Auditing Frameworks Need Resource Isolation: A Systematic Study on the Super Producer Threat to System Auditing and Its Mitigation

Peng Jiang, Ruizhe Huang, Ding Li, Yao Guo, and Xiangqun Chen,
MOE Key Lab of HCST, School of Computer Science, Peking University;
Jianhai Luan, Yuxin Ren, and Xinwei Hu, *Huawei Technologies*

https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-peng

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

Auditing Frameworks Need Resource Isolation: A Systematic Study on the Super Producer Threat to System Auditing and Its Mitigation

Peng Jiang[1], Ruizhe Huang[1], Ding Li[1*], Yao Guo[1], Xiangqun Chen[1], Jianhai Luan[2], Yuxin Ren[2] and Xinwei Hu[2]

[1]MOE Key Lab of HCST, School of Computer Science, Peking University
[2]Huawei Technologies
[*]Corresponding author

Abstract

System auditing is a crucial technique for detecting APT attacks. However, attackers may try to compromise the system auditing frameworks to conceal their malicious activities. In this paper, we present a comprehensive and systematic study of the super producer threat in auditing frameworks, which enables attackers to either corrupt the auditing framework or paralyze the entire system. We analyze that the main cause of the super producer threat is the lack of data isolation in the centralized architecture of existing solutions. To address this threat, we propose a novel auditing framework, NODROP, which isolates provenance data generated by different processes with a threadlet-based architecture design. Our evaluation demonstrates that NODROP can ensure the integrity of the auditing frameworks while achieving an average 6.58% higher application overhead compared to vanilla Linux and 6.30% lower application overhead compared to a state-of-the-art commercial auditing framework, Sysdig across eight different hardware configurations.

1 Introduction

Auditing frameworks, such as Linux Audit [79] or Sysdig [20], play a vital role in provenance analysis systems for enterprise security. Many companies build Security Operations Centers (SOC) [2, 6, 9, 10, 15, 25, 35] based on auditing frameworks. Moreover, a large body of research work uses auditing frameworks to develop intrusion detection systems [6,16,18,19,27,38,41–43,46,56,90,93,108]. We expect this topic to remain relevant for both industry and academia due to the amount of related work.

Unsurprisingly, attackers are engaged in compromising auditing frameworks. Recent studies have also shown that attackers compromise the kernel module of auditing frameworks to prevent attack traces from being recorded. Paccagnella *et al.* [86] proposed a race condition attack in which an attacker with root privileges can compromise the kernel module of auditing frameworks to hide their malicious actions. To protect the kernel module of auditing frameworks, researchers have proposed multiple approaches, such as Hardlog [14], KennyLoggings [86], and QuickLog [45]. There are also many user-space attacks that mainly involve log tampering [21,24,39,81]. Meanwhile, several cryptographic-based approaches have been proposed to secure the user space transmission and storage of logs [8,28,52,85].

In addition to the vulnerabilities mentioned above, this paper concentrates on the *super producer threat*. This threat does not require attackers to have root privileges to launch attacks on the kernel module. In essence, the attacker can either disable provenance data collectors or intensify DoS attacks on the system under observation by creating a super producer, a process that produces a large number of system provenance events in a brief span of time, with user privilege.

Specifically, by generating a large amount of provenance data that auditing frameworks cannot process in time with a reasonable amount of system resources, a super producer puts the current auditing framework into the *data integrity vs. efficiency* dilemma. On one hand, auditing frameworks may adopt the *pro-performance* strategy [20,26,79], which drops system events if there is too much provenance data. However, under this strategy, attackers may launch the *Provenance Denial of Service (PDoS) attack*, in which the attacker uses the super producer to evict the events of malicious behaviors from the event buffers of auditing frameworks. On the other hand, auditing frameworks may adopt the *pro-integrity* strategy [1], which elastically allocates sufficient resources to ensure that all provenance data can be processed in time. Unfortunately, this strategy may lead to the *Provenance Assisted Denial of Service (PADoS) attack*, in which attackers exploit the pro-integrity strategy to degrade the performance of the whole system, amplifying the DoS attack on the server. Notably, the PADoS attack can even break the protection of `cgroup`.

The main reason for the super producer threat is that the design of existing auditing frameworks breaks the resource and logic isolation of processes, which is critical for modern OSes to achieve high performance for concurrent tasks. Existing solutions collect system events by intercepting system calls in the OS kernel and then processing the collected events in a

centralized user-space collector [1,20,26,79]. The centralized collector handles provenance data equally regardless of the priority, importance, and resource quota of the processes that generate the provenance data. Therefore, a super producer can generate a massive amount of provenance data that occupies all the processing power of the centralized handler and causes the "data integrity vs. efficiency dilemma".

In this paper, we present a comprehensive analysis of the super producer threat and propose a novel auditing framework, NODROP, that balances the trade-off between "data integrity and efficiency". Specifically, NODROP surpasses existing solutions in two aspects. First, it guarantees the integrity of provenance data. NODROP records all provenance data faithfully regardless of the workload. Therefore, a super producer cannot conceal the traces of attacks by generating too many system events, thus mitigating the PDoS attack. Second, NODROP prevents the super producer from degrading the performance of the whole system, avoiding the PADoS attack.

The key insight of our design is to provide isolation to the provenance data collector so that each process consumes its own resource quota to handle the provenance data generated by itself. The logic behind this design is as follows. Like user-space log events (e.g., log4j events), provenance events reflect the status of the running processes. Thus, provenance data should be considered as the logs of the corresponding processes, instead of the OS. Therefore, each process should spend its own resource quota to handle the provenance data it generates.

By isolating the provenance data of each process, we can naturally mitigate the super producer threat. This way, a process that generates a huge amount of provenance data in a short time can only affect its own performance, since it has a limited resource quota. It also cannot interfere with the processing power of other processes. Therefore, the super producer cannot stop the system from recording provenance events of attacks by overwhelming the auditing frameworks.

The main challenge of isolating provenance data from different processes is to achieve efficiency. We use a *threadlet-based* approach that inserts the provenance data processing logic into the memory of running applications. This approach leverages the process isolation strategy of the OS directly, eliminating the extra overhead of adding a new isolation strategy to the auditing framework. This approach also reduces process scheduling and its associated cache miss costs.

We thoroughly evaluate NODROP with eight different hardware configurations and five baselines. Our evaluation shows that NODROP faithfully records all provenance data, preventing the PDoS attack. On the contrary, current pro-performance auditing frameworks (i.e., Sysdig) can drop up to 90% of provenance data while the super producer is running. NODROP can also prevent the PADoS attack. Specifically, NODROP only slows down three popular applications by 4.0% on average, regardless of the workload generated by the super producer. For comparison, existing pro-integrity auditing frameworks can slow down the applications by up to 59.1% on average. More importantly, when the super producer increases its workload, the application performance decreases accordingly with existing auditing frameworks. NODROP is also efficient compared with the SOTA pro-performance collector, Sysdig. NODROP has, on average, 6.30% less application overhead than Sysdig. In summary, our evaluation proves that NODROP can address the super producer threat while incurring lower system overhead than existing auditing frameworks.

To sum up, this paper makes the following contributions:

- To the best of our knowledge, this is the first thorough systematic study on the super producer threat and related attacks to auditing frameworks.
- We identify that the root cause of the super producer threat is the lack of resource isolation in the user space component of existing auditing frameworks.
- To address the super producer threat, we propose a novel auditing framework NODROP that efficiently isolates resources for provenance data handling by enforcing processes to consume their own resource quota to handle the provenance data generated by themselves.
- Extensive experiments demonstrate that NODROP can address the super producer threat, as well as its efficiency.

Availability: NODROP is available at: https://github.com/PKU-ASAL/NoDrop

2 Background

Auditing frameworks are the fundamental part of provenance analysis [57], which is a technique that monitors system activities to detect and investigate attacks [29,31,97,98]. Popular auditing frameworks include Sysdig [20], LTTng [26], and Linux Audit [79]. These three auditing frameworks are the most widely cited in provenance-based detection solutions [29,31,36,41,43,48,55,60,61,97,98,105,106,108,109]. Besides, there are more recent auditing frameworks from academia, including Camflow [1,90], Hardlog [14], Kenny-Loggings [86], and QuickLog [45]. We thoroughly investigate the auditing frameworks published in industry and academia in recent years. We summarized the auditing frameworks in Table 1.

The existing auditing framework has a "centralized" architecture [1,20,26,79], as Figure 1 shows. This framework intercepts provenance events through a kernel module and digests them based on user-specified rules (e.g., sending the provenance data to a remote log server or storing it to local files) in a centralized user-space module, called `collector`.

Table 1: A comparison of auditing frameworks. * *means the collector is implemented by us. Per-core thread means each CPU core has a dedicated processing thread for provenance data. Per-core buffer means each CPU core has a dedicated event buffer.*

Name	Computation isolation	Data isolation	Synchronization	Strategy
Sysdig [29, 31, 36, 97, 98, 106]	Single thread	Per-core buffer	Asynchronous	pro-performance
Linux Audit [41, 43, 48, 108, 109]	Single thread	Single buffer	Asynchronous	pro-performance
LTTng [55, 60, 61, 105]	Single thread	Per-core buffer	Asynchronous	pro-performance
Camflow [40, 90, 91]	Per-core thread	Per-core buffer	Asynchronous	pro-integrity
KennyLoggings [86]	Single thread	Single buffer	Asynchronous	pro-performance
Hardlog [14]	Single thread	Single buffer	Synchronous	pro-performance
QuickLog [45]	Single thread	Single buffer	Asynchronous	pro-performance
Sysdig-Camflow*	Per-core thread	Per-core buffer	Asynchronous	pro-integrity
Sysdig-Integrity*	Single thread	Per-core buffer	Synchronous	pro-integrity
NODROP *	Per-thread threadlet	Per-threadlet buffer	Synchronous	performance and integrity

Current auditing frameworks mainly differ in how they handle massive amounts of provenance data. They adopt two strategies: pro-performance and pro-integrity.

