National College of Ireland

Linux Based Rootkit Detection Leveraging eBPF

MSc Research Project
Cybersecurity

Brian O'Neill
Student ID: x23321156

School of Computing
National College of Ireland

Supervisor: Raza Ul Mustafa

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Brian O'Neill
Student ID:	x23321156
Programme:	MSCCYB_A **Year:** 2024/25
Module:	Practicum
Lecturer:	Raza Ul Mustafa
Submission Due Date:	11 August 2025
Project Title:	Linux Based Rootkit Detection Leveraging eBPF……
Word Count:	10134…… **Page Count:** 22……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Brian O'Neill……

Date: 11 August 2025……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

AI Acknowledgement Supplement

1 Research Paper

Your Name/Student Number	Course	Date
X23321156	MSCCYB_A	11/08/2025

This section is a supplement to the main assignment, to be used if AI was used in any capacity in the creation of your assignment; if you have queries about how to do this, please contact your lecturer. For an example of how to fill these sections out, please click here.

2 AI Acknowledgment

This section acknowledges the AI tools that were utilized in the process of completing this assignment.

Tool Name	Brief Description	Link to tool
ChatGPT	LLM	https://chatgpt.com/

3 Description of AI Usage

This section provides a more detailed description of how the AI tools were used in the assignment. It includes information about the prompts given to the AI tool, the responses received, and how these responses were utilized or modified in the assignment. **One table should be used for each tool used**.

[Insert Tool Name]	
Query on dynamic probing. This did not work in the end but it is left in a commented out code in the artifact.	

I am trying to address this problem that Landauer et al. suggest should be addressed: "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." What do you suggest is the best approach?	**Idea:**
	• When the **prev_comm == "idle"**, the CPU is **no longer idle**.
	• When the **next_comm == "idle"**, the CPU **just became idle**.
	• You can use this to **gate your probes**.
	Implementation Plan:
	• Keep a BPF map tracking **idle state per CPU**.
	• Only record timing deltas (e.g., filldir64_entry → filldir64_return) when the corresponding CPU is idle or has been idle for N microseconds.
	
	BPF_PERCPU_ARRAY(is_idle_cpu, u64, 1); // 1 = idle, 0 = busy
	
	TRACEPOINT_PROBE(sched, sched_switch) { u32 cpu = bpf_get_smp_processor_id();
	
	if (__builtin_memcmp(args->next_comm, "idle",

```
5) == 0) {
        u64 idle = 1;
    is_idle_cpu.update(&cpu, &idle);
    } else {
        u64 busy = 0;
    is_idle_cpu.update(&cpu, &busy);
    }
    return 0;
}
```

5 Evidence of AI Usage

This section includes evidence of significant prompts and responses used or generated through the AI tool. It should provide a clear understanding of the extent to which the AI tool was used in the assignment. Evidence may be attached via screenshots or text.

6 Additional Evidence:

Idea:

- When the `prev_comm == "idle"`, the CPU is **no longer idle**.
- When the `next_comm == "idle"`, the CPU **just became idle**.
- You can use this to **gate your probes**.

Implementation Plan:

- Keep a BPF map tracking **idle state per CPU**.
- Only record timing deltas (e.g., `filldir64_entry` → `filldir64_return`) when the corresponding CPU is idle or has been idle for N microseconds.

```
BPF_PERCPU_ARRAY(is_idle_cpu, u64, 1); // 1 = idle, 0 = busy

TRACEPOINT_PROBE(sched, sched_switch) {
    u32 cpu = bpf_get_smp_processor_id();

    if (__builtin_memcmp(args->next_comm, "idle", 5) == 0) {
        u64 idle = 1;
        is_idle_cpu.update(&cpu, &idle);
    } else {
        u64 busy = 0;
        is_idle_cpu.update(&cpu, &busy);
    }
    return 0;
}
```

7 Additional Evidence:

[Place evidence here]

Linux Based Rootkit Detection using eBPF

Brian O'Neill
Student ID: x23321156

Abstract

Kernel rootkits inject code into kernel functions and can remain undetected due to their ability to hide their processes. Existing approaches fall short in the ability to provide adequate detection coverage and can be impractical in terms of the need to develop kernel modules or use extra hardware and can be inefficient use of resources in their detection. To address some of these challenges it is proposed that eBPF be leveraged for its ability to programming the kernel dynamically, and can be used for networking, observability, and security.

Previous work using eBPF for rootkit detection has approached the problem by observing the impact of rootkits on the performance of some system elements as an indicator of compromise. This has seen a degree of success. This work seeks to address the influence of system conditions on detection accuracy. It follows a methodology that creates and analysis timing data from targets gathered within the Linux kernel. By comparing clean data and data gathered in the presence of a rootkit it seeks to obverse difference in execution times of subsystem kernel functions due to the rootkit. Data is gathered dynamically to observe impacts of a rootkit on system events and at the same time gather kernel events that can help discriminate against events that will adversely affect accuracy in detection. Schedular policies are investigated as a mechanism to dampen down processes unrelated to the processes and task of the application at hand and produce cleaner data for analysis. Statistical significance was found in the impacts of rootkits on the chosen timing targets within the kernel and some marginal gains were found in masking off system noise.

Keywords: rootkit detection, anomaly detection, eBPF, kernel probes, tracing.

1. Introduction

1.1. Preamble

Rootkits are programs designed to gain control of a system and remain undetected by altering the operating system. Kernel level rootkits are notoriously difficult to detect pieces of malware that can carry out operations with a high level of privileges on an operating system. Malicious actions may include hiding network communications, gathering sensitive data, process and module hiding. Kernel level rootkits can control their appearance and presence through kernel modifications. There are a number of generations of rootkits that have evolved from those that are executed in user space to those that execute in kernel space, and they have become harder to detect either through Intrusion Detection Systems (IDS) or other security tools (Nadim et al., 2023). Other surveys have highlighted the lack of tools available for Linux Operating Systems (Stühn et al., 2024).

Rootkits targeting Linux in the wild remains prevalent and is often targeted in cloud native environments. In 2021 threat group TeamTNT used the open-source

rootkit Diamorphine to hide crypto mining activities in a sustained attack by running malicious containers in cloud hosting providers. During their attacks they tried to affect multiple distributions of Linux including Ubuntu, Debian, Red Hat, Fedora or CentOS, and more (Morag, 2021). From 2020 to 2024 and possibly later, malware, known as 'perfctl' is also known to hide its presence by utilizing a rootkit on Linux servers (Morag and Revivo, 2024). In 2025 with many organizations continuing the trend of moving from on-premises systems to the cloud, containerized applications running on Linux, rootkit detection should remain an area that requires vigilance.

According to Billimoria, user-space security has improved considerably in recent years. The example he gives is one of Buffer Overflow attacks being straightforward in Linux in the past but difficult to exploit with recent kernels. Kernel space attacks on the other hand have increased (Billimoria, 2024). Additionally, Nadim et al., say 'recently, the kernel-level rootkit technique is employed by more and more malware to gain high privilege in the OS kernel so that they can hide their malicious activities' (Nadim et al., 2023, p. 2). Consequently, it is the intention of this paper to focus on kernel-level rootkit detection.

1.2. Background

1.2.1. Summary of Historical Research

Rootkits were known to exist on UNIX as far back as the 1980s and early iterations were designed to hide their presence either through manipulation of log files of the substitution of UNIX equivalents of coreutils; programs like ls, ps, or netstat. Methods were shared on bulletin boards and magazines. "Hiding Out Under Unix," by Black Tie Affair on Phrack, (an e-zine written for an by hackers), published source code to edit the /etc/wtmp file to remove login records (Dittrich, 2002). By the mid-1990s Phrack was publishing articles discussing way to abuse the Linux kernel by way of loading kernel modules and redirecting syscalls to the attacker's code (halflife, 1996). 'Knark' was an early kernel module rootkit which hid listening sockets, files, and directories, and there are some mentions of it targeting Linux kernel version 2.2 which was first available in 1999. A 2002 GCIH paper described 'The t0rn Rootkit', a binary replacement rootkit, which itself was based on the Linux Rootkit LRK. That same year, 21-year-old 'J0hnny7' the otherwise unnamed author of t0rn was arrested in the UK. The rootkit was believed to have been used as part of the so-called Lion worm use by Chinese hackers, thus prompting the arrest (BBC News, 2002).

