystone: an open framework for architecting trusted execution environments. In: Proceedings of the 15th European conference on computer systems (EuroSys '20). https://doi.org/10.1145/3342195.3387532
27. Mohassel P, Rosulek M, Trieu N (2020) Practical privacy-preserving K-means clustering. In: Proceedings of the privacy enhancing technologies (PoPETs '20), pp 414–433. https://doi.org/10.2478/popets-2020-0080
28. Corrigan-Gibbs H, Boneh D (2017) Prio: private, robust, and scalable computation of aggregate statistics. In: Proceedings of the 14th USENIX symposium on networked systems design and implementation (NSDI '17), pp 259–282. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs
29. Hunt T, Song C, Shokri R, Shmatikov V, Witchel E (2018) Chiron: privacy-preserving machine learning as a service. arXiv preprint arXiv:1803.05961
30. Melis L, Song C, De Cristofaro E, Shmatikov V (2019) Exploiting unintended feature leakage in collaborative learning. In: Proceedings of the 40th IEEE symposium on security and privacy (S&P '19), pp 691–706. https://doi.org/10.1109/SP.2019.00029
31. Ohrimenko O et al (2016) Oblivious multi-party machine learning on trusted processors. In: Proceedings of the 25th USENIX security symposium (USENIX security '16), pp 619–636. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko
32. Kumar N et al (2020) CrypTFlow: secure TensorFlow inference. In: Proceedings of the 41st IEEE symposium on security and privacy (S&P '20), pp 336–353. https://doi.org/10.1109/SP40000.2020.00092
33. Bayerl SP et al (2020) Offline model guard: secure and private ML on mobile devices. In: Proceedings of the 2020 design, automation & test in Europe conference & exhibition (DATE '20), pp 460–465. https://doi.org/10.23919/DATE48585.2020.9116560

34. Xia Y, Liu Y, Chen H, Zang B (2012) CFIMon: detecting violation of control flow integrity using performance counters. In: IEEE/IFIP international conference on dependable systems and networks (DSN '12), pp 1–12. https://doi.org/10.1109/DSN.2012.6263958
35. Yuan L, Xing W, Chen H, Zang B (2011) Security breaches as PMU deviation: detecting and identifying security attacks using performance counters. In: Proceedings of the Asia Pacific workshop on systems (APSys '11), pp 1–5. https://doi.org/10.1145/2103799.2103807
36. Aweke ZB et al (2016) ANVIL: software-based protection against next-generation row hammer attacks. In: Proceedings of the ACM conference on architectural support for programming languages and operating systems (ASPLOS '16), pp 743–755. https://doi.org/10.1145/2872362.2872390
37. Demme J et al (2013) On the feasibility of online malware detection with performance counters. In: Proceedings of the 40th annual international symposium on computer architecture (ISCA '13), pp 559–570. https://doi.org/10.1145/2485922.2485970
38. Tang A, Sethumadhavan S, Stolfo SJ (2014) Unsupervised anomaly-based malware detection using hardware features. In: Proceedings of the 17th international symposium on research in attacks, intrusions and defenses (RAID '14), pp 109–129. https://doi.org/10.1007/978-3-319-11379-1_6
39. Bahador MB, Abadi M, Tajoddin A (2014) HPCMalHunter: behavioral malware detection using hardware performance counters and singular value decomposition. In: 4th international conference on computer and knowledge engineering (ICCKE '14), pp 703–708. https://doi.org/10.1109/ICCKE.2014.6993402
40. Singh B, Evtyushkin D, Elwell J, Riley R, Cervesato I (2017) On the detection of kernel-level rootkits using hardware performance counters. In: Proceedings of the ACM Asia conference on computer and communications security (AsiaCCS '17), pp 483–493. https://doi.org/10.1145/3052973.3052999
