RESEARCH-ARTICLE

Trace of the Times: Rootkit Detection through Temporal Anomalies in Kernel Activity

MAX LANDAUER, Austrian Institute of Technology, Vienna, Austria

LEONHARD ALTON, Austrian Institute of Technology, Vienna, Austria

MARTINA LINDORFER, Vienna University of Technology, Vienna, Vienna, Austria

FLORIAN SKOPIK, Austrian Institute of Technology, Vienna, Austria

MARKUS WURZENBERGER, Austrian Institute of Technology, Vienna, Austria

WOLFGANG HOTWAGNER, Austrian Institute of Technology, Vienna, Austria

Open Access Support provided by:

Austrian Institute of Technology

Vienna University of Technology

Published: 15 December 2025
Online AM: 01 October 2025
Accepted: 25 September 2025
Revised: 10 September 2025
Received: 14 April 2025

Trace of the Times: Rootkit Detection through Temporal Anomalies in Kernel Activity

MAX LANDAUER and LEONHARD ALTON, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria
MARTINA LINDORFER, TU Wien, Vienna, Austria
FLORIAN SKOPIK, MARKUS WURZENBERGER, and WOLFGANG HOTWAGNER, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria

Kernel-space rootkits provide adversaries with permanent high-privileged access to compromised systems and are often a key element of sophisticated attack chains. At the same time, they enable stealthy operation and are thus difficult to detect. Thereby, they inject code into kernel functions to appear invisible to users, for example, by manipulating file enumerations. Existing detection approaches are insufficient because they rely on signatures that are unable to detect novel rootkits or require domain knowledge about the rootkits to be detected. To overcome this challenge, our approach leverages the fact that runtimes of kernel functions targeted by rootkits increase when additional code is executed. The framework outlined in this article injects probes into the kernel to measure timestamps of functions within relevant system calls, computes distributions of function execution times, and uses statistical tests to detect time shifts. The evaluation of our open source implementation on publicly available datasets indicates high detection accuracy with an F1 score of 98.7% across five scenarios with varying system states.

CCS Concepts: • **Security and privacy** → *Malware and its mitigation*; **Intrusion detection systems**; **Operating systems security**; • **Computing methodologies** → **Semi-supervised learning settings**;

Additional Key Words and Phrases: rootkit detection, anomaly detection, kernel tracing

ACM Reference format:
Max Landauer, Leonhard Alton, Martina Lindorfer, Florian Skopik, Markus Wurzenberger, and Wolfgang Hotwagner. 2025. Trace of the Times: Rootkit Detection through Temporal Anomalies in Kernel Activity. *Digit. Threat. Res. Pract.* 6, 4, Article 33 (December 2025), 26 pages.
https://doi.org/10.1145/3770085

Parts of this work were carried out in course of a Master's Thesis at the Vienna University of Technology [1]. The work in this paper has received funding from the European Union—Horizon Europe Research and Innovation program under GA no. 101168144 (MIRANDA) and European Defence Fund under GA no. 101121403 (NEWSROOM). Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union. The European Union cannot be held responsible for them.
Authors' Contact Information: Max Landauer (corresponding author), Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: max.landauer@ait.ac.at; Leonhard Alton, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: leonhard.alton@ait.ac.at; Martina Lindorfer, TU Wien, Vienna, Austria; e-mail: martina.lindorfer@tuwien.ac.at; Florian Skopik, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: florian.skopik@ait.ac.at; Markus Wurzenberger, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: markus.wurzenberger@ait.ac.at; Wolfgang Hotwagner, Digital Safety and Security, Austrian Institute of Technology GmbH, Wien, Austria; e-mail: wolfgang.hotwagner@ait.ac.at.

ACM 2576-5337/2025/12-ART33
https://doi.org/10.1145/3770085

1 Introduction

Among the many types of cyber attacks that security professionals need to deal with on a regular basis, kernel-space rootkits pose a particularly severe threat. Once installed, these rootkits provide attackers with permanent access to compromised systems and enable the execution of arbitrary commands with high (root) privileges; hence, the name rootkit [10]. On top of that, kernel-space rootkits are designed to effectively make themselves invisible to users by interfering with low-level functions of the operating system's kernel to hide their own presence [3, 37]. Real-world examples of cyber attacks involving rootkits can be found in many recent threat reports, for example, the 2020 CrowdStrike Global Threat Report [11] mentions an attack that involved a customized version of a publicly available rootkit that was used to interfere with system functions for stealthy operation. Other sources report the use of rootkits to prevent detection of cryptocurrency-mining malware [20], hiding of a malicious shared library and overwriting symbolic links [24], and hiding of network connections when redirecting network traffic [8].

Most common intrusion detection mechanisms rely on the recognition of signatures such as byte patterns that are known to correspond to certain malware [3]. Unfortunately, this strategy is generally not sufficient for comprehensive protection against rootkits, because no signatures are available for novel rootkits or modified versions of existing ones [7]. As a consequence, alternative detection methods, such as cross-view that locates discrepancies between different system levels to identify hidden objects, have been investigated in the past [28].

However, in their recently published survey, Stühn et al. [38] conclude that prevalent mechanisms for rootkit detection are insufficient. Their evaluation of several common intrusion detection systems demonstrates that no single solution is capable of reliably detecting various types of rootkits. In addition, they find that the detection performance of these tools heavily depends on domain knowledge about the compromised system and the deployed rootkits, which limits their applicability in real-world use-cases. Moreover, Nadim et al. [28] emphasize the problems of forensic approaches and express the need for an intelligent and lightweight approach that is capable of detecting novel rootkits at runtime. In this article, we therefore propose a generalized approach for real-time detection of kernel-space rootkits based on statistical and semi-supervised anomaly detection techniques. The main idea behind our approach is that rootkits need to inject code when interfering with the operating systems to hide their presence, which causes that the overall runtime of that modified code block increases, because additional instructions need to be executed [7, 25]. Our detection method thus captures normal time intervals of and between executed kernel functions within system calls and recognizes significant delays with respect to these normal distributions as indicators for rootkits. Thereby, our approach goes beyond existing works that only use execution times of entire system calls, neglect multimodal features in collected datasets, and do not consider the influence of system conditions and noise [4, 14, 26]. We emphasize that our approach does not aim to replace any existing security measures against rootkits, such as signature-based detection approaches or cross-view, but instead provides an additional line of defense that complements state-of-the-art detection systems.

There are several challenges in designing a reliable detection system based on shifts in distributions of function timings. On the one hand, there is a vast number of potentially relevant functions in the kernel that could be considered for analysis, and there is no trivial way to collect time measurements from all of them. On the other hand, the runtime of functions often depends on their context of execution, which makes some of them unreliable for rootkit detection and sources of false alerts. With this article, we aim to overcome these challenges by answering the following research questions. *RQ1: What system calls enable the observation of rootkits that hide files? RQ2: How can delays of relevant function calls be observed? RQ3: To what degree can anomaly detection techniques leverage system call function timings to uncover hidden rootkit activities?*

While investigating the topic of rootkit detection, we noticed that one of the biggest problems that currently holds back research in the area of anomaly-based rootkit detection is the lack of data that can be used to evaluate rootkit detection approaches [28]. We therefore publish the datasets generated as part of our evaluation online.[1]

[1] https://zenodo.org/records/14679675.

Moreover, our review of open source rootkits for the Linux operating system showed that none of them are able to run on modern kernel versions. We therefore provide the rootkit developed in course of this study as open source code.[2] To facilitate reproducibility of the results presented in this article, we also publish the implementation of our probing and detection mechanisms online.[3] We hope that this will encourage others to extend our evaluation and generate even more public datasets for future research. We summarize our contributions as follows:

—a framework and open source implementation for kernel tracing with **Extended Berkeley Packet Filter (eBPF)** probes,
—an open source rootkit and publicly available datasets of kernel function time measurements, and
—a detection mechanism for delayed function call timings.

The remainder of this article is structured as follows: Section 2 reviews rootkits and detection mechanisms. Section 3 outlines the concept of our approach. In Section 4, we investigate relevant kernel functions and explain how we use probes for time measurement. In Section 5, we describe our approach for detection of anomalies. We evaluate our approach in Section 6 and discuss the results in Section 7. Finally, Section 8 concludes this article.

2 Background and Related Work

In this section, we provide a review of existing open source rootkits and discuss related works in the research area of rootkit detection.

2.1 Rootkits

This section explains the definition of a rootkit, enumerates common methods used by rootkits, and reviews open source rootkits.

2.1.1 Definition. The term rootkit describes a software kit that provides root access to a system, which is the highest-privileged role on a system [10]. While rootkits by themselves are thus not inherently malicious [3], they are often illicitly used as part of cyber attacks that allow adversaries to gain privileged access on a system without permission. After a rootkit has been deployed on a target system, it is often used to hide objects, such as files, processes, open ports, established connections, and the rootkit itself [3, 37], to evade detection and enable attackers with continuous privileged access to prepare further attack steps such as information gathering [39].

2.1.2 Methods. The capability to effectively hide system objects and themselves from manual inspection and automatic detection systems is essential for rootkits to avoid operators that become aware of the intrusion, try to remove the rootkits, or disconnect the system from the network. There are several methods by which rootkits interfere with systems, and it is common to differentiate them based on the layer where the rootkit resides: kernel-space, where all kernel activities are carried out with highest privileges, and user-space, which is a lower-privileged domain containing user applications and libraries [38]. One of the most straightforward methods used by user-space rootkits is to exchange system binaries such as "ls" with modified versions that skip certain elements. This method has become outdated nowadays as it is trivial to detect by checksums [5]. A modern alternative is to wrap functions in dynamically linked system libraries; in particular, many user-space rootkits exploit the "LD_PRELOAD" environment variable [38]. This variable instructs the dynamic linker to load a shared library before any other libraries when a program is executed, which allows custom and potentially malicious libraries to override functions in standard libraries.

Since user-space rootkits rely on the replacement of visible components such as binaries or libraries, their ability to hide from system-level inspection is limited. In contrast, kernel-space rootkits are generally considered to be significantly more difficult to detect since they have higher control over the system [42] and are thus better

[2] https://github.com/ait-aecid/caraxes.
[3] https://github.com/ait-aecid/rootkit-detection-ebpf-time-trace.

at evading detection by intrusion detection systems residing in user-space [28]. In this article, we therefore focus only on kernel-space rootkits. There are several methods by which kernel-space rootkits operate; we enumerate some of the most common ones in the following: (i) Loading the rootkit as a **Loaded Kernel Modules (LKM)**; (ii) Leveraging eBPF, which serves a similar purpose as LKM, but offers advantages such as higher stability and prebuilt hooks for system calls [38]; (iii) Loading the rootkit into the initial file system of an operating system, which is loaded before the actual root file system is mounted [22]; (iv) Kernel patching, i.e., adding rootkit functionality to kernel source code, exchanging the kernel image, and forcing a reboot; (v) Containerization, which creates a name space, starts the init system inside a container, and leaves the rootkit outside of that container where it is not visible [23]; (vi) Virtualization, which moves a running operating system into a hypervisor. As we show in the following, rootkits often use several of these methods in combination [33].

