Fine-Grained Kernel Auditing using Augmented Syscall Reference Behavior Analysis and Virtualized Selective Tracing

Chuqi Zhang[§] Spencer Faith[†] Feras Al-Qassas[†] Theodorus Februanto[†] Zhenkai Liang[§] Adil Ahmad[†]
[§]*National University of Singapore*
[†]*Arizona State University*
{chuqiz,liangzk}@comp.nus.edu.sg, {smfaith,fialqass,tfebruan,adil.ahmad}@asu.edu

***Abstract*—Audit logs are widely used for attack investigation in enterprises, but their granularity (system calls and related events) is too coarse-grained to be useful for attack forensics when adversaries launch advanced kernel exploits. Such exploits manipulate kernel memory to hijack kernel control-flow, and these aspects (i.e., the executed anomaly control flows and their capabilities) are not visible in today's audit logs.**

APPARE is an auditing framework designed to comprehensively and efficiently capture sophisticated in-memory kernel exploit behaviors. APPARE implements anomalous control-flow logging, where it leverages an augmented hybrid approach to (a) dynamically profile representative system call workloads, and (b) generalize the profiles by using LLM-assisted code semantics reasoning to differentiate reference (benign) and anomalous function executions within the kernel. APPARE uses efficient hardware tracing techniques to record anomaly control flow behaviors, as well as the historical contexts to reveal where control flow divergences (hijacking) happen. APPARE leverages virtualization extensions and features available in modern architectures to achieve end-to-end tamper-proof logging, persistence, and management. Our analysis and evaluation show that APPARE effectively captures attack behaviors in the exploits we analyzed, while incurring a geometric mean slowdown of only 2.0% across diverse programs.

1. Introduction

The widespread adoption of memory-unsafe languages in commodity operating systems (OSs) makes them vulnerable to sophisticated *in-memory* exploits that hijack control-flow during execution for compromise. Despite the evolution of advanced defenses such as memory layout randomization (e.g., KASLR, RANDSTACK) and kernel control-flow integrity (e.g., KCFI) [1–5], attackers continue to devise methods to bypass these protections. For instance, RetSpill [6] and DirtyCred [7] can circumvent all defenses currently deployed in Linux, including KCFI [8].

Given the persistent threat posed by control-flow exploits, there is a critical need in mission-critical (e.g., enterprise) infrastructure for systems that can help investigate compromises resulting from control-flow hijacks. Such investigations aim to either hunt for compromises through intrusion detection or conduct comprehensive post-mortem forensic analysis. Traditionally, helping in attack investigations is the role of the audit system, a kernel component that records sensitive interactions by user programs (e.g., process and file creation) in the form of *audit logs*. The seminal BackTracker system [9] showed that these logs can trace attack origin and vectors if the system is compromised.

Unfortunately, current audit logs are too coarse-grained for the investigation of sophisticated kernel exploits (§2). Specifically, modern audit systems *only* record information at the granularity of system call entry-points [10, 11] and user-space context between system calls [12–15], completely *missing malicious behaviors that occur entirely in-memory and inside system call executions*. For instance, consider the control-flow hijacking exploit of CVE-2021-22555 [16]. Current audit systems would log routine I/O-related system calls (e.g., `write`, `close`), but overlook the critical anomalous control-flow executed within the syscalls—execution of a sequence of task credential manipulation functions that elevates the attacker's privilege to root [16].

Enhancing the granularity of audit logs to effectively investigate these sophisticated exploits requires tracking fine-grained control-flow during system calls, but this is not straightforward. The kernel has tens of thousands of functions. Indiscriminately tracing all functions produces significant runtime overhead and an overwhelming volume of logs, an aspect that also complicates the identification and investigation of malicious activities within logs [17].

Our work—APPARE—solves the aforementioned problems by only logging *anomalous* and *unlikely* control-flow events during system calls (§4). The former refers to executions that are unreachable during a system call as inferred through control-flow graphs generated by static analysis techniques. The latter refers to executions that are reachable but may never happen due to imprecision in static analysis or may happen *very infrequently* in less-explored and niche code-paths. Combining both allows APPARE to define a *reference behavior* of each system call that is general enough to avoid logging common (benign) control-flow executions, while also being able to capture diverse exploit conditions. Our evaluation shows that this reference behavior only contains $0.01\% - 7.11\%$ of all kernel functions (75.1% smaller on average than naive approaches), allowing the system to log all kernel functions in evaluated exploits, while incurring a performance overhead of only $0.2\% - 6.8\%$.

We address two main technical challenges in APPARE's design. First, it is challenging to decipher the unlikely control-flow required to define reference syscall behavior. Prior research [18–20] leverages workload profiling alone, but this leads to severe under-approximation of runtime behavior, increasing the performance and storage overheads, and scalability problems due to strict workload-dependence. Second, it is challenging to selectively trace anomalous control flow in a secure and efficient manner using current approaches like `ftrace` or Processor Tracing (PT) [14]. Neither approach allows a fine-grained (function-level) toggle required for the distinct reference behavior of each system call. Even worse, generated traces are vulnerable to OS tampering during exploits, harming log integrity and availability.

To effectively identify system call reference behaviors, APPARE augments representative workload profiling with static kernel knowledge base extraction and retrieval augmented generation (RAG)-based code semantics reasoning (§5.1). APPARE pre-profiles a small set of representative workloads (e.g., OS developer-maintained test suites) to derive seed behaviors for syscalls. Simultaneously, APPARE extracts static kernel control-flow (call graph) and function semantics knowledge. APPARE generates generic syscall reference behaviors by expanding from the profiled seed functions with augmented hybrid analysis. Starting from seed functions, APPARE traverses the kernel call graph and employs a large language model (LLM), which retrieves historical execution contexts as well as the corresponding semantics information, and reasons about uncovered but plausible newly explored control-flow transitions.