Several solutions, such as Sysdig [20], Linux Audit [79], and LTTng [26], follow the "pro-performance" strategy. The rationale behind this strategy is that the auditing framework should minimize the system run-time overhead and maintain the performance of critical services on the monitored host. Current solutions limit the CPU usage of the auditing framework by allowing only *one collector thread*. If this thread cannot process all the provenance events in time, it will either stop receiving events or drop them.

Solutions that adopt the "pro-integrity" strategy [1, 90] try to allocate enough resources to the auditing framework to handle all the provenance events. For instance, Camflow [90] uses a multi-threaded model that dynamically allocates computational resources based on the provenance data generation speed.

3 The Super Producer Threat

The existing auditing frameworks use a centralized architecture that exposes them to the super producer threat. This threat occurs when an attacker exploits a super producer to consume the computational resources of other processes, breaking the logic and resource isolation between them. As a result, current auditing frameworks face a dilemma between data integrity and efficiency.

Figure 1 illustrates the super producer threat and the "data-integrity vs. efficiency" dilemma. The figure shows three user-space applications (the super producer, the malware, and the Nginx server) and an auditing framework that processes all the provenance data of these applications. The arrows indicate the direction of provenance data flow, and the width of the arrows reflects the amount of provenance data.

The super producer produces considerable system provenance data that exhausts the collector's computation capacity. As a result, the collector either drops the provenance data of other applications or competes for more computational resources, implicitly breaking the resource quota of each application. Thus, it becomes feasible to exploit PDoS and PADoS attacks.

The pro-performance strategy restricts the resource quota of the collector to prevent performance degradation of the whole system [20, 26, 79], but this exposes the system to the PDoS attack. Figure 1 shows that the collector's limited resources cannot cope with the high rate of provenance data generation by the super producer, and the collector will drop events when overloaded. Moreover, the collector does not separate the provenance data from different applications, so other critical provenance events of the malware may be evicted, enabling the attackers to conceal the malware from detection.

The pro-integrity strategy gives the collector more resources to prevent the loss of provenance data [40, 91], but this exposes the system to the PADoS attack. Figure 1 shows how the collector consumes more resources to handle all provenance events, while the resources of other applications are reduced accordingly, resulting in significant performance interference for the whole system. Moreover, since the super producer indirectly affects the system performance by using the collector, it only requires moderate resources to generate a large amount of provenance data. Hence, existing isolation mechanisms (e.g., `cgroups`), which limit the resource usage of the super producer, cannot effectively stop the PADoS attack.

3.1 Research Challenges

Addressing the super producer threat is conceptually challenging. One possible solution is to suppress the super producer's generation speed of provenance data with some specified threshold. However, this strategy is not systematic. First, it is hard to set an effective threshold considering the dynamics of the systems. Second, attackers may use a set of super-producer processes to avoid reaching the threshold.

Another straightforward solution is to provide isolation in-

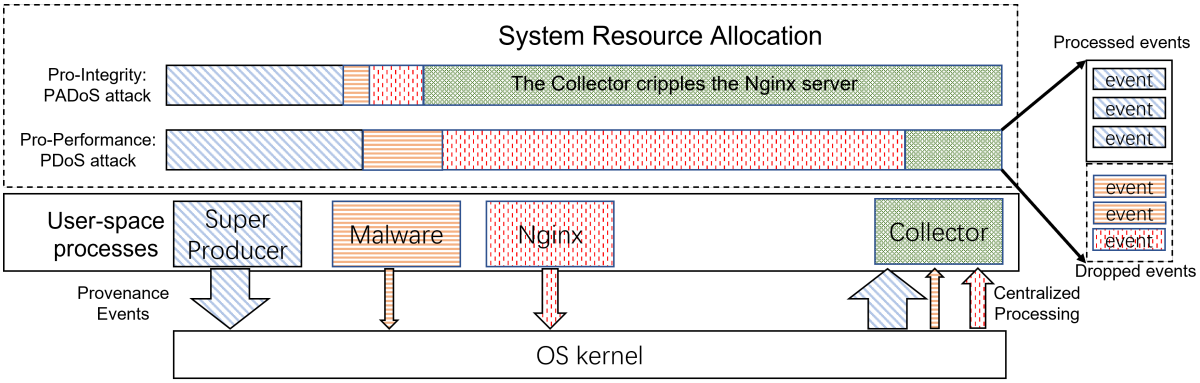

Figure 1: The design of existing auditing frameworks and the "data integrity vs. efficiency dilemma".

side the user-space collector. However, this strategy requires complex user-space logic in the collector, increasing the runtime overhead and difficulty in adapting to different systems. Note that simply providing a separate event buffer for different processes is not sufficient because other computational resources, such as the CPU, also need to be isolated. More importantly, resource isolation or scheduling policy inside the collector may be inconsistent or conflict with the original policy made by the OS. Thus, different policies interfere with each other, causing all of them ineffective.

In summary, we need to redesign the auditing framework architecture, which can adaptively suppress the super producer, isolate provenance data, avoid performance interference, and respect the OS resource management policies.

3.2 Attack Scenario and Threat Model

We consider a common scenario of a multi-tenant web server [7] as the potential context for PDoS and PADoS attacks. We suppose that two Internet-facing applications, the *target app* and the *victim app*, from different users, are running on the server at the same time. These two applications have their own resource quota (i.e., in separate `cgroups`). An auditing framework is running to monitor both the *target app* and the *victim app*. For the PDoS attack, the attacker's objective is to disable the auditing framework by attacking the *victim app*. Then, the attacker can try to compromise the *target app* without leaving traces in the provenance data. For the PADoS attack, the attacker's aim is to paralyze the *target app* by compromising the *victim app*.

We define our threat model as follows: the attacker can transform the *victim app* into a super producer. This does not imply that the attacker has to compromise the *victim app*. It is enough to generate a large number of requests for a complex dynamic web application. Moreover, we assume that the attacker is aware of the auditing framework that is deployed on the server.

We also make the following assumptions about the security of the auditing frameworks. First, the kernel modules that collect provenance data are not vulnerable to attacks. Second, the provenance data is stored and transmitted securely and reliably. Third, the user space module that analyzes the provenance data is protected by existing intrusion detection systems that can alert us if the attacker tries to disable or compromise the auditing frameworks [20, 26]. The protection of the kernel module, the transmission, and the storage of provenance data is beyond the scope of this paper.

4 Design of NODROP

This section describes the design of NODROP, which aims to prevent PDoS and PADoS attacks by attacks by achieving thread isolation for provenance data processing. Similar to Sysdig, NODROP captures system call, thread switching, and system signal events, and allows users to provide custom logic for processing these events.

4.1 Design Goals

NODROP is designed to address the super producer threat by satisfying the following properties:

G1 - Zero Data Lost. NODROP must record all provenance data generated by the system to prevent PDoS attacks.

G2 - Performance Isolation. NODROP must ensure that a super producer cannot affect overall system performance, preventing attacks such as PADoS.

G3 - Low Overhead. NODROP should not introduce significantly higher costs than existing auditing frameworks.

4.2 Design Principles

NODROP is guided by two design principles: self-consuming execution and synchronized logging buffer.

Self-consuming execution. With NODROP, each thread processes its own generated provenance data. This differs from the centralized processing architecture of current auditing frameworks and addresses the super producer threat by achieving resource and data isolation. Each thread uses its own resource quota to process its provenance data in a dedicated buffer, preventing a super producer from evicting other

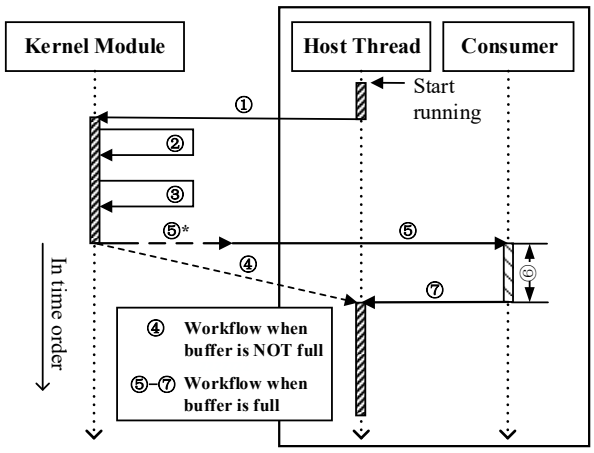

Figure 2: The workflow of NODROP. ① - ⑦ are the seven steps of NODROP to handle a system call event.

threads' system events or slowing them down by occupying their resource quota.

Synchronized logging buffer. NODROP allocates a dedicated logging buffer for each thread and dynamically instruments provenance data processing logic into each thread when the buffer is full. This synchronized strategy ensures no provenance data is overwritten or lost, regardless of the super producer's data volume.