2.1.3 Open Source Kernel-Space Rootkits. We review common open source rootkits for the Linux operating system and analyze how they interact with the kernel. *Puszek*[4] is an LKM rootkit that first locates pointers to specific system calls in the system call table and then replaces the getdents system call to hide files. Instead of wrapping an entire system call, *Suterusu*[5] replaces a function within the getdents system call, named filldir. Specifically, it does so when filldir is passed as a function pointer in one of the arguments of the context actor. *Diamorphine*[6] and *Reveng_rtkit*[7] are LKM rootkits that make use of *kprobes*, a mechanism for debugging and tracing, to locate the system call table and wrap around the getdents system call. *Reptile*[8] is another LKM rootkit that uses khook[9] for binary patching of the filldir function; the mechanism behind this hacking tool is similar to kprobes. *Generic Linux Rootkit (GLRK)*[10] makes use of *ftrace*, a function tracer built into the Linux operating system, to execute code before and after function calls, which is equivalent to function hooking. However, the current implementation of the rootkit is designed for privilege escalation and does not support hiding; thus, no function is wrapped. Finally, *Boopkit*[11] is an eBPF rootkit that enables process hiding, remote activation, and command execution. Specifically, the rootkit uses probes at system calls to skip the names of predefined process IDs when listed. Boopkit is another example of a rootkit that wraps the getdents system call for hiding.

Table 1 summarizes important features of the reviewed rootkits, including the function wrapped by the rootkit for the purpose of hiding. Note that most rootkits manipulate multiple system calls to achieve various goals and that our review solely focuses on hiding capabilities. The table also specifies the kernel versions supported by each rootkit. Notably, none of the reviewed rootkits that employ function wrapping are compatible with modern Linux kernel versions, limiting their applicability for evaluations. To address this gap, we develop a new rootkit, which we introduce in Section 6.1.

2.2 Rootkit Detection

This section explains rootkit detection methods with a focus on learning-based detection approaches.

2.2.1 Methods. Given the diverse types of rootkits and the various ways in which they interact with systems, it stands to reason that many different methods have been developed for rootkit detection. In their recent study, Nadim et al. [28] divide existing methods into six classes: (i) Signature-based methods make use of a predefined list of static signatures such as byte patterns that correspond to known rootkits [3]. Despite their simplicity, these methods pose a highly robust form of rootkit detection with low false-positive rates, which is why most

[4] https://github.com/Eterna1/puszek-rootkit.
[5] https://github.com/mncoppola/suterusu.
[6] https://github.com/m0nad/Diamorphine.
[7] https://github.com/reveng007/reveng_rtkit.
[8] https://github.com/f0rb1dd3n/Reptile.
[9] https://github.com/milabs/khook.
[10] https://codeberg.org/sw1tchbl4d3/generic-linux-rootkit.
[11] https://github.com/krisnova/boopkit.

Table 1. Overview of the Analyzed Open Source Rootkits

Rootkit Name	Supported Kernels	Rootkit Method	Hooking Mechanism	Wrapped Function
Puszek	4.x	LKM	System call table	getdents
Suterusu	2.6–3.x	LKM	Function pointer in argument	filldir
Diamorphine	2.6–6.1	LKM	kprobes	getdents
Reveng_rtkit	5.11	LKM	kprobes	getdents
Reptile	3.10–5.x	LKM	Binary patching	filldir
GLRK	6+	LKM	ftrace	-
Boopkit	5.16+	eBPF	eBPF	getdents

well-known host-based intrusion detection systems primarily rely on signatures. Unfortunately, they are unable to detect unknown rootkits for which no signatures exist [3] and can also be evaded by slightly modifying existing rootkits as well as mutating rootkits [32]. (ii) Behavior-based detection aims to recognize actions of rootkits through anomalous states or behavior patterns observed in the operating system, such as unusual errors. (iii) Cross-view-based detection compares the visibility of the same objects on separate domains. Since rootkits often only hide objects on one domain, any divergences may indicate the presence of rootkits [32]. For example, different outcomes obtained from enumerating kernel modules in the user-space and searching loaded modules in memory may indicate a hidden rootkit. However, note that this method does not necessarily reveal the rootkit itself, but only some of its hidden objects [3]. (iv) Integrity-based detection recognizes changes to the kernel's static or dynamic data structures as opposed to recognizing the effects of such changes. In particular, changes of parts that are most often targeted by rootkits, such as patching of the system call table, are viable indicators for rootkit activity [3]. (v) Hardware-based detection leverages some of the other concepts mentioned in this enumeration but conducts analyses on an external device that cannot be accessed from the monitored host. (vi) Learning-based detection requires training data to capture a model that is subsequently used to classify unseen test data comprising data from both normal system behavior and rootkit activity. In the most common case, the training data only comprises normal instances and anomalies that are detected as instances that deviate from the normal behavior model. Given that learning-based detection is the most relevant detection concept for this article, we summarize existing works in this research area in the following.

2.2.2 Learning-Based Detection. As mentioned in the previous section, source code of rootkits may reveal their malicious nature, but signature-based approaches fail to classify unknown rootkits. In order to overcome this problem, researchers have trained neural networks on byte patterns of many malware samples to generate a model that enables classification of unseen samples. Raff et al. [30, 31] thereby present an important milestone by resolving the challenge of designing neural networks that are capable of processing large malware files with enormous byte pattern lengths. However, recent research has shown that these neural networks may be vulnerable to adversarial attacks [2]. Moreover, the source code of kernel-space rootkits is usually not accessible for inspection by conventional intrusion detection systems [28, 42]. As a consequence, researchers have considered to use dynamic features that are captured during execution of rootkits rather than static information such as their source code and byte patterns [12, 15].

Learning-based detection based on dynamic analysis methods requires collection of data that is affected by rootkit activity and available in sufficient volumes to enable model training. Some authors have therefore turned to **Hardware Performance Counters (HPC)**, which are special registers in microprocessors used for counting events [36]. Wang and Karri [40] compare the event counts from normal and rootkit samples. Singh et al. [37] identify relevant HPCs through experimentation with five synthetic rootkits. Sayadi et al. [35] analyze HPCs as time-series. These approaches generally rely on machine learning methods such as **Support Vector Machines**

(SVM), naive bayes classifiers, decision trees, and neural networks. Das et al. [13] point out some downsides of HPCs, in particular, the need for expert knowledge to understand and collect HPCs correctly, diversity of HPCs across different processors, non-determinism of counters, and overcounting. Pattee et al. [29] add that lack of documentation for HPCs, need for dedicated hardware, as well as energy constraints for machine learning in resource-limited devices pose issues for such detection approaches. Lu and Lysecky [25] point out that rootkits are able to mimic normal behavior patterns and recommend to focus on event timing, which is more difficult for them to replicate.

Several approaches therefore make use of event timing rather than count data for detection. Zimmer et al. [43] detect buffer overflow attacks in cyber-physical systems by measuring timing bounds at specific checkpoints. Similarly, Salem et al. [34] use inter-arrival curves to estimate lower and upper bounds for event occurrence times. Lu and Lysecky [25] feed event timings into a one-class SVM and find that this approach achieves better detection accuracy than range-based models that estimate limits of allowed time delays. Thereby, they split up these timings into fine-granular subcomponent timing models to reduce influences of the operating system on time measurements and increase model accuracy. Carreon et al. [7] point out that lumped timing models suffer from high variability, which limits their ability to detect anomalies. The authors thus also focus on subcomponent timings and use the boundaries of cumulative distribution functions of time measurements to assign anomaly scores to unseen samples.

Some timing-based approaches specifically focus on system calls from standard operating systems. For example, Ezeme et al. [18] propose a framework that captures the order of system calls as well as their relative duration measured in CPU cycle counts and predict the expected counts for detection. Luckett et al. [26] use neural networks to classify normal system and rootkit behavior based on system call timing. While the overall idea is similar to our work, they pursue supervised classification rather than detection and measure the runtime of entire system calls only. Another issue in their paper is that the method for data collection is not sufficiently described, which limits reproducibility of their results [27]. Dawson et al. [14] also capture isolated system call timings with strace; specifically, they test their detection approach using the system calls *open*, *close*, *read*, *futex*, *mmap2*, and *clock_gettime*. Brodbeck [4] measures system call latencies in mobile operating systems. Their findings suggest that rootkits can cause delays, but they do not analyze the complex distributions of system call timings in detail and also do not evaluate any detection algorithms. Contrary to these works, which capture the timing of entire system calls, our approach measures the inter-arrival timings of various functions within system calls, which allows us to isolate and reduce the effect of irrelevant factors such as the time needed to write and read from disk. Moreover, we are able to conduct our analyses on more fine-granular levels and capture even very slight time shifts, which can be of advantage when detecting rootkits that only modify these inner functions, such as Reptile (cf. Section 2.1.3). In addition, our detection method is designed for multimodal features that are prevalent in time measurements of system calls. We also investigate the influence of varying system conditions in detail. In the following, we outline the overall concept of our approach.

3 Concept

In this section, we first describe our threat model and then provide an overview of the detection approach proposed in this article.

3.1 Threat Model

Following the attacker-centric perspective recommended by Carlini et al. [6], we formulate our attack model based on the attacker's goals, knowledge, and capabilities.

3.1.1 Attacker's Goals. We assume that the attacker has previously gained root access to a compromised system through some unknown attack vector and has not been detected so far. The attacker's primary goal is to maintain stealthy and persistent access over the compromised system by hiding their own activities as well as

malicious artifacts (e.g., files or processes). The attacker achieves this by installing a kernel-space rootkit that manipulates system call functions, in particular within system calls responsible for directory enumeration. We emphasize that the detection of the original intrusion that enabled the attacker to access the system and deploy the rootkit is explicitly not targeted by the detection algorithm proposed in our article, which solely focuses on the detection of system interference from active rootkits.

3.1.2 Attacker's Knowledge. We assume that the attacker has detailed knowledge of the compromised system's operating system and is therefore able to install the kernel-space rootkit without triggering intrusion detection mechanisms that may be in place. We point out that our approach is designed to detect common rootkit activities; it is not designed to protect against adversaries with white-box access to our detection system that adapt their behavior to evade detection. We therefore assume that the attacker is not aware that the detection approach presented in this article is actively monitoring the system for changes of kernel function timings. Even if the attacker has knowledge about the existence of such timing-based detection approaches in general, we assume that they are not aware of the exact set of kernel functions monitored on this system.

3.1.3 Attacker's Capabilities. The attacker is capable of installing a kernel-space rootkit through some arbitrary attack vector that is not detected by any intrusion detection system present on the compromised system. Through that rootkit, the attacker is able to manipulate kernel function calls in such a way that certain system objects, e.g., specific files or processes, are effectively hidden from users and applications. After rootkit deployment, the attacker is able to ensure functional correctness of the modified system call so that neither a normal system user nor an application perceives any technical instability of the compromised system. We point out that any adversarial attacks on our detection system are considered out of scope, i.e., the attacker cannot observe or manipulate the time measurements continuously collected by our detection system, nor can they ascertain or reconstruct how these measurements are taken.

3.2 Overview of the Detection Approach

When kernel-space rootkits interfere with operating systems, they usually do so by wrapping around certain functions of system calls and thereby modifying the code executed at that position. As a consequence, the duration it takes to run the changed code is different in comparison to the runtime of the original code [7]. Specifically, the additionally executed code increases the overall runtime of the code block. The approach presented in this article hinges on the assumption that function delta times, i.e., time intervals between certain positions in the executed code, increase sufficiently to enable differentiation between normal system behavior and rootkit activities.