41. Kuruvila AP, Meng X, Kundu S, Pandey G, Basu K (2022) Explainable machine learning for intrusion detection via hardware performance counters. IEEE Trans Comput Aided Des Integr Circuits Syst 41:4952–4964. https://doi.org/10.1109/TCAD.2022.3149745
42. Mushtaq M et al (2018) NIGHTs-WATCH: a cache-based side-channel intrusion detector using hardware performance counters. In: Proceedings of the 7th international workshop on hardware and architectural support for security and privacy (HASP '18), pp 1:1–1:8. https://doi.org/10.1145/3214292.3214293
43. Li C, Gaudiot J (2022) Detecting Spectre attacks using hardware performance counters. IEEE Trans Comput 71:1320–1331. https://doi.org/10.1109/TC.2021.3082471
44. Zhang Y, Makris Y (2020) Hardware-based detection of spectre attacks: a machine learning approach. In: Proceedings of the Asian hardware oriented security and trust symposium (AsianHOST '20), pp 1–6. https://doi.org/10.1109/AsianHOST51057.2020.9358255
45. Zhou B, Gupta A, Jahanshahi R, Egele M, Joshi A (2018) Hardware performance counters can detect malware: myth or fact? In: Proceedings of the Asia conference on computer and communications security (AsiaCCS '18), pp 457–468. https://doi.org/10.1145/3196494.3196515
46. Das S, Werner J, Antonakakis M, Polychronakis M, Monrose F (2019) SoK: the challenges, pitfalls, and perils of using hardware performance counters for security. In: 2019 IEEE symposium on security and privacy (S&P '19), pp 20–38. https://doi.org/10.1109/SP.2019.00021
47. Levy A et al (2017) The case for writing a kernel in rust. In: Proceedings of the 8th Asia-Pacific workshop on systems (APSys '17), pp 1:1–1:7. https://doi.org/10.1145/3124680.3124717
48. Levy AA et al (2015) Ownership is theft: experiences building an embedded OS in rust. In: Proceedings of the 8th workshop on programming languages and operating systems (PLOS '15), pp 21–26. https://doi.org/10.1145/2818302.2818306
49. Levy A et al (2017) Multiprogramming a 64kB computer safely and efficiently. In: Proceedings of the 26th symposium on operating systems principles (SOSP '17), pp 234–251. https://doi.org/10.1145/3132747.3132786
50. Cerdeira D, Santos N, Fonseca P, Pinto S (2020) SoK: understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In: Proceedings of the 41st IEEE symposium on security and privacy (S&P '20), pp 1416–1432. https://doi.org/10.1109/SP40000.2020.00061
51. Evenchick E (2018) RustZone: writing trusted applications in rust. https://github.com/ericevenchick/rustzone. Accessed 23 May 2021
52. Wan S, Sun M, Sun K, Zhang N, He X (2020) RusTEE: developing memory-safe ARM TrustZone applications. In: Proceedings of the 2020 annual computer security applications conference (ACSAC '20), pp 442–453. https://doi.org/10.1145/3427228.3427262
53. Wang H et al (2019) Towards memory safe enclave programming with Rust-SGX. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security (ACM CCS'19), pp 2333–2350. http://dx.doi.org/10.1145/3319535.3354241
54. Fortanix (2019) Enclave development platform. https://edp.fortanix.com. Accessed 23 May 2021
55. Ngo Q, Nguyen H, Le V, Nguyen D (2020) A survey of IoT malware and detection methods based on static features. Inf Commun Technol ICT Express 6:280–286. https://doi.org/10.1016/j.icte.2020.04.005

56. Zhang N, Sun K, Shands D, Lou W, Hou YT (2018) TruSense: information leakage from trustzone. In: Proceedings of the 2018 IEEE conference on computer communications (INFOCOM '18), pp 1097–1105. https://doi.org/10.1109/INFOCOM.2018.8486293
57. Dhodapkar AS, Smith JE (2003) Comparing program phase detection techniques. In: Proceedings of the 36th annual IEEE/ACM international symposium on microarchitecture (MICRO-36), pp 217–227. https://doi.org/10.1109/MICRO.2003.1253197
58. Hamerly G, Perelman E, Lau J, Calder B (2005) Simpoint 3.0: faster and more flexible program phase analysis. J Instr Level Parallelism 7:1–28