These principles are realized through a threadlet-based approach. A threadlet is a self-contained piece of code that exists in the memory space of a host thread. NODROP instantiates provenance data processing logic as a threadlet, inserts it into the memory of the currently running thread, and consumes the provenance data stored in the dedicated buffer. This type of threadlet is referred to as a consumer throughout the paper.

The threadlet-based design of NODROP offers three advantages: (1) It allows NODROP to use each running thread's resource quota to process the provenance data generated by that thread, without breaking the OS's resource isolation mechanism or slowing down the entire system. (2) Synchronized insertion of a threadlet into a running thread's memory space provides data isolation, preventing a super producer from manipulating or overwriting provenance data in other threads. (3) The threadlet-based architecture is more resource-efficient and lightweight than a conventional thread, eliminating most thread switch overhead such as scheduling delay, cache movement, and priority inversion.

In summary, the threadlet-based architecture satisfies three requirements: ① all provenance data is collected; ② a super producer cannot slow down the entire system; and ③ overhead is optimized.

4.3 High-Level Workflow

NODROP consists of two main components: a kernel module and a user-space consumer. Like other auditing frameworks, the kernel module intercepts system calls to collect provenance data and stores it in a dedicated logging buffer. The consumer, implemented as a threadlet, processes the provenance data. In NODROP, provenance data includes three types of system events: system call, thread switching, and system signals. This definition is the same as that used by Sysdig [20].

The workflow of how NODROP handles system calls is shown in Figure 2. When a thread invokes a system call, the kernel captures it (①) and executes it (②). The kernel module of NODROP then catches and records the system call in a dedicated in-kernel logging buffer (③). If the buffer is not full, NODROP returns control to the host thread (④). If the buffer is full or the thread exits, control is passed to the consumer (⑤). Before transferring control, NODROP checks if there is a consumer in the running thread's memory. If not, it first instruments the consumer into the running thread(⑤*). Once the consumer has control, it uses the running thread's resource quota to process the transferred provenance data, providing performance and data isolation between threads (⑥). When processing is complete, control is returned to the original thread (⑦). The workflow for handling thread switching and system signals differs from that for system calls only in how the kernel is entered (① in Figure 2); all other steps are the same.

4.4 Design Details

This section describes the detailed design and implementation of key components in NODROP. Since NODROP 's kernel module is similar to that of Sysdig [20], its details are not discussed. Instead, the focus is on the unique design of the in-kernel logging buffer (§4.4.1), the user-space consumer (§4.4.2), and consumer instrumentation (§4.4.3).

4.4.1 In-Kernel Logging Buffer

The in-kernel logging buffer is used to store provenance data in the kernel, improving the efficiency of auditing frameworks. When the buffer is full, the consumer consumes provenance data from the front end. Meanwhile, the kernel module pushes incoming provenance data to the end of the buffer until the buffer is full.

NODROP uses a per-thread buffering scheme, allocating one logging buffer for each thread. This is chosen to meet design goal G2, which requires isolation of provenance data from each thread. This buffering scheme naturally distinguishes provenance data from every thread.

In NODROP, each system event contains the metadata and the argument data. The metadata includes the basic attributes of a system event, such as its type, timestamp, and size of the system event. The argument data is distinct for different types of system events. For system call events, each element in the argument data is the value of a system call parameter. If the size of the parameter values is large (e.g., the content of `read`), NODROP follows the practice of Sysdig by truncating

the parameter values that are larger than 80 bytes. For thread switching, the argument data contains the IDs of the previous thread and the next thread, respectively. For signals, the argument data contains the signal ID and the process ID, which captures the signal.

4.4.2 The Consumer

The consumer is designed as a user-defined threadlet function for processing provenance data. It has one parameter that points to a copy of the in-kernel logging buffer in user-space and returns nothing. The key task for NODROP is to ensure the efficiency and security of the consumer. To achieve this, the conventional threadlet is improved.

Kernel interaction. To efficiently and correctly pass the in-kernel logging buffer to the consumer, NODROP uses `mmap` to directly map the buffer from the kernel to user space. This avoids the overhead of copying data and restricts the consumer from accessing other in-kernel memory.

Memory protection. The key challenge for the consumer is to ensure that it cannot be compromised by the host process. Since the host thread or other parallel threads are in the same process as the consumer, they could potentially modify the consumer's data. To prevent this, NODROP uses a comprehensive approach that combines address space randomization, an isolated heap, and MPK to protect the consumer. Figure 3 summarizes the memory protection mechanisms used by NODROP. The consumer is located in a randomized memory region with a dedicated protection key. This region includes a separate stack, heap, and mapped logging buffer.

First, NODROP randomizes the consumer's address to prevent attackers from obtaining it, making it more difficult to compromise the consumer. When the consumer is first instrumented, NODROP randomly chooses the threadlet's loading address, preventing attackers from obtaining the consumer's address before the threadlet is run.

Second, NODROP allocates a dedicated heap and stack for the consumer to prevent memory-overflow-based attacks (e.g., ROP) from the host thread. Isolating the heap is challenging because an attacker could potentially compromise the heap allocator that allocates memory from the consumer's dedicated heap region. To prevent this, NODROP pre-loads a customized heap allocator for the consumer that always allocates memory from the dedicated heap.

Finally, NODROP enforces memory isolation using MPK, a hardware primitive that achieves in-process memory isolation by controlling protection keys and their permissions with a thread-local register called `PKRU` [89, 107]. To protect the consumer, NODROP binds a dedicated `key` to all consumer memory. By updating the `PKRU` value, access permission for that key is enabled at the consumer entry point and disabled after it exits. Similar to previous approaches using MPK [89, 107], NODROP ensures that the host thread cannot modify the `PKRU` register by scanning executable code to validate

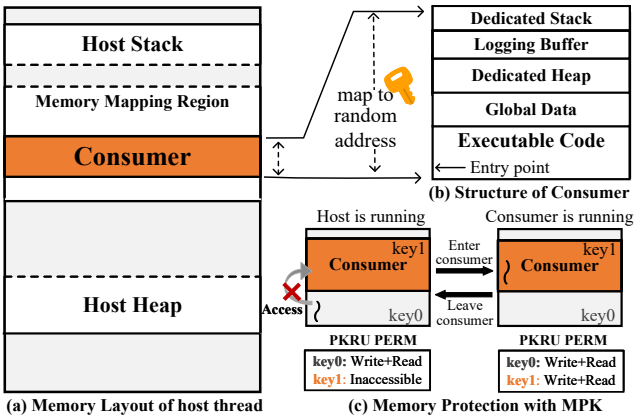

Figure 3: Memory layout of the host thread (*a*) and structure of consumer (*b*). *The consumer is mapped at a random address in the `mmap` region of the host thread. The consumer has its own 5 MPK-protected sections: executable code, global data, dedicated heap plus stack, and the logging buffer mapped from the kernel.*

that it contains no `PKRU`-related instructions. In NODROP, all consumers in the same process share the same key. Since the total number of MPK keys is limited, enough keys are left for later use. Using one MPK key does not compromise NODROP 's security because MPK protection is thread-local [54, 89].

Privilege escalation. Sandbox or container threads may have limited execution privileges. However, suppose the consumer uses the sandbox or container threads as host threads and inherits the limited privilege. In that case, it cannot perform necessary processing on the provenance data, such as writing the data to files or network [16, 18, 38, 46, 56, 90, 93]. To this end, NODROP elevates the consumer's privileges by raising the `RLIMIT` and `CAPACITY` to unlimited and disabling the `SECCOMP`. NODROP recovers these privileges when switching back to the sandbox or container thread.

Atomic execution. Because of the privilege escalation, the sandbox or container thread may abuse the higher privilege of the consumer, thus breaking the system security policy. To avoid such abuse and other data-racing between the consumer and the application thread, NODROP ensures the atomicity of the consumer execution by disabling all signals while the consumer is running. NODROP delays the upcoming signals until the consumer exits. NODROP enables all signals when switching back to the sandbox or container thread. At this moment, NODROP delivers the delayed signals to their handlers of the host thread. Therefore, there is never interleaved execution between the consumer and the sandbox or container thread.

4.4.3 Consumer Instrumentation

NODROP blocks the running thread and invokes the consumer (⑤ in Figure 2) when either the in-kernel logging buffer is full or the running thread exits. The instrumentation process of the consumer is described in detail here.

Threadlet initialization. NODROP prepares and initializes the consumer execution environment when the running thread traps into the kernel for the first time (① in Figure 2). Specifically, NODROP allocates the space of the in-kernel buffer and a per-consumer control block. The control block stores the meta information describing a consumer, including its running state, the address of the user-defined entry function, memory layout, and protection key.

Consumer loading. When the consumer is first instrumented, NODROP loads its binary(⑤* in Figure 2). The loading process is similar to that of the `execve` system call. The kernel module reads and parses the consumer's ELF file, allocates memory space, and loads all segments into memory, as shown in Figure 3(b). Additionally, NODROP reserves an MPK key and initializes the consumer control block. This loading phase only occurs once for each thread, so its overhead does not significantly degrade system performance. Furthermore, existing in-memory template caching techniques can be leveraged to further optimize this cost [96].

Consumer invocation. When invoking the consumer (⑤ in Figure 2), NODROP's kernel module maps the logging buffer, saves the register context of the running thread, updates `PKRU` value, elevates the privilege and finally upcalls to the consumer's entry point.