Figure 1 depicts an overview of our approach. Given that we aim to recognize rootkit activity through changed function delta times, we need to collect a baseline of measurements from an operating system that is free from rootkits and compare them to measurements from a system affected by rootkits. In both systems, we assume that normal user interactions take place continuously and that some of these executed commands trigger system calls affected by the rootkit, e.g., commands that enumerate files. As part of our approach, we inject probes into the kernel to capture execution times of inner functions of system calls. In particular, for a predefined set of functions that are likely to be affected by rootkit behavior, the probes collect and store the absolute timestamps of entering and returning from those functions as depicted in step (1) in Figure 1. We outline kernel tracing and probe injection in Section 4. Note that measurements are stored separately for the normal and rootkit case; in practice it is obviously not straightforward to make this differentiation and collect clean datasets; however, we argue that the detection of changed system call function timings in comparison to any past system state can already be useful indication for potential rootkit activity. We discuss practical and online applicability of our approach in Sections 6.4.3 and 7 in more detail.

Step (2) computes the delta times between pairs of timestamps collected in step (1). To this end, we consider two strategies: (i) computation of delta times between entry and return point of certain functions, and (ii) computation

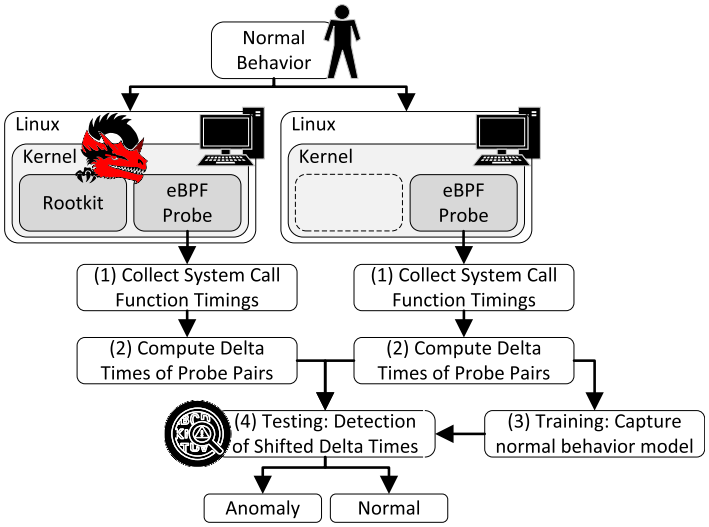

Fig. 1. Overview of our concept that measures system call function timings to detect rootkits through delta time shifts.

of delta times between two subsequently encountered probes independent of the respective function. We discuss each method in Section 5.1. In steps (3) and (4), we then apply machine learning to classify unseen delta times and detection of rootkits. Note that we pursue semi-supervised detection, meaning that a portion of only the normal delta times is used to generate a model of normal behavior [9]; all unseen samples that deviate too much from that model are detected as anomalies that are potentially caused by rootkit activities. Our method is suitable for offline detection, where data are forensically analyzed to differentiate between normal and rootkit samples, and online detection, where the model is incrementally updated and recognizes unusual changes of delta times on-the-fly. The following sections explain our methods for data collection and outlier detection in detail.

4 Kernel Tracing

In this section, we discuss the relevance of functions in system calls and describe our method to inject probes for time measurement.

4.1 Analysis of Relevant System Call Functions

Log data are commonly used for anomaly detection in the cyber security domain [21]. Unfortunately, rootkits have the highest privileges on a system and are thus able to manipulate the contents and generation of log data in such a way that their presence remains hidden, e.g., by suppressing certain log messages. Even though rootkits may alter the entire system at will, they cannot easily replicate the system behavior as if they were not present on the system, since any action that they perform still needs to be executed by the kernel, which is where they leave detectable traces.

One way to monitor kernel activities is to analyze system calls, which are an interface for user programs to request resources and interact with the kernel. Most operating systems offer hundreds of system calls,[12] with some of the most common ones being open, read, write, and fork. Some modern host-based intrusion detection systems are capable of monitoring single invocations of system calls and sequences of system calls have long been used for malware detection [19]; however, rootkits do not necessarily affect multiple system calls, but may

[12] https://man7.org/linux/man-pages/man2/syscalls.2.html.

```
1  |            | x64_sys_call() {
2  |            |     __x64_sys_getdents64() {
3  |            |         __fdget_pos() {
4  |   0.476 us |             __fget_light();
5  |            |             mutex_lock();
6  |   3.255 us |         }
7  |            |         iterate_dir() {
8  |            |             security_file_permission();
9  |            |             down_read_killable();
10 |            |             dcache_readdir() {
11 |            |                 filldir64() {
12 |   0.699 us |                     verify_dirent_name();
13 |   1.727 us |                 }
14 |   0.458 us |                 _raw_spin_lock();
15 |   0.466 us |                 _raw_spin_unlock();
16 |            |                 filldir64() {
17 |   0.544 us |                     verify_dirent_name();
18 |   1.469 us |                 }
19 |            |                 scan_positives() {
20 |   0.462 us |                     _raw_spin_lock();
21 |   0.461 us |                     _raw_spin_unlock();
22 |   0.469 us |                     dput();
23 |   4.787 us |                 }
24 |            |                 /** loop **/
25 |   0.473 us |                 dput();
26 | 118.077 us |             }
27 |            |             touch_atime() {
28 |            |                 atime_needs_update() {
29 |   0.464 us |                     make_vfsuid();
30 |   0.466 us |                     make_vfsgid();
31 |            |                     current_time();
32 |   4.233 us |                 }
33 |   5.222 us |             }
34 |   0.498 us |             up_read();
35 | 132.216 us |         }
36 |            |         __f_unlock_pos();
37 | 138.769 us |     }
38 | 139.968 us | }
```

Fig. 2. Excerpt from the getdents call stack and function timings collected with the function tracer (ftrace).

only affect a single system call or just one of its inner functions. As a consequence, existing intrusion detection systems do not monitor kernel activity in sufficient granularity to uncover rootkit activities.

Due to the fact that many system calls exist and each of them involves a myriad of functions, making a reasonable selection for monitoring is vital to limit the amount of data to be analyzed and focus on those points that are most likely targeted by rootkits. Rather than manually sifting through all available system calls to make this decision, we reviewed seven open source rootkits (cf. Section 2.1) and found that every single one of them makes use of the *getdents* system call or one of its inner functions to enable hiding of files. We also noticed that getdents is explicitly mentioned in technical reports on cyber attacks involving rootkits [20]. This is intuitively reasonable since getdents is the only interface for a user program to list the contents of a directory in Linux, which is an interface that rootkits need to control in order to hide objects, such as files or themselves [4]. We display a shortened version of the getdents system call in Figure 2, which we generated with the function tracer[13] (ftrace). Note that we omit irrelevant functions from the code for brevity and that executed functions may differ depending on the context in which getdents is invoked. As visible in the figure, the function tracer also provides time measurements that describe how long it took to complete single functions. These timings are the primary data source for our detection algorithm. In order to collect time measurements in a structured way, we inject probes into the kernel, which we describe in the following section.

[13] https://www.kernel.org/doc/html/latest/trace/ftrace.html.

```
1  BPF_RINGBUF_OUTPUT(buffer, 1 << 4);
2  struct event {
3      unsigned long time;
4      u32 pid;
5      u32 tgid;
6  };
7  int probe(struct pt_regs *ctx) {
8      struct event *event = buffer.ringbuf_reserve(sizeof(struct event));
9      if (!event) {return 1;}
10     u64 pid_tgid = bpf_get_current_pid_tgid();
11     event->tgid = pid_tgid >> 32;
12     event->pid = (u32) pid_tgid;
13     event->time = bpf_ktime_get_ns();
14     buffer.ringbuf_submit(event, 0);
15     return 0;
16 }
```

Fig. 3. Implementation of our eBPF probe.

4.2 Injection of eBPF Probes

While the function tracer is a viable method to obtain function time measurements, we opt for the more modern eBPF to implement time-measuring probes. The eBPF is a Linux kernel technology that enables developers to build programs that run securely in kernel-space. Figure 3 shows our implementation of an eBPF probe. Each probe first obtains identifiers for the current process (pid) and thread group (tgid) in Line 10, which are stored in the respective variables in Lines 11 and 12. Then, the current time is measured in nanoseconds in Line 13. Finally, these variables are written as an event in the ring buffer (Line 14), which is created in Line 1 and managed by the **BPF Compiler Collection**[14] **(BCC)** that is also used to inject the probes.

BCC supports injection at enter or return points of all functions that can be traced by eBPF. For each probe, we therefore obtain two measurements that we refer to as probe-enter and probe-return, respectively, where we use the name of the function to refer to the probe. For example, for function filldir in Figure 2, we obtain time measurements from probes filldir64-enter (Lines 11 and 16) and filldir64-return (Lines 13 and 18). To avoid that our script collecting the events from BCC influences time measurement by triggering system calls when storing the data, all events are held in memory until all probes are unloaded. Note that some functions cannot be traced by eBPF and are thus neglected for our analyses.

5 Detection

This section outlines our detection algorithm. We explain two strategies to derive delta times from time measurements and then describe an approach to detect anomalies based on shifted delta times.

5.1 Computation of Delta Times

The main idea behind our detection approach is that it takes more time to execute code of functions that are wrapped by the rootkit in comparison to executing the original functions without any additions made by the rootkit [7]. To capture the duration of time intervals between any two probes $p1$ and $p2$ based on the absolute time measurements collected as described in the previous section, it is necessary to subtract the timestamp of an event observed at probe p_2 with the timestamp of the event observed at probe $p1$ that chronologically occurs before. In the following, we use the colon to denote this pairing of probes, i.e., p_1:p_2. For example, filldir64-enter:verify_dirent_name-enter are delta times between the probes at filldir64-enter (Lines 11 and 16 in Figure 2) and verify_dirent_name-enter (Lines 12 and 17). Note that processes are often running in parallel and their workflows interleave, causing that chronological sorting alone is not sufficient to correctly pair probes. We

[14] https://github.com/iovisor/bcc.

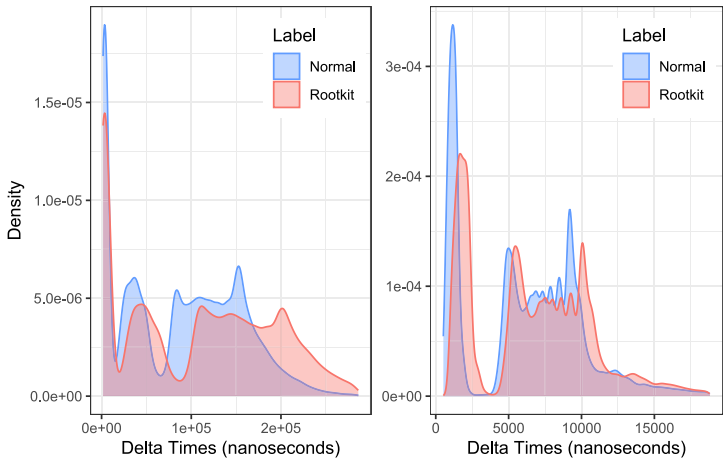

Fig. 4. Time measurements at probes iterate_dir-enter:iterate_dir-return (left) and filldir64-return:filldir64-enter (right) depicting longer delta times when the rootkit is active (red) in comparison to normal system behavior (blue).

therefore first split all measurements into groups by process identifier (pid) and then sort all timestamps in each group in ascending order before subtraction.

To reduce the immense number of possible combinations of probes to a manageable amount, we propose two strategies for probe pair selection. The first strategy only considers enter and return probes injected at the same function and subtracts the most recent timestamp of a probe-enter event from the probe-return event, effectively measuring the time it takes to execute that function. Accordingly, we refer to this strategy as function-grouping in the following. Note that the described procedure is repeated for each event from every probe. The second strategy sequentially iterates through all events in chronological order and subtracts the timestamp from the latter event with the timestamp from the former event, independent of the probe and its corresponding function. We refer to this strategy as sequence-grouping. Each strategy has its own benefits. While sequence-grouping captures short delta times between adjacent functions, which enables fine-granular analysis, function-grouping captures entire functions and may thus be better suited to detect rootkits that wrap around these functions.