Stühn et al. gives an overview on research on rootkits covering a period from 2004 onwards. Some of them focus on kernel level rootkits targeting the system table in older Linux Kernels such as described by Levine et al. According to their paper syscall addresses in the system call data structure are named as valuable attack targets, as calling them passes control to the kernel and subsequently return output to the caller. In the interim time between the call and the return execution can be jumped to malicious code. This is easily detectable by looking for expected op code values. Another example of rootkit they provide is to redirect execution to an entirely different system call table by overwriting the pointer to the original table (Levine et al., 2006). As such their focus of research is quite narrow. As an aside that may play into the research in this paper later on; rootkits using syscall hooks would not find it easy to hijack a timer that was measuring syscall execution duration. Joy et al., also underline identify the syscall table modification as a target for kernel level rootkits or by modifying the syscall target itself (Joy et al., 2011). Stühn's paper mentions that studies on rootkit detection in digital forensics are limited. One paper in this area addresses the problem of locating paging structures of malicious processes from

memory snapshots (Saur and Grizzard, 2010). They predicate their work on x86 hardware support for mapping virtual addresses to physical addresses through paging processes and being able to map out all processes in memory. Identification of rouge processes is done by comparison to expected processes. Stühn et al., give an overview of detection tools available and approaches to forensics but conclude that tools to detect Linux rootkits are inadequate and dependent on details knowledge of both the system and rootkit in question for successful detections. They call for a more generalized approach.

In response to this and other analysis pointing to rootkit detection deficits, Landauer et al. propose a more generalized solution by looking for rootkit related temporal anomalies in the kernel. Their detection method "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" (Landauer et al., 2025, p. 1). Key results from their paper demonstrate a high degree of accuracy with an F1 score of 98.7% across five scenarios.

1.3. Research Question

The research question here attempts to build upon the suggestions of future work from Landauer et al. Namely, how can the accuracy of rootkit detection based on kernel function timing be improved through dynamic probing and prioritising probing mechanisms and by way of accounting for general system 'noise' – that is events from the system that would potentially introduce bogus timing data into observations.

1.4. Importance

As has been touch upon earlier a generalised approach using temporal anomalies and indicators of compromise has the potential to overcome narrowly scoped detection methods such as signature-based detection. Such methods can add to existing techniques to provide better detection for novel or unknown rootkits.

Time-based monitoring of kernel code execution has the potential to operate with a minimal footprint, without impacting the overall functioning of the system, as could be achieved with eBPF. This has advantages in cloud native environments that use the likes of Kubernetes requiring 'sidecars containers' for observation of processes in the kernel. Typically these would operate alongside application containers in the same pod in order to provide supporting functionality ("Sidecar Containers," 2025). eBPF also allows for the development of custom tooling which can be injected dynamically into the kernel at runtime.

Work that the approach here is building on claims advantages of more fine-grained timing of system calls whereby they also account for timing of what happens inside the system call. Other tracing methods tends to take observations from the syscall level only without examining functions internal to the syscall. This approach could potentially pinpoint delays that rootkits introduce while the system calls themselves may remain within acceptable bounds. It also considers the problem of system noise and conditions which could skew the results towards false positives. Lastly an increased number of features are brought into collected data sets when compared to similar work (Landauer et al., 2025).

1.5. Research Objectives

The main objective of the following work is summarized as follows:

1.5.1 To investigate and implement a dynamic probing mechanism.

1.5.2 To investigate the impact of prioritizing probing processes.

1.5.3 To evaluate the effectiveness of the combined approach of dynamic probing and probe prioritization on detection accuracy quantitatively.

1.5.4 To account for the impact of system noise on detection accuracy.

1.5.1. Tests that will show whether objectives have been met.

As this work is primarily based on timing data deltas, statistical significance is sought between data gathered under the normal running of the system and when a rootkit is present. The main statistical method opted for was the use of the Mann-Whiteny U test. Reasoning for this is described later in the paper. Simple summary statistics were also used to observe the impact of changing kernel scheduling policies, which were run as part of an attempt to prioritize processes and tasks produced by the application itself.

1.5.2. Overview of methods

Function calls internal to system call and other kernel level events are captured sequentially and duration between them are measured. Data is captured with and without a rootkit present and compared for anomalies. This is described more fully in section 3. Kernel level events are captured in an attempt to account for system noise. Noise here is defined as 'the time spent by a CPU executing instructions not belonging to a given application task assigned to that CPU while the task is ready to run'. This is discussed more in section 2.4.

1.5.3. Limitations

Rootkits may subject kernel-space detection mechanisms to damage or evasion targeting the detection software itself in a guest OS. On a longer timeline an extended solution may be required such as moving the software to another layer such the hypervisor of a virtual machine or other side channel to put it outside of the domain of influence of a potential rootkit attack.

There a potentially multiple kernel functions relevant to which the solution here could target this paper focuses on filldir64 as proof of concept.

1.5.4. Assumptions

For the choice of the CPU scheduling policy, it was decided to opt for the aggressive SCHED_FIFO in order to ensure prioritisation of the probing mechanism and the default of SCHED_NORMAL for comparison. Further work may be needed to minimise its impact on system performance either by lowing the priority through adjustment of the priority scale value or change of policy or both.

It is a given that this solution would not on its own expected to be the sole function in detecting rootkits on a Linux OS. Rather its main purpose is to meet the objectives already set out in 1.5.

1.6. Report Structure

The rest of the paper is comprised of four sections. Section 2 lays out a review of literature relating to rootkits, approaches to their detection, also the use of timing mechanisms as a vehicle for detecting them and how eBPF can serve as a contemporary approach to developing a custom solution to gathering the necessary data from the kernel. Section 2 also review sources that are relevant to this particular paper in dealing with system noise their sources, scheduling prioritisation and the like. Section 3 deals with the particulars of the tracing targets that were identified in section 2, the logic to gather data on them and the types of measurements relevant to assessing them and lays out the architecture of the application, discusses implementation details. Section 4 analyses the results of analysis done on the data. Section 5 is a discussion section and concludes the paper.

2. Literature Review

2.1. Rootkit detection methods.

Stühn et al., provide an overview of Kernel Level Rootkit Techniques. These are rootkits that operate in kernel mode as opposed to user mode. X86[64] processors

support four privilege levels called ring levels. Though other processors can support up to seven modes of execution such as ARM-32 (ARM-64 also support four levels) (Billimoria, 2024). Kernel mode is where the core of the operating system executes as well as kernel modules such as device drivers, networking, I/O and other third-party kernel modules. Ring levels are used for handling faults, such as page faults when a program tries to access a portion of memory not mapped to physical memory, segmentation faults when a program tries to access memory that has its access restricted, or general protection faults which occur in response to an access violation by running code (x86 specific). Ring 3 has the least level of privileges whereas process cannot access the memory of other processes and resides is used by user space processes. Ring 0 has the most level of privileges and can do anything, meaning it can directly affect the system and its resources. Communication between each ring happens with well-defined interfaces. Inner domains and their interfaces will control what information can be passed to outer domains. “Rootkits can abuse this hierarchical flow of information to alter the system state presented to the user” (Stühn, Hilgert and Lambertz, 2024, p. 3). Nadim et al., note that modern rootkits have evolved from residing on disk to becoming in-memory processes with those injecting their code into the kernel space being hard to detect. They assert that the majority of the latest generation of rootkits are kernel level (Nadim, Lee and Akopian, 2023).

Stühn et al. state that almost all analyzed rootkits achieves code running inside the kernel by loading as Loadable Kernel Modules (LKMs). These modules are usually a means to provide kernel level functionality without having to build the functionality into the kernel itself directly, this could be to support a device driver which needs to use a hardware peripheral or a new file system. Loaded LKMs can be hidden from the user by deleting entries from an associated linked list in the kernel itself and exported to /proc/modules in Linux. Kernel object references can also be deleted from /sys/module. As a brief aside: 2020 saw the introduction of LSM BPF which allows eBPF programs to attach to the LSM kernel interface, which potentially allows for security tooling to be developed without the need for kernel modules (Rice, 2023). eBPF also provides low level network 7 monitoring and access (XDP) and system call prebuild hooks but is more stable than LKMs and can be used dynamically.