Consumer exit. We add a new system call that the consumer uses to exit (⑦ in Figure 2). This system call does the reverse of consumer invocation. Concretely, it releases the logging buffer, recovers the privilege configuration, restores the `PKRU` and other registers, and lastly, switches back to the instrumented thread.

5 Evaluation

In this section, we evaluate whether NODROP can address the super producer threat without introducing significantly higher system overhead than existing approaches. Specifically, we focus on the following questions:

- RQ 1: Can NODROP avoid dropping provenance data?
- RQ 2: Can NODROP prevent a super producer from slowing down other applications?
- RQ 3: What is the run-time overhead of NODROP?
- RQ 4: Can data reduction techniques address the super-producer threat?
- RQ 5: Can increasing the buffer size address the super-producer threat?

To ensure the generalizability of our evaluation, we run experiments on four hardware configurations: 1 CPU core with 2GB memory (C1 and C5), 4 CPU cores with 8GB memory (C2 and C6), 16 CPU cores with 32GB memory (C3 and C7), and 32 CPU cores with 64 GB memory (C4 and C8). We also conduct our experiments on both physical and virtual machines, resulting in eight different configurations in total. C1-C4 represent VM configurations, while C5-C8 represent PM configurations. All machines run OSes of Ubuntu 18.04. NODROP is also tested on other Linux distribution like openEuler [82] 20.03 which shows the similar results.

5.1 RQ 1: Event Drop

To answer this research question, we first conducted a controlled measurement study on the number of provenance events dropped by NODROP and pro-performance solutions such as Sysdig, LTTng, and Linux Audit. Then, we simulated a realistic web server to evaluate how well NODROP can prevent PDoS attacks.

In the controlled experiment, we launched a bash script to mimic a super producer that generates a large amount of provenance data in a short period. The script forks n processes, where n is the number of CPU cores, and each process repeatedly invokes the `write` system call and performs the `count++` operation. We adjusted the proportion of the `write` and `count++` operations to control the generation speed of system call events over 30 seconds. We ran the experiment on four hardware configurations, on both virtual and physical machines. The percentage of dropped events represents the potential success rate of a PDoS attack.

Figure 4(a), 4(b), 4(c), and 4(d) show the results for cases with 1 CPU core and 2GB memory, and 32 CPU cores with 64 GB memory, on both virtual and physical machines. The results for other configurations are similar to those shown in Figure 4, but due to space limitations, we have included them in our GitHub repository. The x-axis represents the number of system call events generated by the kernel per second, while the y-axis represents the number of system call events processed per second by the user-space component. The gap between y and x represents the number of events dropped. The diagonal line in the figure is the ideal line, indicating that no events have been dropped. The blue dot-dashed line and magenta dash line represent Sysdig and LTTng, respectively. We have omitted the line for Linux Audit for clarity, as it drops nearly all events while the super producer is running.

Our evaluation shows that NODROP drops ZERO events while a super-producer is running. In Figure 4, the line for NODROP overlaps with the diagonal line, indicating that no events have been dropped. In contrast, existing pro-performance solutions drop most of the generated system events, allowing for a high success rate for PDoS attacks. Specifically, on machines with 1 CPU core and 2 GB memory, Sysdig drops 31% and 33% of system call events on virtual and physical machines, respectively. On 32-core machines, Sysdig drops 98.5% and 98.9% of system call events on virtual and physical machines, respectively. Similarly, LTTng drops 71.9% of total system call events on a 32-core virtual machine and about 60% of total events on a 32-core physical

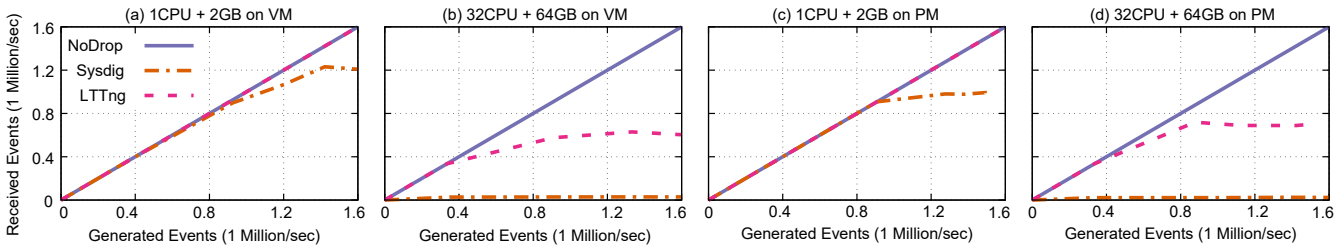

Figure 4: The number of events dropped by NODROP and baselines with different hardware configurations. *The x-axis is the number of generated events in 30 seconds, and the y-axis is the number of events handled by the auditing framework.*

machine. LTTng does not drop events on machines with 1 CPU core and 2 GB memory because it adopts a dynamic buffer mechanism that can hold more events than Sysdig. However, as we will show in the next section, this design also causes LTTng to introduce more system overhead than Sysdig. For Linux Audit, since it is not as well optimized as Sysdig and LTTng [26], it drops nearly all system call events when the super producer reaches 1% of its highest event generation speed.

5.1.1 Preventing PDoS in realistic web-apps:

We evaluated whether NODROP can prevent PDoS attacks by simulating a production-level web-server. We used the 'web-serving' benchmark from CloudSuite as the implementation of the *victim app*. This benchmark hosts a production-quality social networking engine, which we hosted with Apache 2.4, MariaDB 10.1, PHP 7.4, and Elgg 3.3. The *target app* was implemented as a static website that provides a "ping lookup" service to users, but it had a command line injection vulnerability that allowed attackers to run remote code. We hosted the *target app* in a different process of Apache and evaluated PDoS in two widely adopted resource isolation methods: (1) the *target app* and *victim app* were scheduled and isolated by the default Linux configuration, which focuses on maximizing system utilization while providing fairness and performance isolation in a best-effort way; and (2) the *target app* and *victim app* were in different `cgroups` with isolated CPU utilization. All applications were deployed on an Intel Xeon Silver server with 16 CPU cores, 128 GB memory, and a 10 TB HDD.

Our implemented PDoS attack consists of three steps. First, we simulate 20 visitors accessing the *victim app* and turn it into a super producer. Second, the attacker waits for three to five minutes to ensure that the auditing frameworks are overloaded. Third, the attacker initiates a command & control connection to the *target app* by exploiting a command line injection vulnerability. We consider the PDoS attack successful if the auditing frameworks do not record any provenance data about the command & control connection. For each auditing framework, we conduct 120 PDoS attacks and count how many of them are successful.

Results: Our experiment shows that NODROP can prevent PDoS attacks, while all three baseline methods are vulnerable to the PDoS attack. The attack success rates for these baseline methods are higher than 90% in all cases, with at least 107 out of 120 PDoS attacks being successful. These results are shown in Table 2 for both isolation methods. Furthermore, using `cgroup` does not prevent the PDoS attack because the *victim app* does not exceed its quota, and the collector is unable to isolate provenance events internally.

Table 2: Attack success rate of the PDoS attack (#successful/ #attempts)

	Sysdig	LTTng	Linux Audit	NODROP
Default	120/120	107/120	120/120	0/120
Cgroup	115/120	107/120	120/120	0/120

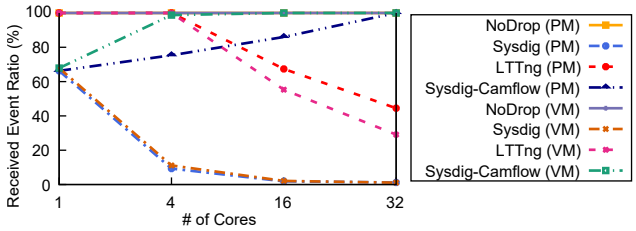

Figure 5: Summary of how the auditing frameworks drop events with different hardware configurations. *The line of* NODROP *overlaps with the 100% events received line, indicating no event has been dropped across all configurations.*

5.2 RQ 2: Application Slowdown

To answer this research question, we implemented realistic PADoS attacks using realistic web applications. In the PADoS attack, we assumed that the *target app* and the *victim app* were running in different `cgroups`. We implemented the *target app* and the *victim app* in different `cgroups` to allow for absolute isolation between them. An auditing framework was running in both `cgroups` to monitor the *target app* and the *victim app*.

We implemented the *victim app* using the same script as in §5.1, and we implemented the *target apps* using three widely used web applications: Nginx [49], Redis [71], and OpenSSL [95]. By leveraging the script, we were able to control the generation speed of provenance data in the super producer (the *victim app*). We evaluated Sysdig, Linux Audit, and LTTng to show how they affected the performance of the *target app*. To adopt a pro-integrity strategy, we implemented two additional versions based on Sysdig. The first one, called Sysdig-Camflow, optimized Camflow using a more efficient

kernel module from Sysdig while preserving the per-core thread user-space collector from Camflow. The second one, called Sysdig-Integrity, integrated synchronized event processing [14] into Sysdig to ensure the integrity of provenance data. Sysdig-Integrity blocked the currently running process when its event buffer was full and woke it up once the buffer had been processed. In this way, Sysdig-Integrity ensured zero event loss and was a guaranteed pro-integrity solution.