Figure 4 depicts two illustrative distributions of delta times, where the left one corresponds to probes iterate_dir-enter:iterate_dir-return using function-grouping and the right one corresponds to filldir64-return:filldir64-enter using sequence-grouping. We select these samples, because they both show a visible delay of delta times when the rootkit is active in comparison to delta times collected during normal system operation. In the following section, we describe a mechanism that automatically detects these shifts.

5.2 Shift Detection

The key feature of our detection approach is to automatically recognize shifts in delta times that are computed as described in the previous section. However, the density curves plotted in Figure 4 reveal that even though there is a visually apparent shift in the distributions of delta times, it is non-trivial to measure that shift and determine what degree of shift is still tolerable and likely a product of natural variation rather than caused by the rootkit. In particular, delta times from some probes may be influenced by system conditions and thus too volatile for use as a baseline. To be able to assess the amount of expected variation, we therefore assume to have several batches of data at our disposal, where each batch was taken independently over a certain period of time and contains sufficiently many delta times to estimate their distribution at that point in time. This allows us to apply statistical tests on the data despite varying system conditions and noise. In the following, we refer to batches

collected during normal system operation and batches collected while a rootkit is active on the system as normal and rootkit batches, respectively.

Due to the fact that the distributions of delta times are multimodal (i.e., involve multiple local peaks) and contain outliers, it is not feasible to simply rely on the mean and standard deviation for statistical testing [26]. To overcome this issue, we compare the quantiles of delta time distributions from different batches, because quantiles accumulate close to peaks where robust measurements of delta time shifts are possible. For example, a natural way of determining the shift for the distributions in Figure 4 is to measure the horizontal offset at some prominent peaks. Moreover, quantiles enable to determine whether only a portion of the delta times has been shifted, which can be relevant if only some invocations of the same function are affected by the rootkit.

We use equidistant spacing between 0 and 1 for a predefined number of quantiles q to capture most of the entire distribution while at the same time ensuring robustness against outliers. For example, for the trivial case of $q = 1$, the median (or 0.5-quantile) will be used, while for $q = 4$, the 0.2-, 0.4-, 0.6-, and 0.8-quantiles will be used. We combine the quantiles of delta times for a set of normal batches in a training set $V_{p_1:p_2}$ of size $n \times q$, where n is the number of training batches and q is the number of quantiles. For an unseen batch of delta time quantiles x, which can be from the remaining normal batches or one of the rootkit batches, we compute the squared Mahalanobis distance using Equation (1), where $\mu(V_{p_1:p_2})$ is the mean of delta times for each quantile in $V_{p_1:p_2}$ and Σ^{-1} is the inverse covariance matrix of $V_{p_1:p_2}$. We then compute the p-value for a specific $p_1{:}p_2$ using a χ^2-test with q degrees of freedom as stated in Equation (2), where cdf is the cumulative distribution function [17].

$$D^2_{p_1:p_2} = \left(x_{p_1:p_2} - \mu(V_{p_1:p_2})\right)^\top \Sigma^{-1} \left(x_{p_1:p_2} - \mu(V_{p_1:p_2})\right), \tag{1}$$

$$\text{p-value}_{p_1:p_2} = 1 - \chi^2(cdf(D^2_{p_1:p_2}, q)). \tag{2}$$

Low p-values close to 0 indicate that delta times at one or multiple quantiles are significantly shifted from the expected means considering the variations observed in the training data, while p-values close to 1 indicate the opposite. Note that neither the distribution of delta times nor their quantiles are assumed to follow a normal distribution. Instead, we only assume that the values of the same quantile (e.g., the 0.5-quantile) of multiple batches are normally distributed, which is a fair assumption for the purpose of shift detection in delta time distributions that are otherwise similar. Changes of delta time distributions in test batches other than shifts may also reflect in quantile differences; however, we do not consider this problematic as these changes also indicate anomalous behavior that should be detected. We also emphasize that our proposed method is semi-supervised, because we assume that our training set only contains normal data and is free from anomalies introduced by rootkits, while test data may contain batches from both classes.

The aforementioned computation of p-values is applicable to a single combination of probes $p_1{:}p_2$, independent of the grouping strategy used. However, many functions within the getdents system call could be targeted by rootkits, and it is not possible to know beforehand on which combination of probes to focus on. We therefore propose to involve as many probes as possible for shift detection and combine their respective p-values. Given that rootkits can wrap any function, i.e., only affect delta times collected from a single probe, we consider it sufficient if one of the p-values is below a certain threshold θ to detect the entire batch as an outlier that potentially indicates rootkit activity. The sample is only considered normal if all p-values are above the threshold. In the following section, we evaluate the effectiveness of this detection method by deploying a rootkit on a real kernel and analyzing the delta times.

6 Evaluation

This section covers the evaluation of our work. We introduce a novel open source rootkit and describe the generated public datasets, which we use to evaluate our detection approach.

```
1 static bool hook_filldir64(struct dir_context *ctx, const char *name, int namlen,
      loff_t offset, u64 ino, unsigned int d_type) {
2     struct readdir_callback *buf = container_of(ctx, struct readdir_callback, ctx);
3     if (strstr(name, MAGIC_WORD)) {
4         buf->result = -ENOENT;
5         return false;
6     }
7     return orig_filldir64(ctx, name, namlen, offset, ino, d_type);
8 }
```

Fig. 5. Wrapper for the filldir function.

6.1 CARAXES: A Cyber Analytics Rootkit

We initially planned to select one of the open source rootkits reviewed as part of Section 2.1 for our evaluation; however, we realized that none of these rootkits are able to run on modern kernels. For example, Diamorphine overwrites the system call table to perform system call wrapping, which is not possible since Linux kernel version 6.9 started removing the system call table as a security measure to avoid speculative execution. Reptile wraps around the filldir function rather than an entire system call, but this function has been changed in Linux kernel version 6.1. In addition, Reptile relies on the function "kallsyms_lookup_name," which is not available in Linux kernels above version 6. Similar issues exist for other rootkits. An exception to this observation is GLRK, which works without issues but unfortunately does not support file hiding.

To overcome this problem, we present a **Cyber Analytics Rootkit for Automated and eXploratory Evaluation Scenarios (CARAXES)**. We base our implementation on GLRK, which we extend with functionality to hide files if they contain specific keywords in their names or belong to a certain user or group. Process hiding is implicitly supported by specifying user and group identifiers of processes to be hidden. We implement two different ways of hooking into the kernel: wrapping the getdents system call (entire code of Figure 2) and wrapping the filldir function (Lines 11 and 16 in Figure 2) within that system call. We noticed during our experiments that our probing mechanism (cf. Section 4.2) prevents the rootkit from wrapping getdents; for this reason, we focus on filldir wrapping for our evaluation. Figure 5 displays the filldir wrapper, which checks in Line 3 if the name of an enumerated file contains the keyword *MAGIC_WORD* and skips the file in case of a match, rendering it invisible. Otherwise, the wrapper passes all parameters to the original filldir function in Line 7 and returns its result.

CARAXES is designed to be compatible with modern Linux kernel versions and has been tested on version 6.11. Its implementation and functionality are based on existing open source rootkits and are not tailored to favor our detection approach. Instead, CARAXES is intended to be generally applicable for a wide range of experimental scenarios. We make CARAXES publicly available as open source to allow others to generate new datasets and extend the rootkit with additional features (cf. Section 7).

6.2 Data Generation

This section outlines our procedure for data generation and explains which scenarios we consider for system variations.

6.2.1 Procedure. To generate the datasets that we use to evaluate our probing framework and detection approach, we set up a data generation procedure involving the rootkit described in the previous section. Figure 6 visually summarizes our setup, which consists of the following steps: (1) We create a directory that contains two files; one arbitrary file and one that should be hidden by the rootkit. We identify the file to be hidden through its name, i.e., the rootkit is programmed to hide all files containing the keyword "caraxes" in their names. (2) We inject eBPF probes at specific functions using our framework from Section 4. An obvious choice for probing is filldir because it is the function that is wrapped by the rootkit. However, we point out that the function is also an interesting choice as it is called twice in close succession (Lines 11 and 16), which provides relevant measurements for sequence-grouping. We additionally select the following three functions for probing due to their distinct

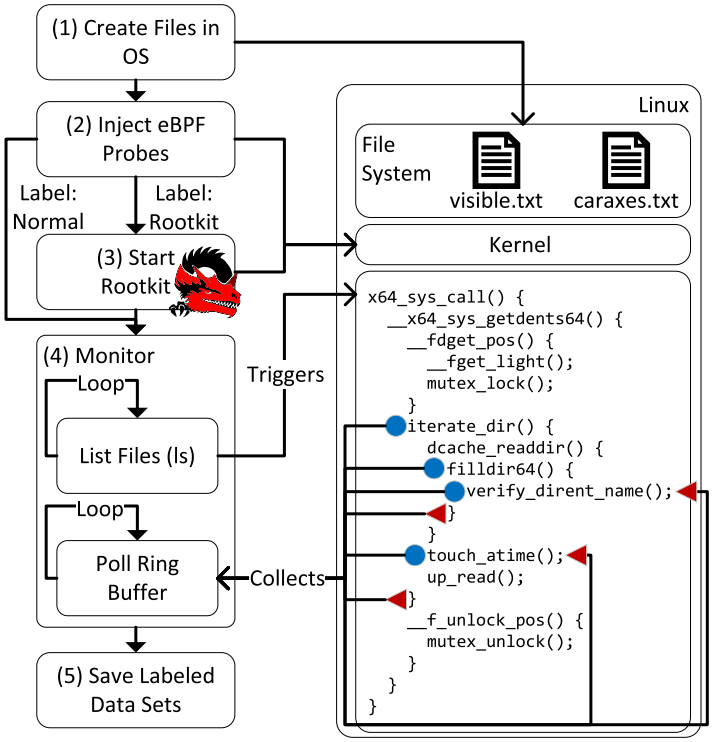

Fig. 6. Overview of our data generation procedure.

relations to the wrapped filldir function: iterate_dir (Line 7 in Figure 2), because it is the enclosing function of filldir and should be directly affected by time delays of its inner functions, verify_dirent_name (Line12), because it is an inner function of filldir, and touch_atime (Line 27), because of its proximity to the affected filldir function. Note that we originally selected dcache_readdir (Line 10) instead of iterate_dir as it is the direct enclosing function, but our tracer was unable to inject the probes at this position (cf. Section 4.2). (3) We then either activate the rootkit or proceed without an active rootkit; this decision determines the label of the resulting dataset. (4) We then start a loop that repeatedly executes the "ls" command to list files in the previously generated directory containing the two files, which performs getdents system calls that involve the wrapped function. The command is executed 100 times without any waiting time in between. In parallel, we run another loop that continuously polls the ring buffer to collect time measurements from probes. (5) After completion, the measurement times are written from memory to the file system, including meta-information such as labels and parameters of the generation procedure.

We refer to the dataset resulting from this procedure as one batch of data. We run this procedure many times to collect a sufficiently large amount of normal and rootkit batches to train and test our approach. Between each batch, we pause the iteration for 10 seconds using the sleep command in order to ensure that there are no artifacts from previous batches introduced during data collection. Since each iteration takes around 5–10 seconds to complete, their start times are around 15–20 seconds apart. Consequently, the multi-iteration experiment takes several hours, enabling analysis of trends or concept drift in the data as time progresses.