Kernel level rootkits can insert themselves into system calls in order to change information that system calls present to the user. They can replace function pointers in the sys call table, replace the entire table by way of interrupt handler modification, reroute system calls to a different function by using an unconditional jump operation, use ftrace (a Linux tracing framework) to perform a similar result; again, resulting in a call to an unintended and maliciously placed function. Stühn et al. mention another mechanism used by rootkits is the Virtual Filesystem Switch (VFS) which acts as an abstraction over filesystem variants like Ext4, ZFS, BtrFS and more. This allows for the management of multiple files systems by the kernel. Pseudo-files are used to create interfaces to handler functions in the kernel. Accessing such files will call the handler. Common pseudo-file targeted by rootkits are /proc/modules, rootfs, procfs, sysfs and devfs. Nadim et al. categorize subversion of the VFS by kernel-level rootkits as Dynamic Kernel Object Hooking, which describes process hiding via function pointer modification of lookup function the process directory’s /proc inode. Summing up findings of Stühn et al, they conclude that after analyzing 21 rootkits that and seven tools to detect rootkits that no could detect all rootkits tested. Furthermore, their performance worsened when the system and rootkit were not

known, which reflects real world scenarios. They tested using a variety of plugins for the Volatility Framework (extracts digital artifacts from RAM), for which no eBPF plugin was available, and suggested development of same. Nadim et al. presented findings in terms of strengths and weaknesses not of particular tools but of approaches to detection, some of which were distilled and categorized in Table 1. In their paper, Rootkit Detection Mechanisms for Linux Systems' Lu et al., propose rootkit detection mechanisms based on kprobes (kernel probes) and focus on cross-view based detection as a means to detect hidden processes. Kprobes have been available on Linux since 2005. They can 'trap' at almost any kernel code address which allows a handler routine to be invoked. Using kprobes they hook two system calls (init_module, delete_module) to observe if a module has been successfully loaded or unloaded to a linked list which tracks what kernel modules are running. (It is a tactic of rootkits to remove their references from this list). At the same time a separate process compares resulting records of these calls with information from a user-space kernel module auditing tool (the other view of the system) (Lu et al., 2023). They document successful experimental results that compare favorably against other methods. However, they also depend on a kernel module for their use of kprobes. This is not necessary with the use of eBPF which does not require a kernel module to be developed and can utilize kprobes to attach to syscalls dynamically and other kernel functions. Landauer et al. evaluated a number of LKM rootkits and observed all of them had commonality in the functions they wrapped when trying to hide themselves, namely getdents and filldir (Landauer et al., 2025).

2.2. Measuring anomalies in the timing of system calls.

Ezeme et al. look at the behaviour of system calls sent to the operating system by proposing a framework for anomaly detection in time-driven and even-driven processes. The core of their model is to capture the order of system calls and relative duration between them. The premise; malicious code will create anomalous execution times or delays when compared to time windows between system calls as gathered by observed traces (Ezeme et al., 2022). This is defined this as follows:

$$B: (t_i, t_{i+1}) \longmapsto t_{i+1} - t_i;\ I \in N$$

This gives a duration between system call timestamps, t1, t2, …, tn. This could be an important method when applied rootkit detection where the order of system calls is maintained in order to avoid detection through signatures or other methods and where delays, or indeed sudden speed-ups, provides a layer of abstraction or generalization towards detection.

The critique offered by Landauer et al., for this strategy is that further granularity is possible in the capture of such timing data. By contrast, the method Landauer et al. 'goes beyond existing works that only use execution times of entire system calls, neglect multimodal features in collected data sets, and do not consider the influence of system conditions and noise' (Landauer et al., 2025, p. 1).

Luckett et al., propose using neural networks to perform analysis on system call timing for Rootkit Detection. Their greatest success came with a recurrent neural network with a detection accuracy of 95.9%. They assert that although rootkits can hide themselves, they must still make use of system calls (Luckett et al., 2016). This claim may not always hold true in relation to kernel-level rootkits as system calls are usually an interface between user space and the kernel, and as they are already operating in the privileged context of ring zero, they can therefore call kernel

functions directly or hook internal kernel functions. Though it may allow them to intercept system call handlers and modify them in order to hide itself or control behavior. It could however be expected that kernel-level rootkits hook certain system calls to hide themselves from user-space inspection. Luckett et al. first tried standard statistical measures on data sets but say this method failed due to the data being nonlinear. Though they do not elaborate on why, possible reasons that this did not work could be due to non-significant increases in mean values describing delays in system calls, or bursts of activity that do not show when aggregated to a value expressed as a mean value. Normal system activity may contribute to noise making it hard to distinguish between deviations that are either benign or caused as the result of rootkit behavior. Timing data could also follow probability distributions that follow more than one mode, which may align with Landauer et al and their want for examining multi-modal distribution.

Dawson et al., introduce a 'phase-space algorithm' to collect time-series data on system calls comparing nominal and abnormal behavior. Focused on ecosystems that make use of virtual machines, they do this by ways of collecting data from a virtual machine from the level of a hypervisor, arguing that this will prevent the rootkit from avoiding detection as won't be able to affect detection software which is effectively in a different environment from the virtual machine. Their approach to system call tracing is centered on isolated system calls as opposed to Luckett et al., whose timing data is on full system call traces meaning all system calls made by a process or system during execution. The implementation of Dawson et al.'s phase-space algorithm results in measuring differences in mean dissimilarity and average standard deviation in the baseline data set and compares test sets to those measures, averaging results to produce a result of average dissimilarity (Dawson et al., 2018). Dissimilarity defines the variation from the baseline in relation to a predefined threshold. Interestingly, their algorithm had the option of applying a quadratic fit filter to remove noise which they felt they need not use based on the reliability of their data set.

As alluded to Landauer et al. supersedes many of these approaches. Whereas other approaches find that rootkits can cause delays in system calls, many are scoped to the system calls themselves and overlook finer details of more complex distribution patterns that can occur in the measurement of such timing data. "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" (Landauer et al., 2025, p. 4). Detecting anomalies often use the approach of creating models from normal behavior to create a baseline and look for deviations from that baseline. Frequently this is done by monitoring a system's execution sequence of function calls. This level of granularity does not however protect against mimicry malware, which, will keep the execution sequence but interleave malicious functions with system functions (Carreón et al., 2020).

Some arguments have been made that detecting anomalies by targeting syscall be premised on the idea that in environments such as server environments a limited number of tasks get executed and get executed repeatedly following predictable code paths (Forrest et al., 2008). The cited work of Forrest et al., while establishing baselines for normal system profiles, they do not account for system noise but point

out that under generalised models will lead to false positives and the opposite to false negatives. They also observed that errors resulted in categorisation as anomalies that were non malicious. A proposal for handling noise will be proposed further in this paper.

Authors	Instrumentation	Target	Granularity	Noise Handling	Prediction Model	Dynamic Probing
Landauer et al. 2025	eBPF probes	Syscalls (getdents/filldir64)	Inter-call timing of complete call stack (subsystem kernel functions measured in nanoseconds)	Statistically	Statistical anomalies detected with semi-supervised	✗
Dawson et al. 2023	strace	Syscalls (open, close, read, futex, mmap, clock_gettime)	Inter-call timing of syscalls	Quadratic fit filter	Phase space algorithm	✗
Ezeme et al. 2022	Unnamed 'kernel traces' done with a python 'instrumentation script'.	Syscalls (330 types not identified by name)	Inter-call timing of syscalls	✗	RNN (LSTM)	✗
Luckett et al. 2016	Unspecified	Syscalls	Unspecified	✗	RNN	✗

Table 1.

The work here is based on the back of future work recommendations of Landauer et al. A brief synopsis of their solution follows. They instrument the kernel with eBPF probes in order to collect system call function timings under both normal conditions and with the rootkits present. Deltas are computed on syscall sub-function timings based on data gathered from pairs of eBPF probes. That is, timestamps are gathered both at the entry and exit points on the functions and also deltas of probes irrespective of function e.g. time between probes spanning functions. Measurements are calculated in nanoseconds and stored separately for the baseline and rootkit measurements. Baseline measurements are used to train a normal behaviour model. Detection then is made through a measurement of shifted delta times between the data sets, categorising the outcome as either and anomaly or normal. The model predicts on unseen deltas.