We also reported the performance of applications on a vanilla machine as the "No Consumer" to show the baseline performance of the system without any auditing frameworks in the user space. To measure the system performance of our applications, we ran their corresponding benchmark scripts and reported the official performance scores reported by the scripts. This avoided statistical data bias that could have resulted from poorly self-implemented benchmark scripts.

Figure 7 shows detailed results for Nginx with 1 CPU core and 32 CPU cores on virtual machines. The results for other configurations are similar, but we have included them in our GitHub repository due to space constraints. The x-axis of Figure 7 represents the workload of the super producer, namely the speed of generated events per second, while the y-axis represents the official performance score reported by the benchmark scripts. Since the number of consumer threads in Sysdig-Camflow equals the number of CPU cores, Sysdig-Camflow and Sysdig are identical on a single-core machine. Therefore, we merged the lines for Sysdig-Camflow and Sysdig in Figure 7(a). Overall, our experiment showed that an attacker could paralyze the *target app* by turning the *victim* into a super producer in a different `cgroup`.

Our experiments showed that (1) NODROP prevents the PADoS attack, (2) existing pro-integrity solutions suffer from the PADoS attack across different hardware configurations, and (3) pro-performance collectors also slow down the *target app*. For all three web applications on all eight hardware configurations, when the workload of the super producer increased, NODROP maintained stable application performance regardless of the increasing workload from the super producers. For example, in Figure 7, NODROP was at most 5.1% slower on a single-core virtual machine and at most 0.5% slower on a 32 CPU cores virtual machine than the ideal "No Consumer" baseline. This overhead was also lower than that of the baselines. This result proves that NODROP is robust against PADoS attacks. In other words, an attacker cannot slow down applications in different `cgroup` by running a super producer.

We also notice that the pro-performance collectors can also be vulnerable to the PADoS attack. On the single-core virtual and physical machines, Sysdig, LTTng, and Linux Audit also decrease the performance of the *target app* proportionally to the event generation speed of the super producer. This is because Sysdig, LTTng, and Linux Audit limit the resource usage of their user-space component by allowing only one user-space thread. Thus, Sysdig, LTTng, and Linux Audit behave the same as Sysdig-Camflow on a single-core platform, making them de facto pro-integrity collectors. Linux Audit has the worst performance because it relies on Netlink [99] to pass events from the kernel to the user-space component, which leads to a less efficient kernel module than other auditing frameworks.

5.2.1 CloudSuite Setting

To further evaluate how well NODROP can prevent PADoS attacks in a production environment, we also implemented PADoS attacks using the CloudSuite setting that we used in §5.1.

Our implemented PADoS attack consisted of three steps. First, similar to PDoS, the attacker used 20 visitors to generate a flood of remote requests to the victim app. We further adjusted the workload of visitors to generate increasing pressure on the victim app. Second, the attacker waited for five minutes to ensure that the auditing framework had enough resources. Third, the attacker started a normal DoS attack on the target app. In this case, the required workload for the DoS attack on the target app was substantially reduced.

In our experiment, we measured the performance reduction of the target app to evaluate the effectiveness of the PADoS attack. To accurately measure the performance of the target app, we also used *wrk* [94] to issue HTTP requests to the target app and reported the number of returned HTTP responses as its performance metric.

Results: Our evaluation showed that pro-integrity collectors could substantially decrease the performance of the *target app* and, thus, amplify possible DoS attacks. We showed the performance of the *target app* in Figure 6. Note that the request sending rate on the x-axis represents the speed at which requests reach the *victim app*, not the request processing speed of the app. To better understand how pro-integrity collectors decrease the performance of the *target app*, we plotted the case of Sysdig and the case of no provenance collectors (No Consumer) in Figure 6 as baselines.

With the default Linux configuration (Figure 6(a)), we observed that pro-integrity collectors accelerated the DoS attack, requiring less workload to slow down the *target app*. Specifically, the curve for No Consumer remained flat when the system had sufficient spare resources to digest attack traffic. When the workload exceeded 34K packets/s, however, the whole system became overloaded, and the performance of the *target app* rapidly decreased. However, when provenance collectors were present, the *target app* had smaller turning points (for Sysdig, Sysdig-Camflow, and Sysdig-Integrity, these turning points were 32K, 15K, and 15K, respectively) and much lower performance than No Consumer. This was because pro-integrity collectors consumed too many system resources and competed with other applications, overloading the server more easily. In general, Figure 6(a) shows that pro-integrity collectors can amplify DoS attacks since they blindly compete

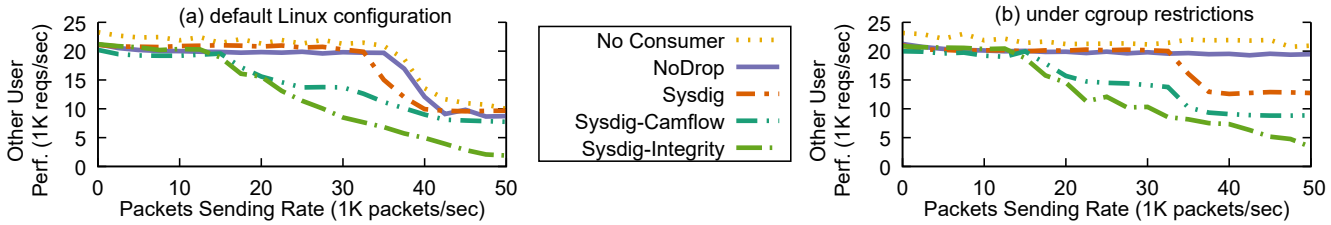

Figure 6: The performance of the *target app*(static page) in the case CloudSuite with a different workload from the super producer.*The x-axis is the packet sending rate of the super producer, and the y-axis is the performance score of wrk, measured in the number of returned HTTP responses. Higher is better.*

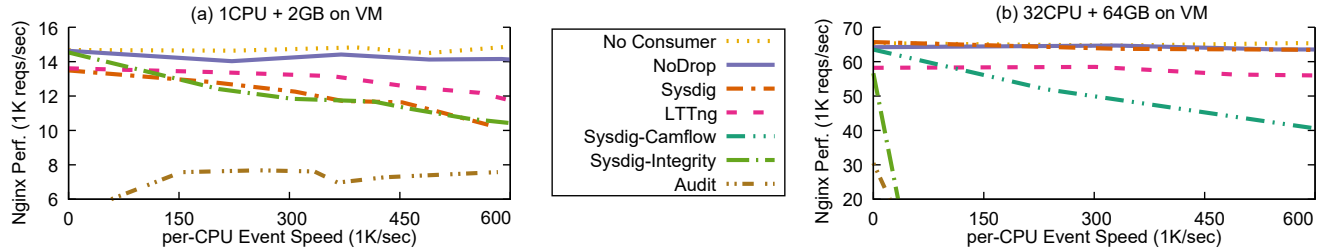

Figure 7: The performance of Nginx on different platforms and hardware configurations under various workloads from the super producer. *The x-axis is the speed of generated events in 30 seconds, and the y-axis is the performance of Nginx, measured in the number of requests per second.*

for resources with other applications.

Figure 6(b) reports the results under `cgroup` restrictions. The *target app* and the *victim app* had 20% and 80% CPU utilization limitations, respectively. Consequently, the super producer could not use up all system resources, and the *target app* was not impacted. The flat line for No Consumer in Figure 6(b) indicates this, showing that the DoS attack was ineffective. However, pro-integrity collectors compromised the isolation of `cgroup` and greatly influenced the *target app*. In the case of Sysdig, Sysdig-Camflow, and Sysdig-Integrity, the worst performance loss for the *target app* reached 40.1%, 55.6%, and 83.7%, respectively. Please note that placing a `cgroup` limitation on provenance collectors would cause a PADoS attack as studied above. In contrast to existing provenance collectors, NODROP preserved `cgroup` isolation and successfully protected the *target app* from DoS attacks.

5.3 RQ 3: Runtime Overhead

This research question evaluated the runtime overhead of NODROP by measuring its impact on the OS and on running applications, both with and without a super producer. We used Sysdig as a baseline, repeated measurements ten times, and employed statistical methods, including the Wilcoxon signed-rank test, to test for significant differences between Sysdig and NODROP. We used p-value [22] as our evaluation metric and concluded that results were statistically significant when $p < 0.05$.

5.3.1 OS Overhead

Similar to evaluations in other auditing frameworks [86], we used *lmbench* [76] to measure the OS overhead of NODROP. On average, the OS overhead of NODROP was 35% higher than vanilla Linux across eight different hardware configurations. We also showed the relative overhead percentage compared to Sysdig for four configurations in Table 3 and left the results for other configurations in our GitHub repository.

Our experiments show that the OS overhead of NODROP is similar to that of Sysdig. On average, the OS overhead of NODROP is 0.2% lower than Sysdig across eight different hardware configurations. Overall, we cannot find a statistically significant difference between Sysdig and NODROP. We conclude that the differences between the OS overhead of NODROP and Sysdig are mainly due to the randomness of a dynamic system. The OS overhead measures the performance of the kernel modules of NODROP and Sysdig. Since NODROP implements its kernel module in the same way as Sysdig, their OS overhead should be the same.