6.2.2 Scenarios. In addition to noise, trends, and concept drift originating from the system, parameters of the evaluation setup need to be taken into consideration when analyzing the data. To investigate the influence of

Table 2. Overview of the Generated Datasets

Scenario	Label	Start time	End time	Batches	#Events (median)
Default	Normal	10:58:25	11:44:24	150	39,538
	Rootkit	11:44:44	12:17:18	100	52,524
File Count	Normal	12:17:37	13:05:46	150	78,016
	Rootkit	13:06:06	13:40:04	100	89,368
System Load	Normal	13:40:24	14:29:06	150	39,274
	Rootkit	14:29:27	15:03:56	100	52,115
ls-basic	Normal	15:04:14	15:49:45	150	33,230
	Rootkit	15:50:04	16:22:15	100	39,818
Filename Length	Normal	16:22:33	17:08:35	150	39,544
	Rootkit	17:08:55	17:41:30	100	52,120

these factors, we carry out the generation of normal and rootkit batches in five different scenarios: (i) *Default*. The procedure is executed as described in Section 6.2.1. (ii) *File Count*. Other than in the default scenario where one normal file and another file to be hidden are generated, this scenario involves a random selection of 10–100 files of each type. (iii) *Filename Length*. In contrast to the default scenario where names of files consist of at most eight random characters (and the keyword "caraxes" for files to be hidden), the lengths of file names in this scenario are randomly selected in the range of 20–60 characters. (iv) *ls-basic*. We replace the "ls" command that is used to enumerate files with a custom implementation named "ls-basic"[15] that does not rely on any libraries and allows us to know exactly which system calls are invoked. (v) *System Load*. We run the tool stress-ng[16] in background while collecting the data to simulate a system under load. The datasets are generated on Ubuntu 22.10 with Linux Kernel 5.19, 32GB RAM, and 8 vCPUs.

6.3 Datasets

This section provides an overview of three datasets published alongside this article: one dataset of time measurements at probes and two datasets of delta times derived from these time measurements.

6.3.1 Time Measurements at Probes. Table 2 summarizes the first dataset of probe measurements collected as described in Section 4.2. The table differentiates between normal and rootkit batches as well as the scenario in which they are collected (cf. Section 6.2.2). In addition, we also provide the start and end times of collection in the table, which shows that we collected the data successively by iterating through each scenario and alternating between normal and rootkit cases. For each scenario, we collect 150 normal and 100 rootkit batches. The reason for this is that we intend to use 50 batches (a third of the normal data) for training and thereby leave a balanced test set of 100 normal and 100 rootkit batches, respectively. The last column of the table states the median number of events per batch, which shows that event counts vary across scenarios and that rootkit batches involve more events compared to normal batches of the same scenario.

Figure 7 provides a more detailed view of the number of events per batch by additionally separating the counts by probe. The figure reveals several interesting aspects. First, the number of time measurements varies across probes. For example, the iterate_dir function is invoked less frequently than other probed functions, causing that fewer timestamps are collected from the probes of that function. This is simply explained by the program workflow that calls some functions more often than others, e.g., filldir is invoked multiple times within the iterate_dir function (cf. Figure 2). Second, the enter and return probes of each function yield roughly the same number of measurements. While there are some outliers indicating that few measurements have not been

[15] https://github.com/ait-aecid/rootkit-detection-ebpf-time-trace/blob/main/ls-basic.c.
[16] https://github.com/ColinIanKing/stress-ng.

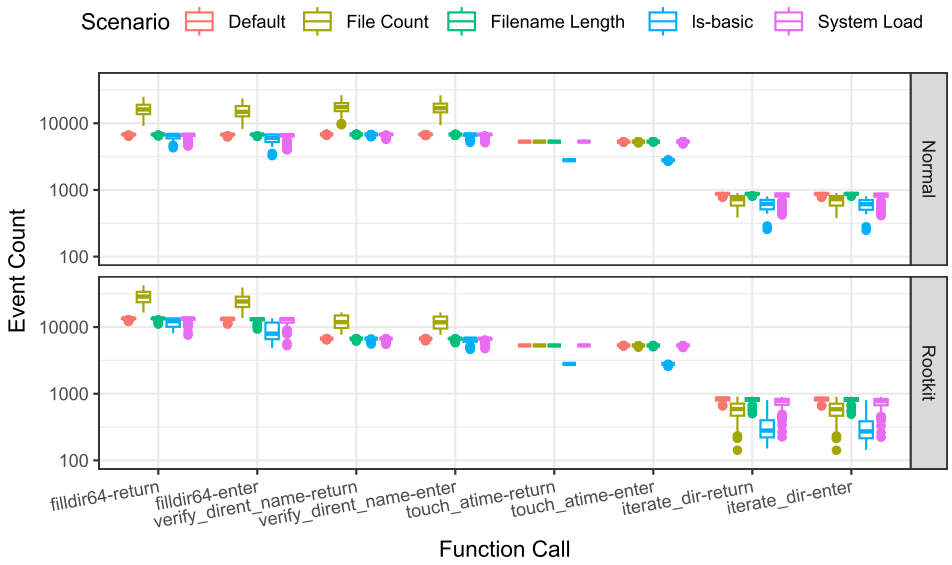

Fig. 7. Number of events per probe across scenarios.

successfully retrieved from the ring buffer, the overall distributions suggest that most events have been collected and all probes have been adequately captured. Third, event counts collected at probes of different functions vary across scenarios. For example, increasing the number of files triggers more invocations of the filldir and verify_dirent_name functions. Fourth, there are some differences in the number of events depending on whether a rootkit is active or not, which explains the event counts stated in Table 2. The most significant divergence occurs with the filldir function, which is reasonable since the rootkit injects a new function at the same position that also invokes the original function, which increases the total number of collected measurements at that probe.

The latter observation implies that in our dataset the number of event counts can be used as a trivial way of detecting rootkits. However, this is not necessarily the case in general since rootkits may be injected into the kernel in many different ways (cf. Section 2.1.2); some of which do not trigger an additional function call. For example, rootkits may substitute the targeted function entirely by replacing the kernel with a modified version and forcing a reboot. Analysis of function timing is also capable of detecting changes within the code and is not limited to function calls. In addition, while detection based on event counts fails if the specific function targeted by the rootkit is not probed, timing-based detection is still able to recognize delays as long as the affected function is invoked within the probed function, causing the entire runtime of the outer function to increase. Finally, function timing is non-trivial to replicate for rootkits that try to mimic normal behavior in comparison to other features collected from the kernel [25]. In the following, we therefore only focus on function timing for detection.

6.3.2 Delta Times. Based on the dataset of time measurements described in the previous section, we compute two datasets of delta times using function-grouping and sequence-grouping following the strategies outlined in Section 5.1. To visualize the delta times, we compute the median value of delta times for each pair of probes in every batch and apply **Principal Component Analysis (PCA)**. Figure 8 shows biplots of the first two principal components for delta times computed using function-grouping (left) and sequence-grouping (right), where symbols indicate the collection scenario (cf. Section 6.2.2) and color differentiates normal (blue) from rootkit (red) batches. It is apparent in the left plot that many batches corresponding to rootkit behavior exhibit increased delta times at probes iterate_dir-enter:iterate_dir-return, which is also the combination of probes we depict for the default scenario in the left side of Figure 4. While this suggests that many of these batches can

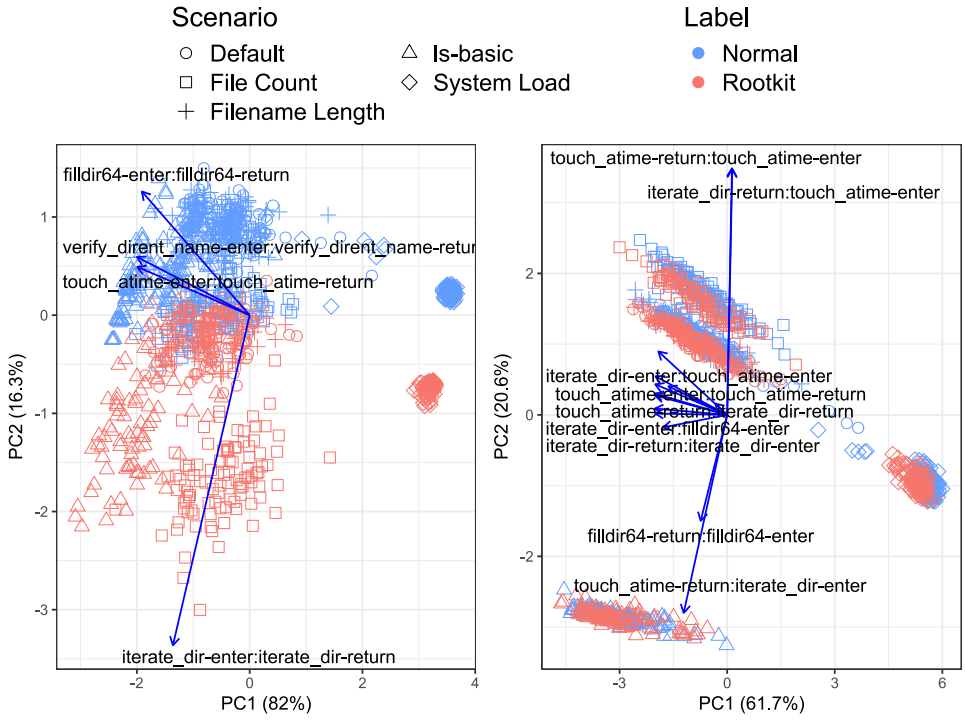

Fig. 8. Biplots of delta times computed via function-grouping (left) and sequence-grouping (right).

be correctly detected as outliers, it seems difficult to discern batches corresponding to different scenarios, with the exception of the system load scenario where batches are far away from the others and both classes are well separable. The plot on the right-hand side suggests that sequence-grouping performs better at separating batches from different scenarios, with the exception of batches corresponding to the scenario where filename lengths are varied that mostly overlap with batches from the default scenario. Contrary to the plot on the left-hand side, however, rootkit batches are not as simple to separate from normal batches. In the following, we evaluate our detection approach with both strategies and compare the results.

6.4 Results

This section presents the results of our evaluation. We first highlight that delta times are shifted at certain probes, which indicates rootkit activity. We then apply our detection algorithm and evaluate its ability to detect rootkit from normal activity given a training set of only normal data.

6.4.1 Delta Time Shifts. While the biplots presented in the previous section provide a rough indication about the probes that are suitable to detect rootkits based on function timing, they do not communicate precise delays and combine the influence of several probes. Moreover, while we only use the median for PCA, we aim to involve multiple quantiles for detection (cf. Section 5.2). Inspired by the shift function that has been used to compare which parts of distributions are shifted [16, 41], we subtract each quantile of delta times collected from rootkit batches with those of normal batches. In the following, we always use nine quantiles ($q = 9$) for our analyses and detection evaluations.