2.3. eBPF

eBPF was used by Landauer et all to inject probes into the kernel to take measurements. For the implementation outlined in this paper a similar probe structure is expected. Similarly, the artifact accompanying this paper injects probes using eBPF into the kernel. However here, additional targets are added to evaluate the impacts from other events in the system, effectively being treated as 'noise'. This will be described more fully under methodology and other sections. For now, it is noted that they capture current process ids, thread groups, and high precision timestamps using *bpf_ktime_get_ns( ).* Probes are injected using the BPF Compiler Collection (BCC), a toolkit for kernel tracing and manipulation.

eBPF is a relatively new technology that allows dynamic programming of the kernel through a lightweight virtual machine. Originally developed for the Linux kernel, eBPF is now also being developed for Windows. It has been described as like putting JavaScript into the kernel in the sense that it can run custom code dynamically, extending kernel functionality without modifying the underlying core of the operating system. When software needs to interface with hardware in any way the kernel has to be involved. eBPF allows software to be developed and run custom programs that can observe and change what's happening in the kernel and have eyes across everything that is happening on the machine. eBPF can observe and potentially modify the behavior of any process that is running, which in turn is potentially useful for security applications, system and network observability, and generally 'instrumenting the kernel'. At the same time eBPF carries with it an in-kernel verifier to ensure eBPF programs are safe to run. It is noted for its high-speed processing and performance gains in some scenarios when compared with other methods and tools in policy enforcement, networking and security.

eBPF has its roots stretching back to the Berkely Packet Filter which as its name suggests is a network packet filter using an instruction set similar to assembler. In 2014 this evolved into "extended BPF" now simply referred to as eBPF starting in the Linux Kernel version 3.18. It continues to evolve and became a separate subsystem in 2018. In 2020 eBPF introduced the ability to attach to the Linux Security Module (LSM).

Caviglione et al, regard it as "the most powerful tool for gathering data in the perspective of detecting stegomalware, covering both code tracing and packet inspection" (Caviglione et al., 2021, p. 3). eBPF does not require additional kernel modules compared to other similar solutions. It uses event driven architecture to hook to particular events and trigger on event execution, making it an efficient solution for continuous monitoring.

The benefits of using eBPF as applied in this paper's solution include:

- eBPF programs execute within the kernel and can directly interact with system level events.
- eBPF programs ensures safety through a verifier, whereby kernel crashes and security breaches are prevented by checking their logic before running.
- The overhead of eBPF programs is minimal. By using a just-in-time (JIT) compiler the execution speed of such programs is close to native in performance. The JIT achieves this by turning eBPF bytecode into optimised machine code.
- eBPF allows for custom collection of metrics and in-kernel data aggregation by observation of kernel trace points and function calls.
- It has the potential for real-time security monitoring through inspection of system calls and other kernel processes (yunwei37 and zhaowenjiie, 2024).

Landauer et al., leveraged eBPF for injection of probes and say existing intrusion detection systems do not have enough granularity to monitor kernel activity in enough detail to uncover rootkits which might only target inner functions (Landauer et al., 2025).

2.4. Noise and Prioritization

2.4.1. Sources of Noise

When discussing the idea of noise in relation to the Linux operating system, de Oliveira et al. have a definition which will be useful in the context of trying to account for it with this solution. They define a generalised OS noise as 'the time spent by a CPU executing instructions not belonging to a given application task assigned to that CPU while the task is ready to run' and go on to say its definition is not limited to OS activity but 'any interfering computational process' including those originating from user-space (de Oliveira et al., 2023, p. 199). The given application in this context will be the rootkit.

Kernel code is executed either in process context or interrupt context. In process context the process or thread system calls are switched to be executed under ring zero or privileged kernel mode and executes kernel code itself. It is merely executed 'within the context of a user-space context of a user-space process or thread', meaning there is no kernel thread executing code for them. Large portions of the kernel execute in this context whether that be code of device drivers, processor exceptions, and CPU scheduling are all carried out in process context (Billimoria, 2024).

Interrupt context is entered when hardware interrupts are driven by a peripheral chip. In this case the CPU will save out the current context and run the interrupt handler code also known as the interrupt service routine (ISR). This too will now operate in the privileged kernel mode (Billimoria, 2024).

In the context of rootkit detection through deviations in timing from a normal baseline there is the need to remove 'noise'. This can be considered as data that is not of interest to the observer or analyst. The noise either needs to be removed from the data before analysis or mitigated in the statistical model so as not to skew results (Chandola et al., 2009). Timing-based anomaly detection tends to use 'lumped' timing measurements which due to other processes on the system such as interrupts. Measuring subcomponents can allow for tighter distributions enables better detection removing timing overhead that has been imposed by irrelevant processes according to Carreón et al (Carreón et al., 2020). They evaluate other papers that utilize statistical based timing methods and conclude that in some cases such methods with a granularity at the syscall level can lead to false positives at a rate as much as 10%. However, reading through their sources this seems to be a misreading of the mathematics which equates to false positive rates as low as 0.10%. The paper they refer to by Yoon et al. does however pertain to real-time embedded systems which points out that such systems are more deterministic in terms of execution time (Yoon et al., 2013).

2.4.2. Dynamic Probing

In regard to dealing with negative influence of varying system condition on detection accuracy e.g. dubbed 'noise' in this paper. One suggested approach is to use dynamic probing by waiting for the processor to become idle. The swapper process according Roel Van de Paar hasn't been use for swapping since the 1990's and has been replaced by demand-paged operating systems. In Linux this has traditionally been the idle process, which waits in an infinite loop, so that there is a task ready to be scheduled. Every CPU has a dedicated idle thread named 'swapper'. The schedular if there are no candidates tasks to run pick swapper and context switch the thread to 'next' (Billimoria, 2024). Swapper will essentially be a no-op instruction that is being

run. This may be a target to measure when the CPU is idling and might be explored to use as a filter.

2.4.3. Scheduling and Prioritization

CPUs on a Linux system are treated as resource that is to be shared among users, processes and threads. The schedular is responsible for arbitration on how access is given to the CPU shared among these competing entities and their need for execution.

The operating system performs task scheduling in Linux on a 'Kernel Schedulable Entity' (KSE), which is essentially a task object. Madden states that the scheduler does not discriminate between processes and threads when it comes to scheduling tasks (Madden, 2019). More specifically however, on Linux the KSE is at the level of the thread. Every process contains at least one thread and so it is at this level of granularity which scheduling takes place at (Billimoria, 2024).

Landauer et al have argued that prioritising the probing process, that is, giving higher priority to probing mechanisms may improve the accuracy of detection of timing deltas that are due to rootkit activity. Setting such priorities for probes could be achieved in Linux through the use its scheduling policies. Every thread on a Linux system is associated with a scheduling policy and it is possible to prioritize these through a priority scale, sometimes called a 'nice value', that will affect how much CPU time a process receives by the schedular.

Schedule policies are divided into real-time and normal policies, where real-time policies will always take priority over normal policies (Madden, 2019). Normal policy is that of SCHED_OTHER or SCHED_NORMAL and is always the default. The base value is 0 with a range of -20 to 19 (lower number has priority). Real-time policy flavours are those of SCHED_FIFO, SCHED_RR and SCHED_DEADLINE. Both SCHED_FIFO and SCHED_RR will take precedence over the default and will delay lower priority tasks if required. De Oliveira et al say of them, "The difference between the two is only for threads at the same priority: in this case, SCHED_RR threads are scheduled in a round-robin fashion with a given time slice, while SCHED_FIFO threads release the CPU only on suspension, termination or pre-emption" (de Oliveira et al., 2023, p. 197). SCHED_DEADLINE will also prioritise tasks over normal policy but allow for an estimate of the time a task will take, exceeding it will allow the kernel to suspend the task.