5.3.2 Application Overhead

To answer this research question, we conducted macro-benchmarks to measure the performance of seven applications, both with and without a super producer. These applications can be divided into two categories: the first category includes I/O-intensive benchmarks such as Nginx [49], Redis [71], and Postmark [53], as well as two other applications, Django [63] for Python and http [30] for Golang. The second category includes CPU-intensive benchmarks, namely OpenSSL [95] and 7-ZIP [84].

Table 3: Performance scores of *lmbench*. *All values are shown as percentages relative to Sysdig. The negative value means* NODROP *is faster than Sysdig.*

Configurations	C1	C4	C5	C8	Ave
Syscall Tests					
NULL syscall	-8.1%	-17%	-8.3%	-7.9%	-10.3%
stat	-9.0%	+5.5%	-1.8%	-0.6%	-1.5%
fstat	+4.2%	-1.7%	+1.7%	+1.6%	+2.3%
open/close file	-6.1%	-2.9%	-0.3%	-1.8%	-2.8%
read file	+7.4%	+7.1%	+4.5%	+7.2%	+6.6%
write file	+7.7%	+7.2%	+12.5%	+12.1%	+9.9%
File Access					
file create (0K)	-15.8%	-7.1%	-10.0%	+2.7%	-7.5%
file delete (0K)	+0.5%	+3.0%	-0.7%	-0.9%	+0.5%
file create (10K)	+0.1%	+2.9%	-3.7%	-0.8%	-0.4%
file delete (10K)	+4.7%	+1.5%	-0.9%	-0.4%	+1.2%
pipe	+3.0%	+0.8%	+6.9%	+1.3%	+3.0%
AF_UNIX	+3.8%	-10.5%	+5.3%	+10.1%	+2.2%

We use the official benchmark tools with their default settings for different hardware configurations to evaluate their performance. We repeat each experiment 10 times and measure the average metrics reported by the benchmarks of each application. Specifically, We use *wrk* [94] configured with 1,000 concurrent connections to benchmark Nginx. For Redis, we use the *redis-benchmark* configured to send 1,000,000 requests and measure the speed of operation `get`. For Postmark, we use the built-in benchmark with the configuration of manipulating 500 files concurrently and launching 100,000 transactions. We rely on the Phoronix Test Suite, one of the most comprehensive benchmark suites of web applications [64, 101], to benchmark Django and http. For OpenSSL, We use the built-in *speed* benchmark configured to utilize all CPU cores and measure the time to compute one rsa4096 signature. For 7-ZIP, we use the built-in benchmark configured to utilize all CPU cores and measure the compression speed in MIPS. We report the relative cost of NODROP to Sysdig for four configurations in Table 4a (without the super producer) and Table 4b (with the super producer). We also measure the overhead relative to the vanilla Linux with no auditing framework running at all. We leave the results of other configurations in our GitHub repository.

On average, the overhead of NODROP is 6.58% higher than vanilla Linux and 6.30% lower than Sysdig across eight different configurations. Our statistical analysis confirms the validity of our data. Both Sysdig and NODROP introduce overhead compared to vanilla Linux since they record and consume provenance events. However, the overheads of NODROP to vanilla Linux are relatively less, with overheads less than 15% for all applications except Postmark. The reasons for the difference in application overheads between NODROP and Sysdig depend on hardware configurations and application categories.

For single-core machines (C1 and C5 in Table 4a and Table 4b), NODROP is more efficient because it eliminates the process scheduling overhead. With a single-core, the OS needs to periodically switch to the Sysdig process for provenance data processing, which leads to higher application overhead.

For multi-core machines, NODROP shows relatively high performance because it not only eliminates the process switching cost of Sysdig but also avoids cross-core data transmission. With multiple cores, the kernel module collects the provenance data on the same core as the running application. However, the Sysdig process accesses the data in parallel on a different core. This introduces a notable overhead of cache coherence across the two cores due to the shared provenance data buffer. On the contrary, NODROP processes the provenance data on the same core as the in-kernel collector, avoiding the cost of cross-core data transmission.

As shown in Tables 4a and 4b, NODROP offers lower runtime overhead than Sysdig for applications with many I/O-intensive processes, such as Nginx, Redis, and http. If there are more I/O-intensive processes than CPU cores, the Sysdig auditing processes will compete for computational resources with the monitored applications. In other words, the auditing processes of Sysdig will interrupt the monitored apps in the same manner as NODROP. However, NODROP employs threadlets by design, which add less scheduling overhead than Sysdig. The fewer the number of CPU cores, the greater the scheduling overhead for Sysdig. Thus, NODROP is typically more effective when there are fewer CPU cores. Moreover, we find that as the number of CPU cores increases, the application's event generation speed decreases due to changes in application architecture. As a result, NODROP offers lower runtime overhead in C4 compared to C1.

For CPU-intensive applications such as 7-ZIP, OpenSSL, and Django, they generate almost no system call events. Therefore, when there is no super producer, the overheads of both NODROP and Sysdig compared to vanilla Linux are much smaller, averaging less than 2% across all configurations. When a super producer is running, the overhead of NODROP remains while the overhead of Sysdig increases. This is because the centralized Sysdig auditing process needs to process a large number of events generated by the super producer and will persistently compete for computational resources with the monitored application.

Sysdig may introduce less runtime overhead to the monitored process when there are spare CPU cores available to host the centralized auditing processes. In this case, there is no resource competition between the monitored and auditing processes. This is also the case for Postmark in C4 of Tables 4a and 4b since our Postmark benchmark is single-threaded. Although NODROP shows higher runtime overhead for Postmark compared to Sysdig, it spares the core that would otherwise be used to host the Sysdig auditing process and prevents event dropping.

Kubernetes-managed application. To monitor IO-bound

(a) **Benchmark WITHOUT super producer.**

Application	Collector	C1	C4	C5	C8
Nginx	NODROP	9.80	3.60	11.35	4.84
	Sysdig	55.30	5.50	37.64	7.20
	DIFF	**-29.30**	**-1.80**	**-19.10**	**-2.20**
Redis	NODROP	8.90	3.10	8.66	2.42
	Sysdig	21.00	5.00	21.00	5.70
	DIFF	**-11.10**	**-2.00**	**-10.20**	**-3.10**
Postmark	NODROP	25.50	19.10	30.60	20.65
	Sysdig	96.30	8.60	95.80	13.50
	DIFF	**-36.00**	**9.70**	**-33.30**	**6.30**
Django (Python)	NODROP	1.30	2.00	1.30	-0.20
	Sysdig	1.10	2.30	1.10	0.30
	DIFF	0.30	-0.30	0.20	-0.50
http (Golang)	NODROP	14.10	2.20	14.66	2.71
	Sysdig	78.90	2.20	65.70	2.20
	DIFF	**-36.20**	0.10	**-30.80**	0.50
OpenSSL	NODROP	0.60	0.10	0.20	0.70
	Sysdig	0.60	0.10	0.20	0.60
	DIFF	0.10	0.00	0.00	0.10
7-ZIP	NODROP	0.30	0.80	1.20	0.70
	Sysdig	0.20	0.70	1.20	0.70
	DIFF	0.10	0.10	0.00	0.00
PostgreSQL	NODROP	7.05	3.80	10.20	4.70
	Sysdig	15.20	4.70	17.40	4.90
	DIFF	**-7.61**	-0.80	**-6.50**	-0.19

(b) **Benchmark WITH super producer.**

C1	C4	C5	C8
10.00	3.20	13.15	1.96
51.93	6.28	58.70	3.20
-27.60	**-2.90**	**-28.70**	**-1.20**
3.70	1.20	4.23	0.27
56.17	7.66	61.10	3.80
-33.60	**-6.00**	**-35.30**	**-3.40**
15.10	33.50	14.30	30.20
65.37	6.37	68.58	7.43
-30.40	**25.50**	**-32.20**	**21.20**
1.20	3.30	1.50	1.40
41.14	2.18	47.74	0.90
-28.30	1.10	**-31.30**	0.50
5.80	4.80	9.80	3.20
47.56	0.48	47.38	0.10
-28.30	**4.30**	**-25.50**	**3.10**
0.80	3.20	0.17	0.08
47.37	5.09	43.30	1.40
-31.60	**-1.80**	**-30.10**	**-1.30**
0.40	1.50	0.20	1.20
50.30	4.86	38.02	3.16
-33.20	**-3.20**	**-27.40**	**-1.90**
11.60	4.50	12.20	6.20
22.30	4.86	25.02	6.30
-9.50	-0.34	**-11.40**	0.00

Table 4: **We measured the processing time per request/transaction for seven representative applications and a Kubernetes-based PostgreSQL. For each application, the first two lines show the relative runtime overhead (%) compared to vanilla Linux, where a lower value indicates performance closer to that of vanilla Linux. The third line shows the relative overhead between NODROP and Sysdig, with values smaller than 0 indicating that NODROP outperforms Sysdig. For brevity, we denote this as DIFF. We report the mean values across 10 runs, with p-values less than 0.05 shown in bold.**

applications and complex systems, we combine PostgreSQL Operator with Pgpool-II to deploy a PostgreSQL cluster with query load balancing and connection pooling capability on Kubernetes [12]. We deploy a Pgpool-II pod that contains a Pgpool-II container and a Pgpool-II Exporter container. The Pgpool-II container Docker image is built with streaming replication mode. We set the replicas to 1, so we have three pods in total. To test the replication functionality, we use a benchmark tool called pgbench [11], which comes with the standard PostgreSQL installation, and we measure the processing time per transaction. We repeat each test 10 times for all configurations, and each test lasts for 20 seconds.