Figure 9 depicts the differences of delta times for function-grouping as a boxplot, separated by probes and scenarios. As visible in the plot, only the iterate_dir-enter:iterate_dir-return probes show a significant shift across

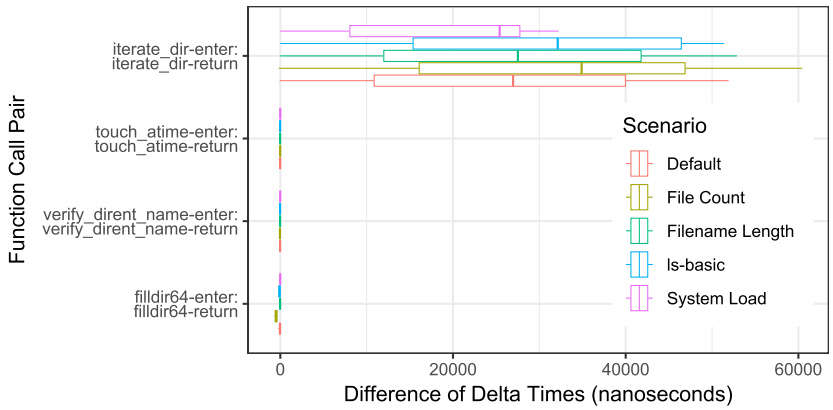

Fig. 9. Delta time shifts for function-grouping.

all scenarios; other probes seem mostly unaffected by the influence of the rootkit. While the shifts spread over a wide range, these results suggest that most of the batches collected during rootkit activity can be detected. In contrast, Figure 10, which depicts the delta times computed with sequence-grouping, shows significantly more diverse patterns. While there is hardly any shift noticeable at some probes in the middle of the plot and some probes in the bottom of the plot show no obvious trends, the probes depicted in the top of the plot clearly indicate shifted delta times. While some of them exhibit a rather high variation for certain scenarios, we find that iterate_dir-enter:filldir64-enter and filldir64-return:filldir64-enter are good indicators as the shifts seem relatively constant and have little variation across quantiles, batches, and scenarios. For this reason, we select filldir64-return:filldir64-enter as an illustrative example to display time shifts on the right side of Figure 4. In the following, however, we do not make any manual selections, but instead use the entirety of the data to leverage as much information as possible and facilitate realistic evaluation.

6.4.2 Offline Detection. To evaluate our detection algorithm described in Section 5.2, we split the batches of each of our two datasets of delta times into training and test datasets. As proposed in Section 6.3.1, we use delta times of 50 normal batches for training and leave 100 remaining normal and 100 rootkit batches for testing. We thereby pursue evaluation in an offline setting, i.e., sample the training data randomly from the normal batches and repeat the sampling 100 times so that we are able to estimate the variance of our results. Note that we evaluate the detection performance separately for every scenario, i.e., we sample the training data only for normal data from one specific scenario and then evaluate detection using the test data of that scenario, and repeat that for every scenario independently. The reason for this is that we do not assume that our normal behavior model generated from data of one scenario is capable of differentiating normal and rootkit batches from another scenario. We thus count true positives (TP) as rootkit batches that are detected as anomalous by our approach, false positives (FP) as normal batches that are detected as anomalous, false negatives (FN) as rootkit batches that are not detected as anomalous, and true negatives (TN) as normal batches that are not detected as anomalous. We sum up all of these counts for every scenario and compute the true-positive rate or recall ($TPR = Rec = \frac{TP}{TP+FN}$), true-negative rate ($TNR = \frac{TN}{TN+FP}$), precision ($Prec = \frac{TP}{TP+FP}$), accuracy ($Acc = \frac{TP+TN}{TP+TN+FP+FN}$), and F1 score ($F1 = \frac{2 \cdot Prec \cdot Rec}{Prec+Rec}$).

We fine-tune the detection threshold θ to maximize the F1 score, but notice that most p-values of rootkit batches are truncated to 0, meaning that best results are achieved when the threshold is set to a very small but non-zero value, such as 10^{-10}. The left side of Figure 11 shows the confusion matrix for our detection results, where we state the relative number of batches that end up in each cell. Note that due to the fact that each scenario

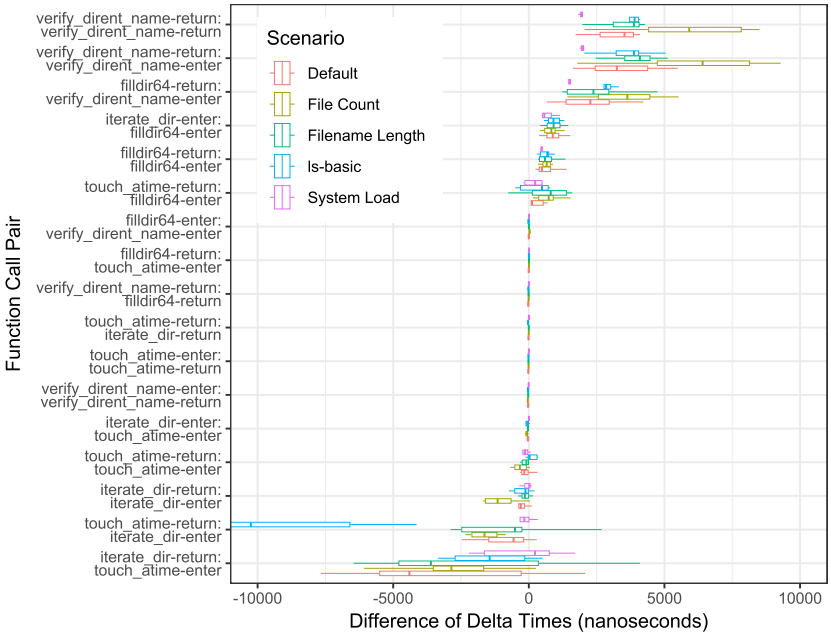

Fig. 10. Delta time shifts for sequence-grouping.

is evaluated separately and the prediction only differentiates between the normal and rootkit class but not the scenario, it is necessary to consider the values for each scenario in the columns on its own. For example, the top left corner of the matrix shows that all rootkit batches and 98.4% of all normal batches from the default scenario have been correctly detected as such, while 1.7% of the normal batches have been incorrectly detected as anomalous and thus contribute to the false-positive counts. For all other blocks in that column only the testing data have been changed, e.g., the block directly below shows that all normal and rootkit batches of the file count scenario are detected as anomalous. We leave these values in the confusion matrix for information, but emphasize that they do not contribute to the metric counts. Interestingly, the default scenario seems to be suitable as training data for the filename length scenario and vice versa. This confirms that filename lengths have virtually no impact on delta times, which was already indicated by the right plot in Figure 8. Overall, function-grouping seems to be suitable to correctly classify most of the batches. In comparison to that, the confusion matrix in the right side of Figure 11 suggests that sequence-grouping is significantly more prone to false positives across all scenarios.

We compare the detection performance of our approach with two state-of-the-art methods for rootkit detection based on event timings: One-class SVM [25] and **Artificial Neural Networks (ANN)** [26]. Due to the absence of publicly available code in the original publications, we re-implement the approaches with some modifications to make them suitable for our data and use-case. For the SVM, we construct features from each pair of probes, using the median delta times as feature values, as illustrated in Figure 8. We then train a one-class SVM with a Radial Basis Function kernel on normal training data and use the resulting model to compute anomaly scores on the test data. These scores are optimized analogously to the p-values in our shift-based detection approach. For the ANN, we follow the reasoning of Luckett et al. [26] and feed the delta times into the neural network without prior quantile transformation as it should be able to handle complex data distributions. Contrary to their approach, however, we pursue semi-supervised detection rather than supervised classification. Therefore, we randomly sample half of the training data to train the model and subsequently use the remaining half to estimate center

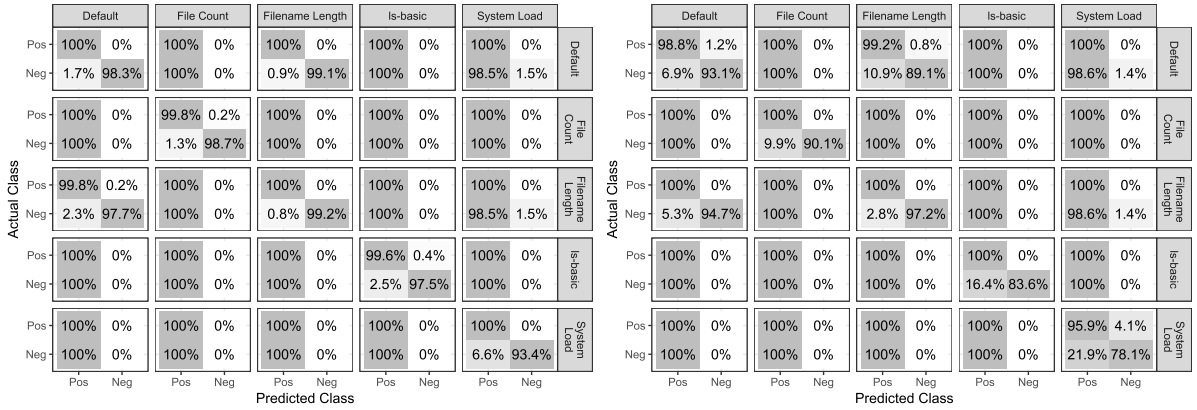

Fig. 11. Confusion matrices for function-grouping (left) and sequence-grouping (right).

Fig. 12. Comparison of evaluation results from detection approaches leveraging delta time shifts, Artificial Neural Networks (ANN), and Support Vector Machines (SVM).

and spread of the normal model. This allows us to compute anomaly scores for the test data and optimize our detection threshold. We refer to our open source repository for details on the implementations (cf. Section 1).

Figure 12 visualizes the evaluation metrics for all aforementioned approaches. For the case of function-grouping, the detection approach based on delta time shifts proposed in this article achieves a median F1 score of 98.7% and thus outperforms the approaches based on SVM ($F1 = 97.1\%$) and ANN ($F1 = 91.6\%$). It is thereby noteworthy that detection based on delta time shifts yields perfect true-positive rates ($TPR = 100\%$) for most iterations while maintaining high true-negative rates ($TNR = 97.6\%$). Considering the evaluation results for sequence-grouping, the plot shows that detection based on delta time shifts ($F1 = 94.6\%$) is roughly on par with SVM ($F1 = 95.0\%$), while ANN ($F1 = 76.3\%$) is once more unable to keep up with the other two approaches. Overall, all approaches evaluated on the sequence-grouping case yield lower detection metrics in comparison to function-grouping, specifically due to low true-negative rates (TNR) caused by many false positives. The main reason for this is that normal and rootkit batches are more difficult to discern in the sequence-grouping case, as shown in Section 6.3.2.

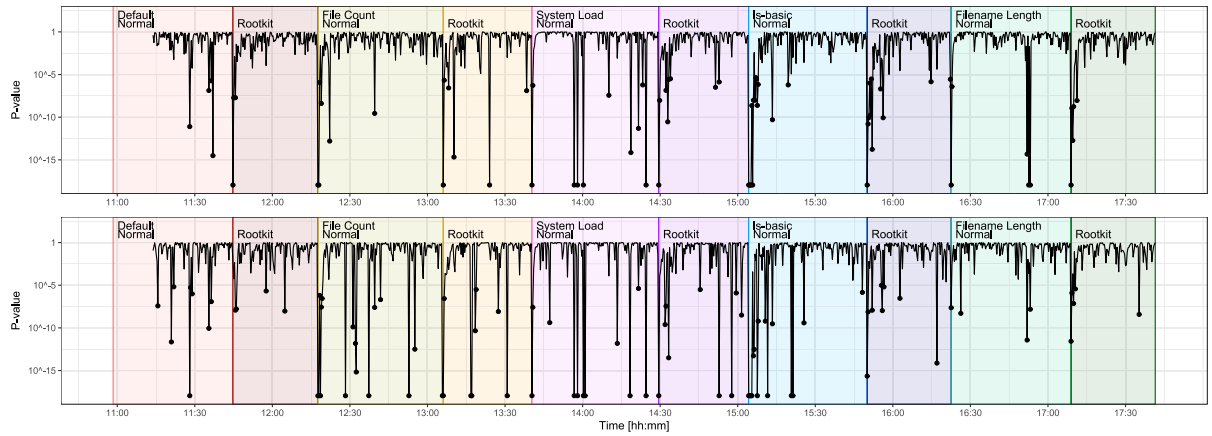

Fig. 13. Online detection of function-grouping (top) and sequence-grouping (bottom).