For the design of the artifact two policies will be used in total. One real-time policies will be chosen (SCHED_FIFO) in order to measure their effect in gathering timing related data. One normal policy (SCHED_NORMAL) will be chosen in order to create a baseline with which to compare different timings of chosen syscalls with the real-time policies.

The motivation for testing with SCHED_FIFO is to use a real-time policy that is aggressive and will give it the greatest opportunity for taking continuous measurements while the probes are being executed. SCHED_FIFO 'has in effect, infinite time slice' (Billimoria, 2024, p. 503). Threads running under this policy will only yield the processor in cases where it blocks on I/O, it stops or dies, a higher-priority real-time thread becomes runnable (Billimoria, 2024). It can monopolize the processor however (Madden, 2019). And this may not be what is needed as it will be required that in the case of gathering metrics from the system in the presence of a rootkit the rootkit itself could not be relied upon to set a higher priority for itself and thus may not be able to gain execution time. However, priority values which can be chosen between 1 and 99 will allow for some level of tuning. The less aggressive

SCHED_RR can be tested which uses time slices in a round-robin and will allow for other threads in the run queue guaranteed quanta of time, with timeslices for this policy defaulting to 100 milliseconds (Billimoria, 2024). Real-time scheduling policies in this context are mitigations against pre-emption whereby a currently running thread is ousted by one with a higher priority. Instead, the higher priority could be given to the proposed tracing tool and have its run-queue latency eliminated.

Brendan Gregg, a deeply respected engineer in the field of performance analysis, says of run-queues (aka schedular latency), that on multi-processor systems the kernel 'typically provides a run queue for each CPU, and aims to keep threads on the same run queue. This means that threads are more likely to keep running on the same CPUs'(Gregg, 2020, sec. 6.2.3). This is a strategy known as CPU affinity where a thread is aimed to be kept on a particular core. So, it might be expected that to some degree on typical multicore processors that real-time policies could be used but not starve other processes of execution. It is expected that an iterative loop of instructions will be used for data gathering and increase the likelihood of cache hits, which could increase the likelihood of CPU affinity as 'threads are more likely to keep running on the same CPUs where the CPU caches have cached their data' (Gregg, 2020, sec. 6.2.3).

2.4.4. Scheduling Measurement

Another angle from which scheduling can be looked at is from the point of view scheduling changes that correlate to system call timing measurements. Such data could be used to reduce feature 'noise'. The premise is that schedular activity could aid in contextualising system call timing. And where schedular events are collected simultaneously with monitored system or subsystem calls, events could be analysed for external interference such as context switches or wakeups from irrelevant processes.

Lipp et al., use a median value of repeated executions in order to eliminate system-level noise due to interrupts handling or changes to scheduling (Lipp et al., 2021). Carreón et al., note that significant variation in timing measurements can be incurred due to interrupts, cache misses, and system architecture (Carreón et al., 2020), though their solution is to address it with more fine grained timing measurements on subcomponents as already mentioned.

2.4.5. IRQs

Interrupt requests (IRQs) are potentially another source of noise. These are interrupts requests from one or more devices. They are managed by an interrupt controller that queues and dispatches them and regardless of interrupt execution context they can pre-empt other execution threads. They themselves will run to completion once started. Interrupts can introduce jitter into the execution of non IRQ events precisely because of their ability to pre-empt other threads (de Oliveira et al., 2023). Accounting for them may further refine the attempt to analyse particular system calls without their timing being affected by other contexts as one might expect IRQs to be unpredictable and introduce non-deterministic delay into system calls. The idea being that once a system call is interrupted, it will have its state saved and continuation of its execution would have to wait until the IRQ completes.

3. Research Methodology

3.1. Overview

This study builds on prior rootkit detection methods. In general, the approach follows anomaly-based intrusion detection systems (AIDS). AIDS establishes baselines of normal behaviour using machine learning, statistical-based, or knowledge-based approaches. Deviations from normal behaviour are potentially indicative of potential anomalies. Anomaly-based approaches have the advantage of not relying on previously seen attacks such as with signature-based approaches to detection. Rootkits have the ability to change log data or syscall outputs, but it is difficult for them for them to determine what normal system behaviour should like in the face of detection software that anomaly based (Khraisat et al., 2019). AIDS can be applied to both network intrusion detection and host intrusion detection. The system described here will be host based.

3.2. CARAXES Linux Kernel Module Rootkit

CARAXES - Cyber Analytics Rootkit for Automated and Xploratory Evaluation Scenarios – is a Linux Kernel Module Rootkit (LKM). Designed for academic research, it was developed for rootkit detection based on function timings and will be the rootkit used for testing here. Landauer et al. evaluated other rootkits and found they were not suitable to run on recent Linux kernel versions. Issues encountered with older rootkits tended towards obsolescence through functions Linux no longer supported including removal of certain kernel entities for security reasons (Landauer et al., 2025). CARAXES hide files or processes through hooking either getdents or filldir with a preference for wrapping filldir.

3.3. Rootkit and Tracing Targets

As the focus of this research is to follow on from Landauer et al., the tracepoint targets for timing will also be against syscalls and their internal kernel functions. Because kernel-mode rootkits can hide processes and modules from user-space, measuring inside syscall internals (not just entire syscalls) can help reveal timing deviations. Landauer et al found that many rootkits wrap the getdents function. This target will also be used as the candidate for probe attachment and measurement of timing deltas.

Tracepoints themselves will act as the hook to call a function (probe) provided. Every time the tracepoint is executed the callback function will fire. Data can be written to user space through the use of eBPF maps. Other tracepoint targets will also be used to help analyses potential noise pollution from the rest of the system by tracing subsystem events such as schedular events. Based on the research done in section 2.4. potential subsystem event candidates can be scouted out in: */sys/kernel/tracing/available events*. Each event can be further explored with its associated format file that describes the fields which can be logged for example for the sched_wakeup event is described in: */sys/kernel/tracing/events/sched/sched_wakeup/format*. Linux kernel code is open source and available for inspection also.

3.4. Scheduling Tracepoints

In part, what needs to be achieved are indicators of jitter. It is expected that processes measured during normal execution will form a basis of what should be regarded as consistent scheduling. That is requests to scheduling should be consistent

and predictable, any variance from the baseline will be consider jitter or unwanted latency. Targets identified as relevant tracepoints are listed in the following table.

#	Tracepoint	Description	Possible Usefulness
1.	sched:sched_switch	Trace context switch of task with a timestamp as an aid to look for unrelated context switches.	Rootkit wrapped kernel functions such as filldir64 may get called with context switch timing variation significantly different than from other contexts.
2.	sched:sched_wakeup	Logs a task waking up another	Differences between sched_wakeup and sched_switch could indicate jitter. Or
3.	sched:sched_wakeup_new	Logs task waking for first time.	Could be correlated with rootkits if there are significant difference between normal (N) and rootkit (R).

Table 2.

3.4.1. Sched:sched_switch

From Table 2 what is expected from #1 is to look for statistically significant differences between timing deltas of context switch to the target function deltas for rootkit and normal conditions.

$Let(\delta_i^N) and (\delta_i^R)$ *represent time difference betwen the context switch and the call to* $filldir64$ *for both a normal run* N *and a rootkit run* R.

$$\delta_i^X = f_{\text{gd}}^X\left(t_j^{(i)}\right) - f_{\text{cs}}^X\left(t_i^{(i)}\right), \quad X \in \{N, R\}$$

Average timing difference can then be sought for each of N and R respectively.

$$\overline{\delta^N} = \frac{1}{k}\sum_{i=1}^{k} \delta_i^N, \quad \overline{\delta^R} = \frac{1}{k}\sum_{i=1}^{k} \delta_i^R$$

If there is or isn't significant difference, it may indicate if a rootkit impacts scheduling or simply normal system noise.

$$\Delta = \overline{\delta^R} - \overline{\delta^N}$$

Mann-Whitney will be used at the statistical test. Justification for is dealt with in section 4.2.

3.4.2. Sched:sched_wakeup

Wakeups will occur when a task is added to the kernel's runqueue and becomes a candidate to run on a given CPU core. After a newly created task is created it can be traced via the sched_wakeup_new tracepoint. After their initial creation and having being enqueued on the runqueue data structure threads can cycle through other states such as sleeping or stopped, zombie or dead. Events such those involving I/O can result in threads being put into a sleeping state subsequently they may be rescheduled and tracing the wake of an already existing thread relative to a newly scheduled thread may indicate system noise of non-rootkit activity. Correlations will be evaluated once the data has been gathered.