59. Ke R, Zhuang Y, Pu Z, Wang Y (2021) A smart, efficient, and reliable parking surveillance system with edge artificial intelligence on IoT devices. IEEE Trans Intell Transp Syst 22:4962–4974. https://doi.org/10.1109/TITS.2020.2984197
60. Ling X, Sheng J, Baiocchi O, Liu X, Tolentino ME (2017) Identifying parking spaces & detecting occupancy using vision-based IoT devices. In: Proceedings of the 2017 global internet of things summit (GIoTS '17), pp 1–6. https://doi.org/10.1109/GIOTS.2017.8016227
61. DPDK Project—Linux Foundation, LLC (2023) About DPDK. https://www.dpdk.org/about/. Accessed 3 June 2023
62. Abadi M et al (2015) TensorFlow: large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/. Accessed 14 May 2021
63. TensorFlow Developers (2021) TensorFlow lite. https://www.tensorflow.org/lite/guide. Accessed 22 May 2021
64. ARM Limited (2021) Trusted firmware-A. https://github.com/ARM-software/arm-trusted-firmware. Accessed 1 Dec 2021
65. TrustedFirmware.org (2020) OP-TEE documentation. https://optee.readthedocs.io/en/latest/general/about.html. Accessed 27 Apr 2021
66. Xu M et al (2019) Dominance as a new trusted computing primitive for the internet of things. In: Proceedings of the 40th IEEE symposium on security and privacy (S&P '19), pp 1415–1430. https://doi.org/10.1109/SP.2019.00084
67. Huber M, Hristozov S, Ott S, Sarafov V, Peinado M (2020) The Lazarus effect: healing compromised devices in the internet of small things. In: Proceedings of the 15th ACM Asia conference on computer and communications security (AsiaCCS '20), pp 6–19. https://doi.org/10.1145/3320269.3384723
68. Suzaki K, Tsukamoto A, Green A, Mannan M (2020) Reboot-oriented IoT: life cycle management in trusted execution environment for disposable IoT devices. In: Proceedings of the 2020 annual computer security applications conference (ACSAC '20), pp 428–441. https://doi.org/10.1145/3427228.3427293
69. Röckl J, Protsenko M, Huber M, Müller T, Freiling FC (2021) Advanced system resiliency based on virtualization techniques for IoT devices. In: Proceedings of the 2021 annual computer security applications conference (ACSAC '21), pp 455–467. https://doi.org/10.1145/3485832.3485836
70. ARM Limited (2020) Armv8-A architecture registers: PMUSERENR-EL0. https://developer.arm.com/documentation/ddi0595/2020-12/AArch64-Registers/PMUSERENR-EL0--Performance-Monitors-User-Enable-Register. Accessed 2021-06-22
71. Ning Z, Zhang F (2017) Ninja: towards transparent tracing and debugging on ARM. In: Proceedings of the 26th USENIX security symposium (USENIX Security '17), pp 33–49. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning
72. ARM Limited (2022) Armv8-A architecture registers: MDCR_EL3. https://developer.arm.com/documentation/ddi0601/2022-03/AArch64-Registers/MDCR-EL3-Monitor-Debug-Configuration-Register--EL3-?lang=en. Accessed 22 May 2023
73. Pinto S, Santos N (2019) Demystifying Arm TrustZone: a comprehensive survey. ACM Comput Surv 51:130:1-130:36. https://doi.org/10.1145/3291047
74. Cerdeira D, Martins J, Santos N, Pinto S (2022) Rezone: disarming TrustZone with TEE privilege reduction. In: Proceedings of the 31st USENIX security symposium (USENIX security '22), pp 2261–2279. https://www.usenix.org/conference/usenixsecurity22/presentation/cerdeira
75. Boundary Devices (2023) Nitrogen8M. https://boundarydevices.com/product/nitrogen8m/. Accessed 3 June 2023
76. Canonical Ltd (2017) stress-ng—a tool to load and stress a computer system. https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html. Accessed 26 May 2021
77. Arm Limited (2016) ARM Cortex-A72 MPCore processor. https://developer.arm.com/documentation/100095/0003/. Accessed 12 Apr 2021
78. m0nad (2021) Diamorphine. https://github.com/m0nad/Diamorphine. Accessed 26 May 2021