6.4.3 Online Detection. In realistic settings, it is usually non-trivial to ensure that training sets are free of anomalies, because manual and forensic investigation of data for rootkit traces (or the absence thereof) is a time-intensive task that requires specific domain knowledge. Moreover, it is usually not possible to assume that batches of the same class and scenario are identically distributed over time, since real systems are affected by concept drift and more recently collected batches usually represent the current system behavior better than ones that have been collected some time ago. Accordingly, while the methodology presented in the previous section is suitable to measure and compare detection capabilities for evaluation, real applications often require online detection that enables incremental processing of batches.

In addition to offline detection, i.e., random selection of training data from normal batches as described in the previous section, we conduct an experiment for online detection where data are processed chronologically and only the most recent batches are used for training. To this end, we run a sliding window of size $w = 50$ over the chronologically sorted batches and use them to predict whether the batch following just after that sequence of batches corresponds to normal or rootkit behavior. We thereby leave the order of the batches unchanged from the sorting displayed in Table 2, meaning that batches from each scenario are processed one after another and normal batches of each scenario are processed first before switching to rootkit batches of the same scenario.

Figure 13 visualizes the p-values computed with the sliding window method for probes iterate_dir-enter:iterate_dir-return using function-grouping (top) and probes filldir64-return:filldir64-enter using sequence-grouping (bottom). In both plots, batches occurring after the switches from normal to rootkit batches and vice versa receive low p-values since their delta values do not fit the distributions of delta values of the preceding batches. These results indicate that rootkits are detected at the next probing point immediately after they become active. Consequently, detection latencies are directly tied to the probing interval, which consists of the time taken for collecting the time measurements from the kernel and computing the p-values, plus the arbitrarily defined delay time in between. Assuming that rootkits can become active at any point in time, the average detection latency is half of that interval.

Similar to the observations made in Section 6.4.2, sequence-grouping appears to suffer from more false positives than function-grouping. To numerically assess the detection performance, we again need to count correct and incorrect classifications. However, online detection with sliding windows implies that training data comprise both normal and rootkit batches when the window passes over the point where these two classes meet. To account for the fact that the model is unreliable in that case, we compute the evaluation metrics as follows. Positives are the first normal and rootkit batches occurring just after the switching from one class to another. We count them as

true positives (TP) if they are detected as anomalous and false negatives (FN) otherwise. Negatives are all batches that occur after at least w batches from the same class have been processed to ensure that the predictions are only counted when the model is trained with either normal or rootkit batches, but not a mix thereof. We count them as false positives (FP) if they are detected as anomalous and true negatives (TN) otherwise. Combining the p-values of all probes and computing the evaluation metrics using the equations stated in Section 6.4.2, we obtain $TPR = 100\%$, $TNR = 97\%$, $Prec = 28.1\%$, $Acc = 97\%$, and $F1 = 43.9\%$ for function-grouping and $TPR = 100\%$, $TNR = 90.7\%$, $Prec = 11.3\%$, $Acc = 90.8\%$, and $F1 = 20.2\%$ for sequence-grouping. This confirms that function-grouping outperforms sequence-grouping across all metrics, which aligns with the results obtained from offline evaluation.

We also apply the benchmark detection approaches based on SVM and ANN (cf. Section 6.4.2) in the online setting for comparison. SVM yields $TPR = 66.7\%$, $TNR = 98.9\%$, $Prec = 42.9\%$, $Acc = 98.6\%$, and $F1 = 52.2\%$ for function-grouping and $TPR = 77.8\%$, $TNR = 95.7\%$, $Prec = 17.5\%$, $Acc = 95.4\%$, and $F1 = 28.6\%$ for sequence-grouping. Different from the evaluation in the offline case, SVM achieves higher F1 scores than our shift-based detection approach for function-grouping and sequence-grouping, respectively. The main reason for this is the higher precision ($Prec$). However, we point out that this improvement comes at the cost of a substantially lower TPR, meaning that several rootkit activities remained undetected. We make similar observations for ANN, which yields $TPR = 66.7\%$, $TNR = 98.4\%$, $Prec = 33.3\%$, $Acc = 98\%$, and $F1 = 44.4\%$ for function-grouping and $TPR = 44.4\%$, $TNR = 99.9\%$, $Prec = 80\%$, $Acc = 99.2\%$, and $F1 = 57.1\%$ for sequence-grouping. Even though sequence-grouping outperforms function-grouping in terms of F1 score in this case, TPR reaches the lowest value of all experiments with only 44.4%.

6.4.4 Impact on System Performance. In addition to detection metrics that have been evaluated in the previous sections, it is important for real-world applications of detection systems that employed mechanisms only impose a minimal and reasonable footprint on system resources to ensure scalability. Due to the fact that our approach relies on ongoing probing of multiple kernel functions in certain intervals for online rootkit detection, it introduces persistent computational overhead that should not be disregarded. Even though we emphasize that our approach is primarily implemented as a proof-of-concept prototype for research purposes and has not been designed for application in production environments, it can provide useful insights regarding the types of resources required for such operations.

To assess the impact on system performance, we conduct an experiment where we monitor key system metrics during probe injection and collection of delta times. Specifically, we select CPU and RAM utilization as well as disk activity as the most relevant indicators. In our illustrative setup, we run our procedure to generate 10 normal batches of data as described in Section 6.2.1; this means that we execute 10 iterations with breaks of 10 seconds in between, where each iteration involves injection of probes into the kernel, execution of 100 "ls" commands, polling of delta times from the ring buffer, and saving the collected datasets to disk. In addition, we test a lightweight variant of this procedure including only 10 "ls" commands to determine the influence of this parameter. During our experiment, we leave the system idle for 3 minutes before, after, and in between execution of these two variants to ensure that the metrics are not affected by any background activities. The experiment is conducted on a virtual machine with 4 vCPUs and 8 GB RAM.

Figure 14 visualizes the system performance metrics monitored throughout our experiment as timelines, where the first shaded interval indicates the execution of the lightweight variant of our procedure and the second shaded interval indicates the execution of the procedure in its default setting (cf. Section 6.2.1). In each interval, the CPU usage peaks from 0% to roughly 25% 10 times corresponding to the 10 iterations of batch generation. At the same time, RAM usage only increases from 2% to 3%. While CPU and RAM usage are mostly similar across the two variants, the lightweight variant requires less disk activity in comparison to the default one. This stands to reason since fewer executions of the "ls" command causes that kernel functions are called less frequently, which in turn means that fewer delta times are captured and stored to disk.

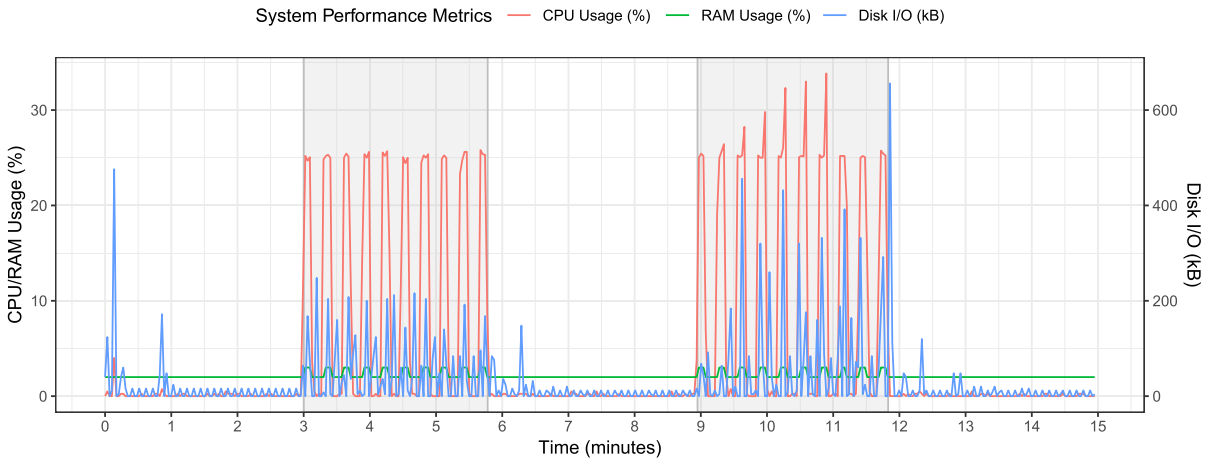

Fig. 14. System performance measurements depicting increased resource utilization when the probing mechanism is active (shaded intervals).

We conclude from our experiment that our approach primarily affects CPU usage; RAM is hardly affected and disk activity is less than 1 MB per batch. Given that our implementation is a research prototype rather than a tool ready to be used in production environments, potential code optimizations could minimize the computational cost of our probing mechanism. Other than that, we argue that smart timing of the execution of our procedure could alleviate the impact on system performance. In particular, since probing only takes a few seconds, it could take place when the system is idle and few computational resources are utilized at that time. We discuss these ideas in more detail in the following section.

7 Discussion

Manipulation of directory listings is one of the main capabilities of most types of rootkits. Even when rootkits target domains other than file hiding, they still need to interfere with file enumeration in order to hide themselves from detection. This article demonstrates the detectability of such rootkits through measurement of time delays of function calls within system calls. Thereby, these delays are caused by additional code that needs to be executed when rootkits wrap around certain functions for file enumeration to hide their presence and other objects. Accordingly, our approach is suitable to detect all types of rootkits that make use of system call wrapping for file hiding, in particular, persistent and polymorphic rootkits.

Based on our insights gained from this work, we answer the research questions as follows: *RQ1: What system calls enable the observation of rootkits that hide files?* Based on a review of existing open source rootkits and specifically their methods to hide files, we found that among many system calls that may be affected by rootkits, the getdents system call is the most relevant as it is the main interface to list file contents on Linux. Thereby, we point out that rootkits do not necessarily have to target the entire system call but may also wrap around some of its inner functions, such as filldir. *RQ2: How can delays of relevant function calls be observed?* Absolute time measurements can be collected through injection of eBPF probes that attach to enter and return points of functions within the kernel. Based on these measurements, delta times between any combination of probes can be computed. For example, this allows to compute the time to execute entire functions (function-grouping) or the time between two subsequently invoked functions (sequence-grouping). *RQ3: To what degree can anomaly detection techniques leverage system call function timings to uncover hidden rootkit activities?* Our proof-of-concept anomaly detection approach relies on statistical tests at certain quantiles of delta time distributions to recognize time shifts in comparison to normal behavior models. The results of our evaluation suggest that this approach is

able to detect rootkit activity with high accuracy; however, false positives have been noticed as a problem that could limit practical applicability. In particular, system conditions have a significant influence on the delta times, which could trigger many false positives in highly dynamic systems.