Brendan Gregg's perf analysis blog describes latency of a task from wakeup to context switch in term of scheduling delay (Gregg, 2017), as does Michael Kerrisk describe latency as the time a task is awoken and the time it is scheduled in (Kerrisk, 2025a). Therefore, there is investigation if there is noticeable interference by comparing baseline and rootkit runs. The logic will be applied in precisely the same manner as it was for sched_switch in section 3.4.1.

3.5. SCHED Policy Statistics

Given the data output summary statistics can be used to check for indications of reduced timing noise between a SCHED_NORMAL run and a SCHED_FIFO run. Basic statistics generated can include standard deviation (STD), median absolute distance (MAD), interquartile range (IQR), and percentiles 95th (P95) and 99th (P99) respectively. Timing deltas on each event types between both runs can help calculate indicators of how noisy each sched policy run is. The following table describes what is being looked for.

Statistic	Description	Indicative Feature
STD	Measures average deviation from the mean.	If the data is less spread out in one policy vs another this might indicate less noise. SCHED_FIFO for instance should be more dedicated to particular processes related to the application and thus reduce events being gathered from the rest of the system.
MAD	Measure the spread of data but is less influenced by outliers.	As above but with the removal of outliers. A low value here might indicate where the noise is. If small, it will indicate noise is in the outliers.
IQR	Looks at data spread in the middle 50% of the data.	Will be a similar indicator to MAD.
P95, p99	Measures below which is the bulk of the data.	Differences in the measure between sched policies may indicate if

		SCHED_FIFO has an impact on outliers in timings.

Table 3.

3.6. Probe Injection

Custom probes are built with eBPF in order to gather data for analysis. Much like Landauer et al, for general timing measurements of the target function an entry and return points such as filldir64-enter and filldir64-return. Other tracepoints are used targeting system events, namely sched_switch, sched_wakeup and irq.

3.7. System Design and Implementation

3.7.1. Architecture of the Data Collector

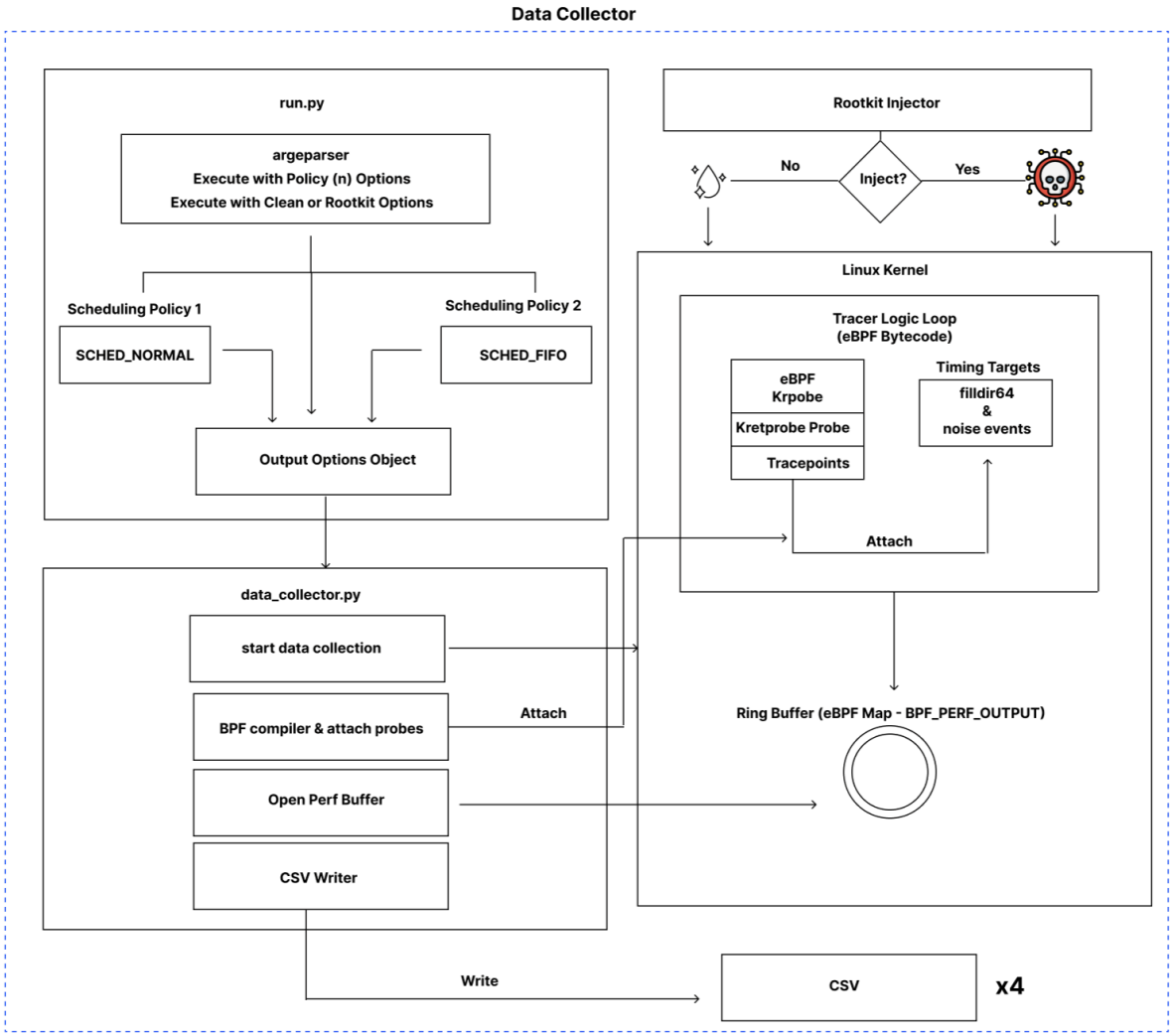

Image 1.

3.7.2. Data Collector Algorithm Overview

Data is collected against a number of targets in the kernel. The filldir64 function is targeted as the subsystem function that will get called by user-space functions such as ls, find, readdir(). Recall that Landauer et al., identified this as part of a sequence of calls related to getdents which all point commonality among the rootkits they researched. Kprobes and Kretprobes are commonly used as guaranteed paired entry and exit points to functions and can be attached almost anywhere in the kernel (Rice, 2023). In this case they are attached to the filldir64 function. This

allows to calculate a delta which are the difference between timestamps applied to both entry and exit points. They have an advantage of being dynamic and allow for attachment within the function body. This is as opposed to tracepoints which need to attach to predefined hooks. BPF is executed on events which include both kprobes and tracepoints.

Tracepoints are used for IRQs and schedular events. Using a kprobe would require extracting event specific information from CPU registers. By using tracepoints event arguments can be accessed directly.

The BPF Compiler Collection (BCC) is used from user-space to attach probes and tracepoints, which are compiled and injected into the kernel. The data collector can be run with the rootkit CARAXES present and active or as a clean run. A shell script is used to trigger functions such as ls or find which will trigger filldir64. In this case ls is used as CARAXES targets it by default.

3.7.3. Dynamic Probing

An attempt was made to implement a dynamic probing mechanism. A BPF array was used to track idle state per CPU by looking for the swapper command name when a sched_switch event occurred. Idle state lookup in the filldir64 kretprobe was used to decide whether to gather data for the target function. However, this sometimes prevented data collection when filldir64 was called.

Instead, the CPU was polled from user space using the python psutil package to get a threshold under which the CPU could be considered ‘idle’. First a threshold was heuristically established with the following Linux top command which was run in parallel to normal system usage and including running the data collector. This is the number 100 minus the system wide idle time sampled once per second. This gave an average of 8.4%. In turn the threshold was decided as 10%.

```
~/Practicum$ while true; do top -bn1 | grep "Cpu(s)" | awk '{print 100 - $8}'; sleep 1; done > cpu_utilisation.txt
```

If the CPU was idle the Linux command ls was called as a subprocess and off the main thread. The reasoning is that the utility function used to measure CPU utilization is blocking. Putting it on another thread allowed for the data collecting probes to operate in parallel. The same logic is not applied to the BPF polling mechanism which prompts the kernel probes to gather data. This preserves the data leaving flexibility for feature engineering and analysis. The idle check around ls execution is, on the other hand, an attempt to reduce unwanted noise when ls executes. When it comes to comparing clean and rootkit data under these conditions it is theorised that the variability of normal system operation is being controlled to some extent. Whether that is sound reasoning or not may have to be revisited at the evaluation stage depending on results.