Since we implemented the privilege escalation as mentioned in §4.4.2, NODROP is able to monitor system behaviors inside the Docker container. The results are shown in Tables 4a and 4b. We find that for each configuration, NODROP introduces less than 15% overhead compared to vanilla Linux. On average, NODROP introduces 3.34% lower overhead than Sysdig. These results show that NODROP is suitable for monitoring Kubernetes-based deployments.

5.4 RQ 4: Effectiveness of Data Reduction

Several log reduction and partitioning techniques, such as CPR [110], LogGC [66], ProTracer [75], and KCAL [73], do not solve the data integrity vs. performance dilemma because they add high computation overhead, amplifying PADoS attacks. To validate their ineffectiveness, we modified Sysdig's code by inlining the CPR algorithm into the kernel and called it Sysdig-CPR.

Design of Sysdig-CPR: Since the CPR algorithm is an offline algorithm that depends on the global properties of graphs, we maintain a temporary graph in an extra 8M kernel buffer for each CPU core. The design of Sysdig-CPR is similar to Sysdig-Integrity, which blocks the currently running process when the event buffer (kernel and userspace shared buffer) is full and wakes it up once the buffer has been processed. Sysdig-CPR does two additional things: it wakes up a kernel thread to run the CPR algorithm when

Table 5: Dropping rate of auditing frameworks with maximum buffer size.

Configuration	C1	C4	C5	C8
Linux Audit	99.2%	99.6%	99.1%	99.4%
Sysdig	19.3%	86.1%	25.5%	88.1%
LTTng	0%	49.5%	0%	52.1%

the kernel buffer is full and copies the reduced events to the shared buffer for consumption; and it notifies the userspace component to consume the buffer. Sysdig-CPR is available at: https://github.com/nodropforsecurity/sysdigcpr

Results: Our measurement follows the configurations of §5.2. We omit Sysdig-CPR for clarity in Figure 7 as well as our GitHub repository because we find that the performance curves of Sysdig-CPR in all configurations are stuck on the horizontal axis. In our experiment, the kernel CPR can handle 2,000 events per second per core, which is consistent with the original paper [110]. Although it can reduce most of the events (more than 70%), the super producer can easily generate 100,000 events per second. This means that the system will take several seconds to handle the generated events, which greatly blocks the running applications.

5.5 RQ 5: Effectiveness of Increasing Buffer Size

One possible approach to avoid event dropping in system auditing frameworks [3, 4, 104] is to increase the buffer size. However, this approach is ineffective. We applied this approach to Sysdig, LTTng, and Linux Audit and evaluated them as follows. For each CPU core, we set the maximum applicable buffer size, which is 768M for Sysdig and LTTng and 77,000 messages for Linux Audit. We cannot increase the buffer size even larger because the system will crash when the buffer size exceeds the threshold. In our measurement shown in Table 5, Sysdig, LTTng, and Linux Audit still drop events with the same super producer configuration in §5.1. The dropping rate is 88%, 52%, and 99% for Sysdig, LTTng, and Linux Audit when the generation speed per core reaches 1.6 million per second. Moreover, as reported by the developers of Sysdig, increasing the size of the ring buffer may significantly slow down the whole system [5, 104]. According to our experiment, when increasing buffer size from 8M to 768M, the events generation speed decreases by 35% under the same stress test.

6 Related Work

Provenance analysis has been widely applied in different security tasks, such as APT attack investigation [18, 32–34, 50, 51, 56, 70, 75, 90, 91, 93] and detecting stealthy security risks [19, 27, 37, 40–43, 46, 69, 77, 78, 83, 92, 102, 103, 108]. There are also methods for precisely and clearly interpreting events to explain applications' behaviors [17, 44, 62, 65, 72, 74, 80, 111]. NODROP benefits these tasks by providing a more reliable data source. Attackers regularly engage in anti-forensic activities to cover their tracks [2]. Several cryptographic-based approaches are proposed to secure logs [8, 28, 52, 85, 86], but none discuss the security of the user-space component of auditing frameworks.

Threadlet is a short sequence of instructions with self-contained memory [58, 59, 67, 87, 88]. NODROP borrows this concept but implements it differently as a piece of code instrumented to a host thread. NODROP provides protections such as MPK, address randomization, and heap isolation.

7 Discussion

NODROP may allow a malicious process to compromise the consumer residents in its memory. To this end, we adopt a comprehensive solution that combines address randomization, dedicated heap, MPK protection, and ensuring the atomic execution of the consumer, as discussed in §4.4.2. Thus, although the consumer shares the same memory space as user applications, they are still protected. We notice that MPK is available in most of the latest Intel server and client-side CPUs. ARM, AMD, RISC-V, PowerPC, and Itanium CPUs [13, 23, 47, 68, 107] also have similar mechanisms.

Although NODROP prevents attackers from slowing down other applications, the attacker can still slow down a process by injecting the super producer logic into the process directly. We consider the thread model of this attack too strong for NODROP. Indeed, as long as the attacker can compromise a process, it is straightforward to slow down the process. How to protect a running process from hijacking is beyond the scope of this paper.

Windows provides the ETW [100] framework for provenance collection, but it only has a kernel module and leaves the user-space logic for customization. Thus, we cannot find an "official" user-space component for ETW. Nevertheless, the super producer vulnerability is about process scheduling and isolation, which is general to both Linux and Windows.

8 Conclusion

This paper is the first to identify the super producer threat to existing auditing frameworks. Through thorough experiments and case studies, we find attackers can either disable existing auditing frameworks or paralyze the whole system with a super producer. Based on our discovery, we propose a novel auditing framework, NODROP, that addresses the super producer threat by providing resource isolation. Our evaluation shows that NODROP prevents the super producer threat while introducing 6.30% lower application overhead on average across eight different hardware configurations than Sysdig.

9 Acknowledgement

We sincerely thank our Shepherd and all the anonymous reviewers for their valuable comments. This work was partly supported by the National Key Research and Development Program (No. 2022YFB4501802), the National Natural Science Foundation of China (No. 62172009, No. 62141208) and Huawei Research Fund.

References

[1] camflow > developer information. https://camflow.org/#query.

[2] Capec-268: Audit log manipulation. https://capec.mitre.org/data/definitions/268.html.

[3] Document falco syscall buffer size adjustment described in blog. https://github.com/falcosecurity/falco/issues/813.

[4] Falco on gke - dropped syscall events. https://github.com/falcosecurity/falco/issues/669.

[5] Falco on gke - dropped syscall events. https://github.com/falcosecurity/falco/issues/669#issuecomment-635476220.

[6] falco security > developer information. https://stackshare.io/falco-security.

[7] Multitenant web applications. https://learn.microsoft.com/en-us/azure/dotnet-develop-multitenant-applications.

[8] Secure logging with syslog-ng. https://archive.fosdem.org/2020/schedule/event/security_secure_logging_with_syslog_ng/.

[9] Vmware carbon black. https://www.vmware.com/products/whats-new/carbon-black.html.

[10] What is a security operations center (soc)? https://www.splunk.com/en_us/data-insider/what-is-a-security-operations-center.html.

[11] pgbench documentation. https://www.postgresql.org/docs/current/pgbench.html, 2020.

[12] pgpool documentation. https://www.pgpool.net/docs/pgpool-II-4.2.3/en/html/example-kubernetes.html, 2020.

[13] Inc. Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Advanced Micro Devices, Inc., 2021.

[14] Adil Ahmad, Sangho Lee, and Marcus Peinado. Hardlog: Practical tamper-proof system auditing using a novel audit device. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1554–1554. IEEE Computer Society, 2022.

[15] Ali Anafcheh. Intrusion detection with ossec. 2018.

[16] Adam Bates, Kevin Butler, Alin Dobra, Brad Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Retrofitting applications with provenance-based security monitoring. *arXiv preprint arXiv:1609.00266*, 2016.

[17] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 887–895, 2017.

[18] Adam Bates, Dave (Jing) Tian, Kevin R.B. Butler, and Thomas Moyer. Trustworthy Whole-System provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 319–334, Washington, D.C., August 2015. USENIX Association.

[19] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 35–44, 2015.

[20] Gianluca Borello. System and application monitoring and troubleshooting with sysdig. 2015.

[21] Kevin D Bowers, Catherine Hart, Ari Juels, and Nikos Triandopoulos. Pillarbox: Combating next-generation malware with fast forward-secure logging. In *International Workshop on Recent Advances in Intrusion Detection*, pages 46–67. Springer, 2014.

[22] William Jay Conover. *Practical nonparametric statistics*. Wiley series in probability and statistics. Wiley, New York, NY [u.a.], 3. ed edition, 1999.

[23] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2021.

[24] dark laboratorys. A better generation of logcleaners. https://web.archive.org/web/20070218231819/http://darklab.org/~jot/logcleaning/logcleaner-ng_1.0_lib.html.

[25] Jessica DeCianno. Indicators of attack vs. indicators of compromise. *CrowdStrike*, 2014.

[26] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Citeseer, 2006.

[27] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298, 2017.

[28] Jake Edge. Forward secure sealing. https://lwn.net/Articles/512895/.

[29] Pengcheng Fang, Peng Gao, Changlin Liu, Erman Ayday, Kangkook Jee, Ting Wang, Yanfang Fanny Ye, Zhuotao Liu, and Xusheng Xiao. Back-propagating system dependency impact for attack investigation. In *USENIX security 2021*.