We foresee several ways to address the issue of negative influence of varying system conditions on the detection accuracy. First, practical implementations of our detection method could make use of dynamic probing intervals rather than regular intervals of 10 seconds used in our data generation procedure. In particular, the probing mechanism could wait for the system to be in an idle state to reduce noise that affects execution times of functions and interferes with time measurements. In addition, detected anomalies could trigger additional probing in short intervals that allow to determine whether the observed shift of delta times is constant and persists over time as it should be expected when a rootkit permanently wraps a function, or whether the currently processed batch of time measurements should be considered an outlier and the system can be regarded as normal despite an anomaly. Such an approach could also be used to assign confidence scores to predictions. Second, the time measurements themselves could be made more robust by assigning high priorities to the probing mechanisms so that other concurrent system processes do not interfere with the observed function call timings. Third, the detection approach based on statistical testing could be replaced with more generic alternatives. In particular, neural networks are well suited to ingest the complex and non-linear nature of delta time distributions and could thus be used to assign anomaly scores to system states. Fourth, in comparison to our approach that uses only data from a single scenario for training, other approaches could leverage data from multiple scenarios for training to generate a single normal behavior model that enables classification of batches from any of these scenarios. As visible in the right plot of Figure 8, delta times are sufficiently different across scenarios so that batches can be first assigned a scenario through clustering with sequence-grouped delta times before carrying out detection with function-grouped delta times. Thereby, our procedure of generating datasets could be extended with new scenarios that introduce other forms of noise to obtain datasets with even more variation of normal behavior. Alternatively, it is also possible to mix two or more existing scenarios so that multiple sources of noise occur at the same time. We refer to the work by Singh et al. [37], who introduce noise by interacting with various programs, such as browsers and benchmark tools. Fifth, in contrast to our semi-supervised approach, supervised approaches could either make use of labeled instances of specific scenarios or even batches generated when rootkits are active to further improve classification and detection performance.

Several modifications could be made to our rootkit to extend the evaluation. In addition to wrapping filldir, the rootkit can alternatively wrap the entire getdents system call for file hiding. It could thus be interesting to investigate to what degree the detection of these methods differ when it comes to detection. Moreover, the rootkit does not only support file hiding, but also process hiding. After experimenting with both functionalities, we noticed that there is no significant difference when it comes to detection, which is why we focused on the simpler case of file hiding in this article. Finally, while the implementation of our approach is designed for and evaluated on Linux machines, the concept of system calls is agnostic to operating systems. Therefore, our concept of measuring low-level function timings for rootkit detection may be transferred to other operating systems, such as Windows [3, 37].

8 Conclusion

This article presents a semi-supervised and anomaly-based approach that leverages statistical testing for rootkit detection based on kernel function timings. The main idea behind this concept is that rootkits need to inject code that modifies the outcome of specific kernel functions to hide their presence from users or detection programs, which increases the runtime of these functions. To facilitate measurement of these time intervals, we present a framework that injects probes into the kernel, attaches them to enter and return points of functions, and polls timestamps for invocations of selected functions. Thereby, we found that the getdents system call and its inner functions are of particular relevance as they are key to enable hiding capabilities of rootkits. We convert

absolute time measurements into delta times using two strategies that focus on functions and sequential steps of program workflows, respectively. Our analysis suggests that function-grouping has advantages when it comes to the detection of shifted delta times, while sequence-grouping is superior when it comes to classification of system states. We collect batches of delta times in five different scenarios using a custom rootkit and test our detection approach in offline and online settings. The results of our evaluation indicate high detection accuracy and leave many interesting research opportunities for future work, such as mechanisms for dynamic probing, rootkit detection across different system states, and experiments with machine learning models other than statistical tests.

References

[1] Leonhard Alton. 2024. *Root Kit Discovery with Behavior-based Anomaly Detection through eBPF*. Master's thesis. Vienna University of Technology.

[2] Kshitiz Aryal, Maanak Gupta, Mahmoud Abdelsalam, Pradip Kunwar, and Bhavani Thuraisingham. 2024. A survey on adversarial attacks for malware analysis. *IEEE Access* 13 (2024), 428–459.

[3] Pablo Bravo and Daniel F. Garcia. 2011. Proactive detection of kernel-mode rootkits. In *Proceedings of the 6th International Conference on Availability, Reliability and Security*. IEEE, 515–520.

[4] Robert C Brodbeck. 2012. *Covert Android Rootkit Detection: Evaluating Linux Kernel Level Rootkits on the Android Operating System*. Master's thesis. Air Force Institute of Technology.

[5] Andreas Bunten. 2004. Unix and linux based rootkits techniques and countermeasures. In *Proceedings of the 16th Annual Conference on Computer Security Incident Handling*, 1–16.

[6] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. 2019. On evaluating adversarial robustness. arXiv:1902.06705. Retrieved from https://arxiv.org/abs/1902.06705

[7] Nadir A. Carreón, Allison Gilbreath, and Roman Lysecky. 2020. Statistical time-based intrusion detection in embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 562–567.

[8] Australian Cyber Security Centre, Canadian Centre for Cyber Security, New Zealand National Cyber Security Centre, CERT New Zealand, UK National Cyber Security Centre, US National Cybersecurity, and Communications Integration Center. 2018. Joint Report on Publicly Available Hacking Tools. Retrieved January 8, 2025 from https://www.ncsc.gov.uk/files/Joint%20report%20on%20publicly%20available%20hacking%20tools%20(NCSC).pdf

[9] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Computing Surveys* 41, 3 (2009), 1–58.

[10] Edward J. M. Colbert and Alexander Kott. 2016. *Cyber-Security of SCADA and Other Industrial Control Systems*. Vol. 66, Springer.

[11] CrowdStrike. 2020. 2020 Global Threat Report. Retrieved January 8, 2025 from https://www.crowdstrike.com/resources/reports/2020-crowdstrike-global-threat-report/

[12] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. 2017. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques* 13, 1 (2017), 1–12.

[13] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of the Symposium on Security and Privacy*. IEEE, 20–38.

[14] Joel A. Dawson, Jeffrey T. McDonald, Lee Hively, Todd R. Andel, Mark Yampolskiy, and Charles Hubbard. 2018. Phase space detection of virtual machine cyber events through hypervisor-level system call analysis. In *Proceedings of the International Conference on Data Intelligence and Security*. IEEE, 159–167.

[15] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 559–570.

[16] Kjell Doksum. 1974. Empirical probability plots and statistical inference for nonlinear models in the two-sample case. *The Annals of Statistics* 2, 2 (1974), 267–277.

[17] Thomas R. Etherington. 2019. Mahalanobis distances and ecological niche modelling: Correcting a chi-squared probability error. *PeerJ* 7 (2019), 1–8.

[18] Okwudili M. Ezeme, Qusay H. Mahmoud, and Akramul Azim. 2020. A framework for anomaly detection in time-driven and event-driven processes using kernel traces. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 1–14.

[19] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. 2008. The evolution of system-call monitoring. In *Proceedings of the Annual Computer Security Applications Conference*. IEEE, 418–430.

[20] Augusto Remillano II, Jakub Urbanec, and Wilbert Luy. 2019. Skidmap Malware Uses Rootkit to Hide Mining Payload. Retrieved from https://www.trendmicro.com/en_us/research/19/i/skidmap-linux-malware-uses-rootkit-capabilities-to-hide-cryptocurrency-mining-payload.html

[21] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. 2019. Survey of intrusion detection systems: Techniques, datasets and challenges. *Cybersecurity* 2, 1 (2019), 1–22.
[22] Rob Landley. 2005. ramfs, rootfs and initramfs. Retrieved from https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt
[23] Michael Leibowitz. 2016. *Horse Pill: A New Kind of Linux Rootkit*. Black Hat USA.
[24] Marc-Etienne M. Léveillé. 2024. Ebury is Alive but Unseen. Retrieved from https://web-assets.esetstatic.com/wls/en/papers/white-papers/ebury-is-alive-but-unseen.pdf
[25] Sixing Lu and Roman Lysecky. 2019. Data-driven anomaly detection with timing features for embedded systems. *ACM Transactions on Design Automation of Electronic Systems* 24, 3 (2019), 1–27.
[26] Patrick Luckett, J. Todd McDonald, and Joel Dawson. 2016. Neural network analysis of system call timing for rootkit detection. In *Proceedings of the Cybersecurity Symposium*. IEEE, 1–6.
[27] Mohammad Nadim, David Akopian, and Wonjun Lee. 2021. A review on learning-based detection approaches of the kernel-level rootkit. In *Proceedings of the International Conference on Engineering and Emerging Technologies*. IEEE, 1–6.
[28] Mohammad Nadim, Wonjun Lee, and David Akopian. 2023. Kernel-level rootkit detection, prevention and behavior profiling: A taxonomy and survey. arXiv:2304.00473. Retrieved from https://arxiv.org/abs/2304.00473
[29] Jordan Pattee, Shafayat Mowla Anik, and Byeong Kil Lee. 2022. Performance monitoring counter based intelligent malware detection and design alternatives. *IEEE Access* 10 (2022), 28685–28692.
[30] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. 2017. Malware detection by eating a whole exe. arXiv:1710.09435. Retrieved from https://arxiv.org/abs/1710.09435
[31] Edward Raff, William Fleshman, Richard Zak, Hyrum S. Anderson, Bobby Filar, and Mark McLean. 2021. Classifying sequences of extreme length with constant memory applied to malware detection. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, 9386–9394.
[32] Geetha Ramani and Suresh Kumar. 2021. Nonvolatile kernel rootkit detection using cross-view clean boot in cloud computing. *Concurrency and Computation: Practice and Experience* 33, 3 (2021), 1–11.
[33] Joanna Rutkowska. 2006. Introducing blue pill. *The Official Blog of the Invisiblethings.org* 22 (2006), 23.
[34] Mahmoud Salem, Mark Crowley, and Sebastian Fischmeister. 2016. Anomaly detection using inter-arrival curves for real-time systems. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*. IEEE, 97–106.
[35] Hossein Sayadi, Yifeng Gao, Hosein Mohammadi Makrani, Jessica Lin, Paulo Cesar Costa, Setareh Rafatirad, and Houman Homayoun. 2021. Towards accurate run-time hardware-assisted stealthy malware detection: A lightweight, yet effective time series CNN-based approach. *Cryptography* 5, 4 (2021), 28.
[36] Hossein Sayadi, Zhangying He, Hosein Mohammadi Makrani, and Houman Homayoun. 2024. Intelligent malware detection based on hardware performance counters: A comprehensive survey. In *Proceedings of the 25th International Symposium on Quality Electronic Design*. IEEE, 1–10.
[37] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. 2017. On the detection of kernel-level rootkits using hardware performance counters. In *Proceedings of the Asia Conference on Computer and Communications Security*. ACM, 483–493.
[38] Jakob Stühn, Jan-Niclas Hilgert, and Martin Lambertz. 2024. The hidden threat: Analysis of linux rootkit techniques and limitations of current detection tools. *Digital Threats: Research and Practice* 5, 3 (2024), 1–24.
[39] Donghai Tian, Rui Ma, Xiaoqi Jia, and Changzhen Hu. 2019. A kernel rootkit detection approach based on virtualization and machine learning. *IEEE Access* 7 (2019), 91657–91666.
[40] Xueyang Wang and Ramesh Karri. 2013. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Proceedings of the 50th Annual Design Automation Conference*, 1–7.
[41] Rand R. Wilcox and David M. Erceg-Hurn. 2012. Comparing two dependent groups via quantiles. *Journal of Applied Statistics* 39, 12 (2012), 2655–2664.
[42] Yun-Che Yu, Ci-Yi Hung, and Li-Der Chou. 2025. Kernel-level hidden rootkit detection based on eBPF. *Computers and Security* 157 (2025), 104582.
[43] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. 2010. Time-based intrusion detection in cyber-physical systems. In *Proceedings of the International Conference on Cyber-Physical Systems*, 109–118.

Received 14 April 2025; revised 10 September 2025; accepted 25 September 2025