3.7.4. Setting scheduling policies

A foreign function library is provided by python to provide c compatible data types which functions from the c standard library (docs.python.org, 2025). Following loading of this library the sched_setscheduler system call can be used to set sched policies (Kerrisk, 2025b). SCHED_OTHER also known as SCHED_NORMAL will be run as is an is the default. The other policy will be SCHED_FIFO set with priority of 50 to begin testing with. These will be used to output two data sets without the rootkit present to test the impact of scheduling policies in general. Associated test results are discussed in section 4.1.

4. Results

4.1 SCHED Policy Statistical Results

The following table shows summary statistics for timing deltas comparing SCHED_NORMAL and SCHED_FIFO. This shows the impact of scheduling policies on reducing system noise by way of using SCHED_FIFO to prioritize events generated by the application and reduce those produced by the system as set out in section 3.5.

Statistic	SCHED_NORMAL Clean	SCHED_FIFO Clean	% Change	Interpretation
STD	2939.06	2703.65	~8%	SCHED_FIFO is roughly 8% tighter around the mean. Timing deltas indicate slight reduction in system noise.
MAD	0.00	0.00	0	Timings tend to cluster around the median so indicates little noise in ether scheduling policy.
IQR	0.00	0.00	0	No significant difference in the spread of data between scheduling policies with the bulk of the data.
P95	7252.00	7203.00	~1%	The values are quite high indicating there may be bursts of activity causing jitter. But there is no impact by changing scheduling policies.
P99	12844.00	12812.00	~1.25%	Same interpretation as P99.

Table 4.

In summary, scheduling policies tested: SCHED_NORMAL and the more aggressive SCHED_FIFO showed little variance in typical events, with most values lying around the median and mean. SCHED_FIFO did reduce noise slightly when looking at the STD but taken with the other measure the change seems to be interpretable as quite small.

4.2 Context Switch to Target Function Deltas

A python script was developed to test for differences in deltas in context switch events to filldir64, looking for statistical differences between normal and rootkit runs. The logic for this is described in section 3.4.1.

Firstly, the distribution of the data was examined in context. Deltas were calculated for a clean run and output to a histogram (Image 2). The data does not follow a normal distribution. This should be expected as data is drawn from system-wide processes.

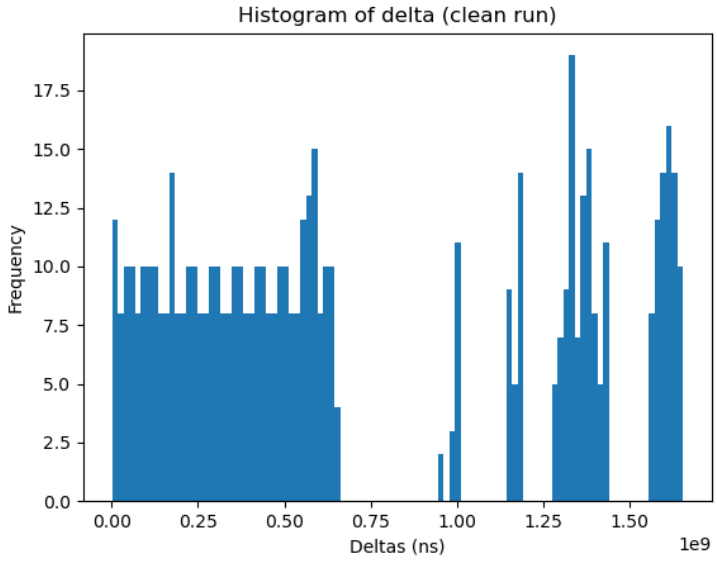

Image 2.

As data is not normally distributed a non-parametric test is required, the equivalent of a non-parametric t-test. There is no one-to-one pairing between data points, and data is gathered from independent samples (the clean run and the rootkit run). A valid test for this is the Mann-Whitney test, which is described as to be used 'when you want to test differences between two conditions and different participants have been used in each condition' (Field et al., 2012). This applies here.

Results from the implementation of the algorithm shows the following.

Measure	**Result**
Clean Run Mean Delta	718630123.45 ns
Rootkit Mean Delta	206897527.76 ns
Delta Difference	-511732595.69 ns
Mann-Whitney U Test	U=121587, p=0.000000

Table 5.

Surprisingly, there is a large difference between clean, and rootkit mean deltas. The average delay between the context switch and the filldir64 function is roughly a half second quicker in the rootkit run. This is unexpected, but the p value being 0.000000 likely means it is so small it has been rounded down. Such a small p value indicates statistical significance and observed measure is incredibility unlikely to have occurred by chance. This is nearly certainly due to the rootkit and may indicate altered scheduling behaviour. The target function seems to run sooner after the context switch. This test shows that the rootkit and clean runs have very different timing delta distributions for context switches to the target function.

4.3 Sched_switch to Sched_wakeup Deltas

The script for generating these deltas was almost identical to the script in section 4.2. with a few minor differences. It picks the last matching sched_wakeup event before the sched_switch and calculates the difference in timestamps taking the wakeup timestamp from the switch timestamp. This follows the previously described task latency from wakeup to context switch as described by Brendan Gregg. The same Mann-Whitney U test was used producing slightly different results.

Measure	**Result**
Clean Run Mean Delta	13196001.14 ns

Rootkit Mean Delta	41735189.65 ns
Delta Difference	28539188.51 ns
Mann-Whitney U Test	U=10495574053.0, p=0.000000

Table 6.

The rootkit is introducing a delay here, unlike the mean delta results in section 4.2. Again, there is statistical significance with the p score. The rootkit appears to have introduced a longer delay from the wakeup to sched_switch, but the times were shorter for the section 4.2 test. One possibility may be that the rootkit is patching target functions. In fact, as an open-source rootkit is being used for testing it is known that it is hooking getdents64 with an 'evil' version. This is typically called in sequence with filldir64 as part of commands like ls in Linux. The code therefore, may execute faster along some paths while introducing overhead in wakeup related parts. Ultimately it is seen that the rootkit is certainly having an impact on the timings. And in this case seems to be introducing more noise.

5. Discussion

Overall comparing SCHED policy statistics gives a picture of an environment that does not appear to have significant differences when it comes to using scheduling policy as a way to prioritize the events and probes within the application and reduce system noise. So, this may not be a sound way to cut down on system noise in an attempt to get more focused data sets. However, widely varying system conditions were not tested, so a larger body of work may need to be done to see if the holds over a variety of workloads and system conditions. In the end there was shown to be marginal gains through using the SCHED_FIFO policy when it came to damping down system noise. However, it was opted to stick with SCHED_NORMAL as this is more reflected of the type of environments that most developers would use Linux in, whether that be in the cloud or on desktop. If real-time policies were to be investigated further, it may be more relevant to the area of high-performance computing.

There was some groundwork done to investigate IRQs with a similar intension to investigate impacts a rootkit may have on its timings and consider them in the context of system noise. However, time did not allow for an implantation. Future work might see how rootkits can change interrupt as to their distribution and frequency and perhaps the time from and interrupt to the next target function e.g. filldir64 or getdents.

The dynamic probing mechanism was not found to work in regard to an implementation in the kernel space code. The alternative was to merely check to see if the CPU was idle. The former may be explored more in the future and later may need greater testing under idle and busy conditions to see its true impacts. However as reasonably good indicators of success were met in detecting anomalies in timing deltas it themselves, it can at least be said not to have had adverse effects on the current solution. Broader experimentation is merely needed.