[30] Athenas Jimenez Gabriela Cervantes. Go benchmarks. https://openbenchmarking.org/test/pts/go-benchmark, 2022.

[31] Holger Gantikow, Christoph Reich, Martin Knahl, and Nathan L Clarke. Rule-based security monitoring of containerized workloads. In *CLOSER*, 2019.

[32] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. {SAQL}: A stream-based query system for {Real-Time} abnormal system behavior detection. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[33] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. {AIQL}: Enabling efficient attack investigation from system monitoring data. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 113–126, 2018.

[34] Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. In Priya Narasimhan and Peter Triantafillou, editors, *Middleware 2012*, pages 101–120, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[35] Andrés Santiago Gómez Vidal. Improvements in ids: adding functionality to wazuh. 2019.

[36] Martin Grimmer, Martin Max Röhling, D Kreusel, and Simon Ganz. A modern and sophisticated host based intrusion detection data set. *IT-Sicherheit als Voraussetzung für eine erfolgreiche Digitalisierung*, 2019.

[37] Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, and Dongyan Xu. Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 57–68. IEEE, 2015.

[38] Jiaping Gui, Xusheng Xiao, Ding Li, Chung Hwan Kim, and Haifeng Chen. Progressive processing of system-behavioral query. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 378–389, 2019.

[39] Steve Hales. Last door log wiper. https://packetstormsecurity.com/files/118922/LastDoor.tar.

[40] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *Network and Distributed System Security Symposium (NDSS'20)*. Internet Society, 2020.

[41] Xueyuan Han, Xiao Yu, Thomas Pasquier, Ding Li, Junghwan Rhee, James Mickens, Margo Seltzer, and Haifeng Chen. SIGL: Securing software installations through deep graph learning. 2020.

[42] W. U. Hassan, A. Bates, and D. Marino. Tactical provenance analysis for endpoint detection and response systems. In *2020 IEEE symposium on security and privacy (SP)*, pages 1172–1189, 2020.

[43] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Network and Distributed Systems Security Symposium*, 2019.

[44] Wajih Ul Hassan, Mohammad Ali Noureddine, Pubali Datta, and Adam Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and Distributed System Security Symposium*, 2020.

[45] Viet Tung Hoang, Cong Wu, and Xin Yuan. Faster yet safer: Logging system via Fixed-Key blockcipher. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2389–2406, Boston, MA, August 2022. USENIX Association.

[46] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data. In *26th USENIX Security Symposium*, 2017.

[47] IBM Corporation. Power isa version 3.0b. IBM website, 2021.

[48] Muhammad Adil Inam, Wajih Ul Hassan, Ali Ahad, Adam Bates, Rashid Tahir, Tianyin Xu, and Fareed Zaffar. Forensic analysis of configuration-based attacks.

[49] NGINX Inc. Nginx 1.21.1. https://www.nginx.com/, 2022.

[50] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[51] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable {Cross-Host} attack investigation with efficient data flow tagging and tracking. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[52] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 19–30, New York, NY, USA, 2017. Association for Computing Machinery.

[53] Jeffrey Katcher. Postmark: A new file system benchmark. *TR3022*, 1997.

[54] The kernel development community. Memory protection keys. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html, 2022.

[55] Wael Khreich, Babak Khosravifar, Abdelwahab Hamou-Lhadj, and Chamseddine Talhi. An anomaly detection system based on variable n-gram features and one-class svm. *Information and Software Technology*, 91:186–197, 2017.

[56] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 223–236, New York, NY, USA, 2003. ACM. event-place: Bolton Landing, NY, USA.

[57] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[58] Peter M Kogge. Of piglets and threadlets: Architectures for self-contained, mobile, memory programming. In *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'04)*, pages 130–138. IEEE, 2004.

[59] Peter M. Kogge. Multi-threading semantics for highly heterogeneous systems using mobile threads. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 281–289, 2019.

[60] Iman Kohyarnejadfard, Daniel Aloise, Michel Dagenais, and Vahid Azhari. Anomaly detection in microservice systems using tracing data and machine learning. 2021.

[61] Iman Kohyarnejadfard, Mahsa Shakeri, and Daniel Aloise. System performance anomaly detection using tracing data analysis. In *Proceedings of the 2019 5th International Conference on Computer and Technology Applications*, pages 169–173, 2019.

[62] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, volume 2, page 4, 2018.

[63] Michael Larabel. Pyperformance benchmark. https://openbenchmarking.org/test/pts/pyperformance, 2022.

[64] Michael Larabel and Matthew Tippett. Phoronix test suite. *Phoronix Media, [Online]. Available: http://www.phoronix-test-suite.com/. [Accessed October 2022]*, 2011.

[65] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, volume 2, page 4, 2013.

[66] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: Garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, New York, NY, USA, 2013. Association for Computing Machinery.

[67] Sheng Li, Amit Kashyap, Shannon Kuntz, Jay Brockman, Peter Kogge, Paul Springer, and Gary Block. A heterogeneous lightweight multithreaded architecture. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[68] ARM Limited. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. ARM Limited, 2011.

[69] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1777–1794, 2019.

[70] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[71] Redis Ltd. Redis 6.0.9. https://redis.io/, 2022.

[72] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 401–410, 2015.

[73] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 241–253, USA, 2018. USENIX Association.

[74] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1111–1128, 2017.

[75] ShiQing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[76] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[77] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1795–1812, 2019.

[78] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, Ramachandran Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[79] Bruno Morisson. Analysis of the linux audit system. *Master's thesis, Information Security Group, Royal Holloway, University of London*, 2014.

[80] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, page 4, USA, 2006. USENIX Association.

[81] OccupytheWeb. How to cover your tracks & leave no trace behind on the target system. https://null-byte.wonderhowto.com/how-to/hack-like-pro-cover-your-tracks-leave-no-trace-behind-target-system-0148123, 2013.

[82] openEuler. https://www.openeuler.org/.

[83] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015.

[84] p7zip. p7zip version 16.02. https://www.7-zip.org/, 2022.

[85] Riccardo Paccagnella, Pubali Datta, Wajih Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. 01 2020.

[86] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[87] Brian A. Page and Peter M. Kogge. Deluge: Achieving superior efficiency, throughput, and scalability with actor based streaming on migrating threads. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2021.

[88] Brian A. Page and Peter M. Kogge. Passel: Improved scalability and efficiency of distributed svm using a cacheless pgas migrating thread architecture. In *2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 27–34, 2021.

[89] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[90] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 405–418, New York, NY, USA, 2017. Association for Computing Machinery.

[91] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime analysis of whole-system provenance. In *Conference on Computer and Communications Security (CCS'18)*. ACM, 2018.

[92] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, pages 583–595, 2016.

[93] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 259–268, 2012.

[94] Open Source Project. Wrk. https://github.com/wg/wrk, 2022.

[95] The OpenSSL Project. Openssl 1.1.1. https://www.openssl.org/, 2022.

[96] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-Grained isolation for scalable, dynamic, multi-tenant edge clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[97] Navin Sabharwal and Piyush Pandey. Monitoring gke using sysdig. In *Pro Google Kubernetes Engine*, pages 257–298. Springer, 2020.

[98] Jorge Salamero. Kubernetes runtime security with falco and sysdig, 2019.

[99] J Salim, H Khosravi, Andi Kleen, and Alexey Kuznetsov. Linux netlink as an ip services protocol. Technical report, 2003.

[100] Thomas Schlabach. *Insight into Event Tracing for Windows*. Bachelor thesis, Offenburg.

[101] Basu A Sharath S. Performance of eucalyptus and openstack clouds on futuregrid. In *International Journal of Computer Applications*, 2013.

[102] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 592–605, 2018.

[103] Yun Shen and Gianluca Stringhini. {ATTACK2VEC}: Leveraging temporal word embeddings to understand the evolution of cyberattacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 905–921, 2019.

[104] Mark Stemm. Cve-2019-8339, a falco capacity related vulnerability. https://sysdig.com/blog/cve-2019-8339-falco-vulnerability/?msclkid=4c2c25afa9b511ec815c58c5f800beec, 2019.05.19.

[105] Afroza Sultana, Abdelwahab Hamou-Lhadj, and Mario Couture. An improved hidden markov model for anomaly detection using frequent common patterns. In *2012 IEEE International Conference on Communications (ICC)*, pages 1113–1117. IEEE, 2012.

[106] Chin-Wei Tien, Tse-Yung Huang, Chia-Wei Tien, Ting-Chun Huang, and Sy-Yen Kuo. Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches. *Engineering reports*, 1(5):e12080, 2019.

[107] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. *erim*: Secure, efficient in-process isolation with protection keys (*mpk*). In *28th USENIX Security Symposium*, pages 1221–1238, 2019.

[108] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, and others. You are what you do: Hunting stealthy malware via data provenance analysis. In *Symposium on network and distributed system security (NDSS)*, 2020.

[109] Shen Wang, Zhengzhang Chen, Ding Li, Lu-An Tang, Jingchao Ni, Zhichun Li, Junghwan Rhee, Haifeng Chen, and Philip S. Yu. Attentional heterogeneous graph neural network: Application to program reidentification. 2019.

[110] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 504–516, New York, NY, USA, 2016. Association for Computing Machinery.

[111] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. In *NDSS*, 2020.