The measurements made between a system running with and without a rootkit were shown to have impacted timing on the targeted function and tracepoints. The effect was shown to be statistically significant, and it was the technology of eBPF that allowed for a custom solution to be created without the need for kernel programming as probes and tracepoints were injected dynamically. The work here shows indications that using a timing mechanism while accounting for system noise can have results of statistical significance, and in principle it is not tied to a particular function or sub function, nor is it only applicable to one particular rootkit. In order to expand work with the type of prototype built, it should be evaluated with a wide range of kernel level rootkits and increase the range of kernel targets based on appropriate research. Accompanying testing would need to be done under a wide variety of system conditions, and workloads and

whether it is robust enough to detect stealthier rootkits or if they are capable of evading the probing mechanisms. These results show promise as a method for further exploration and may provide the basis on which to build data sets that could contribute positively to the success of machine learning algorithms.

References

BBC News, 2002. UK hacking suspect arrested [WWW Document]. BBC News. URL http://news.bbc.co.uk/2/hi/technology/2270962.stm (accessed 6.19.25).

Billimoria, K.N., 2024. Linux Kernel Programming - Second Edition, 2nd ed. Packt Publishing.

Carreón, N.A., Gilbreath, A., Lysecky, R., 2020. Statistical Time-based Intrusion Detection in Embedded Systems, in: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). Presented at the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 562–567. https://doi.org/10.23919/DATE48585.2020.9116369

Caviglione, L., Mazurczyk, W., Repetto, M., Schaffhauser, A., Zuppelli, M., 2021. Kernel-level tracing for detecting stegomalware and covert channels in Linux environments. Computer Networks 191, 108010. https://doi.org/10.1016/j.comnet.2021.108010

Chandola, V., Banerjee, A., Kumar, V., 2009. Anomaly Detection: A Survey. ACM Computing Surveys 41, 15.1-15.58.

Dawson, J.A., McDonald, J.T., Hively, L., Andel, T.R., Yampolskiy, M., Hubbard, C., 2018. Phase Space Detection of Virtual Machine Cyber Events Through Hypervisor-Level System Call Analysis, in: 2018 1st International Conference on Data Intelligence and Security (ICDIS). Presented at the 2018 1st International Conference on Data Intelligence and Security (ICDIS), pp. 159–167. https://doi.org/10.1109/ICDIS.2018.00034

de Oliveira, D.B., Casini, D., Cucinotta, T., 2023. Operating System Noise in the Linux Kernel. IEEE Transactions on Computers 72, 196–207. https://doi.org/10.1109/TC.2022.3187351

Dittrich, D., 2002. Wayback Machine [WWW Document]. web.archive.org. URL https://web.archive.org/web/20111120005404/http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq (accessed 6.18.25).

docs.python.org, 2025. ctypes — A foreign function library for Python [WWW Document]. Python documentation. URL https://docs.python.org/3/library/ctypes.html (accessed 8.9.25).

Ezeme, O.M., Mahmoud, Q.H., Azim, A., 2022. A Framework for Anomaly Detection in Time-Driven and Event-Driven Processes Using Kernel Traces. IEEE Transactions on Knowledge and Data Engineering 34, 1–14. https://doi.org/10.1109/TKDE.2020.2978469

Field, A., Miles, J., Field, Z., 2012. Discovering Statistics using R. SAGE Publishing Ltd., London.

Forrest, S., Hofmeyr, S., Somayaji, A., 2008. The Evolution of System-Call Monitoring, in: 2008 Annual Computer Security Applications Conference (ACSAC). Presented at the 2008 Annual Computer Security Applications Conference (ACSAC), pp. 418–430. https://doi.org/10.1109/ACSAC.2008.54

Gregg, B., 2020. Systems Performance, 2nd Edition, 2nd ed. Pearson.

Gregg, B., 2017. perf sched for Linux CPU scheduler analysis [WWW Document]. brendangregg.com. URL https://www.brendangregg.com/blog/2017-03-16/perf-sched.html (accessed 7.28.25).
halflife, 1996. Abuse of the Linux Kernel for Fun and Profit [WWW Document]. phrack.org. URL https://archives.phrack.org/issues/50/5.txt (accessed 6.19.25).
Joy, Jestin, John, A., Joy, James, 2011. Rootkit Detection Mechanism: A Survey, in: Nagamalai, D., Renault, E., Dhanuskodi, M. (Eds.), Advances in Parallel Distributed Computing. Springer, Berlin, Heidelberg, pp. 366–374. https://doi.org/10.1007/978-3-642-24037-9_36
Kerrisk, M., 2025a. trace-cmd-report(1) - Linux manual page [WWW Document]. man7.org. URL https://man7.org/linux/man-pages/man1/trace-cmd-report.1.html (accessed 7.28.25).
Kerrisk, M., 2025b. sched_setscheduler(2) - Linux manual page [WWW Document]. man7.org. URL https://www.man7.org/linux/man-pages/man2/sched_setscheduler.2.html (accessed 8.9.25).
Khraisat, A., Gondal, I., Vamplew, P., Kamruzzaman, J., 2019. Survey of intrusion detection systems: techniques, datasets and challenges. Cybersecur 2, 20. https://doi.org/10.1186/s42400-019-0038-7
Landauer, M., Alton, L., Lindorfer, M., Skopik, F., Wurzenberger, M., Hotwagner, W., 2025. Trace of the Times: Rootkit Detection through Temporal Anomalies in Kernel Activity. https://doi.org/10.48550/arXiv.2503.02402
Levine, J.F., Grizzard, J.B., Owen, H.L., 2006. Detecting and categorizing kernel-level rootkits to aid future detection. IEEE Security & Privacy 4, 24–32. https://doi.org/10.1109/MSP.2006.11
Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., Gruss, D., 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86, in: 2021 IEEE Symposium on Security and Privacy (SP). Presented at the 2021 IEEE Symposium on Security and Privacy (SP), pp. 355–371. https://doi.org/10.1109/SP40001.2021.00063
Luckett, P., McDonald, J.T., Dawson, J., 2016. Neural Network Analysis of System Call Timing for Rootkit Detection, in: 2016 Cybersecurity Symposium (CYBERSEC). Presented at the 2016 Cybersecurity Symposium (CYBERSEC), pp. 1–6. https://doi.org/10.1109/CYBERSEC.2016.008
Madden, M.M., 2019. Challenges Using Linux as a Real-Time Operating System [WWW Document]. ntrs.nasa.gov. URL https://ntrs.nasa.gov/api/citations/20200002390/downloads/20200002390.pdf (accessed 7.14.25).
Morag, A., 2021. Advanced Persistent Threat Techniques Used in Container Attacks. Aqua. URL https://www.aquasec.com/blog/advanced-persistent-threat-techniques-container-attacks/ (accessed 6.16.25).
Morag, A., Revivo, I., 2024. perfctl: A Stealthy Malware Targeting Millions of Linux Servers. Aqua. URL https://www.aquasec.com/blog/perfctl-a-stealthy-malware-targeting-millions-of-linux-servers/ (accessed 6.16.25).
Nadim, M., Lee, W., Akopian, D., 2023. Kernel-level Rootkit Detection, Prevention and Behavior Profiling: A Taxonomy and Survey. https://doi.org/10.48550/arXiv.2304.00473
Rice, L., 2023. Learning eBPF. O'Reilly Media, Inc.
Saur, K., Grizzard, J.B., 2010. Locating ×86 paging structures in memory images. Digital Investigation 7, 28–37. https://doi.org/10.1016/j.diin.2010.08.002

Sidecar Containers [WWW Document], 2025. . Kubernetes. URL https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/ (accessed 6.23.25).

Stühn, J., Hilgert, J.-N., Lambertz, M., 2024. The Hidden Threat: Analysis of Linux Rootkit Techniques and Limitations of Current Detection Tools. Digital Threats 5, 1–24. https://doi.org/10.1145/3688808

Yoon, M.-K., Mohan, S., Choi, J., Kim, J.-E., Sha, L., 2013. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems, in: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). Presented at the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 21–32. https://doi.org/10.1109/RTAS.2013.6531076

yunwei37, zhaowenjiie, 2024. eBPF Tutorial by Example [WWW Document]. GitHub. URL https://github.com/eunomia-bpf/bpf-developer-tutorial/blob/main/src/0-introduce/README.md (accessed 7.8.25).