To securely and selectively trace anomalous control-flow, APPARE leverages virtualization extensions to maintain restricted code pages, transparent memory-view switching, and guarded stacks (§5.2). For each syscall, APPARE creates pages where all non-reference kernel functions are replaced with illegal instructions. Executing them (i.e., anomalous control-flow) raises a trap caught by APPARE, at which time APPARE enables tracing while transparently switching to unrestricted (normal) code pages. APPARE uses a guarded stack that does not contain the return address while tracing anomalous control-flow. When the stack is exhausted (i.e., control-flow returns to reference functions), another trap is raised for APPARE to disable tracing. The aforementioned mechanisms are combined with hardware-protected tracing buffers and configurations, as well as an asynchronous persistence mechanism to ensure control-flow traces are always available for investigation even after kernel compromise.

We implemented an APPARE prototype for Intel x86 systems that run the Linux OS kernel (§6). Our prototype consists of (1) a *profiler* toolchain that automatically derives syscall reference behaviors, (2) a *logger* (as a virtualization-based monitor) to perform end-to-end online secure logging and persistence, and (3) a *parser* that leverages a static binary loader and trace decoder to parse hardware trace packets into specific kernel functions and chronologically link syscall with control-flow logs. To foster future research, we open-source our implementation prototype. Our source code is at: https://github.com/ASTERISC-Release/Appare.

TABLE 1: Comparison between APPARE (this work) and existing audit systems (deployed or proposed by research).

Proposed Systems	Conventional OS-deployed [11, 21]	Dependency enhanced [12–15]	Tamper-evident [22–24]	Tamper-proof [25–27]	APPARE (this work)
Information sources reported within logs					
System call	Yes	Yes	Yes	Yes	Yes
Control-flow	No	Between-syscalls (userspace)	No	No	**In-syscalls (kernspace)**
Tamper protection against untrusted kernel					
Integrity	No	No	**Syscall**	**Syscall**	**All Sources**
Availability	No	No	No	**Syscall**	**All Sources**

Using our prototype, we evaluated APPARE on several dimensions using exploits, benchmarks, and real-world programs (§7–§8). In terms of security, APPARE demonstrates a tight reference behavior bound (as discussed previously) that allows it to log kernel functions exploited in 10 out of 11 real-world CVEs, including RetSpill [6] that can exploit Linux with KCFI enabled. To gauge performance and storage requirements, we evaluated diverse real-world programs—Nginx, Redis, Memcached, 7zip, and OpenSSL. APPARE's small performance overhead ($0.2\% - 6.8\%$) is $13\% - 77\%$ lower than approaches that define reference behavior using dynamic workload profiling alone, while its storage overhead is (on average) 51% lower, demonstrating its practicality.

2. Motivation

This section describes what information is required for attack investigation (i.e., runtime threat hunting or post-mortem forensic analysis to uncover the root-cause of exploits) under control-flow hijacks and why existing systems cannot obtain the required information.

Consider a real-world exploit that involves *control flow hijacking* through heap-based out-of-bounds (OOB) write and use-after-free (UAF) (CVE-2021-22555 [16]). The adversary exploits an incorrect type conversion in the *ip_tables* sub-module of the Linux *Netfilter*. A heap OOB write is triggered during a `setsockopt` system call. The attacker then crafts a `pipe_buffer` object—which contains a function pointer `ops`—and overwrites its pointer. When releasing the pipe through `close`, the attacker hijacks the kernel control flow to execute a chain of kernel functions that create escalated credentials and remove namespace restrictions for the calling process. For investigation of such an attack, the audit system should *capture both the system call and internal control-flow context, like exploited kernel functions* (Fig. 1).

Sadly, today's audit systems (Tab. 1) are designed to collect and securely maintain *coarse-granular information* within logs. While useful for investigating typical attacks (e.g., due to system misconfiguration), they make investigation ineffective against kernel control-flow hijacks.

In particular, conventional audit systems [11, 21] track information at *system call*-granular user-kernel interactions like network and file system. This is insufficient—for the control flow exploit, the logs will only report a set of commonly-occurring network and file-related system calls. Others track control-flow for auditing, but only during

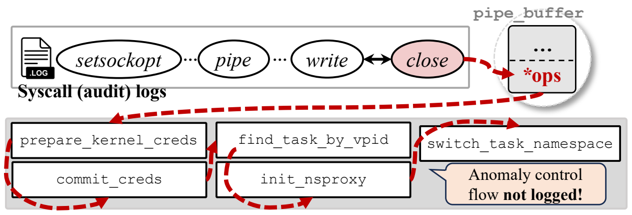


Figure 1: PoC exploit illustration of CVE-2021-22555 [16], including information (syscall entries) logged and missed by state-of-the-art audit systems.

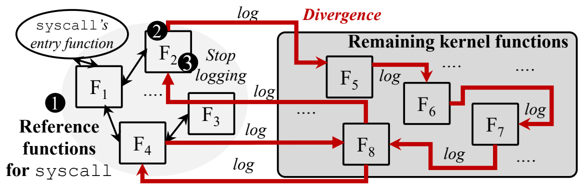


Figure 2: Illustration of our logging approach. F1-F4 are *reference* (allowed) functions for a `syscall`. Any jump to remaining functions is treated as a divergence and logged.

user-space execution of programs to resolve dependencies *between system call events* and improve forensic correlation. RAIN [12] and RTAG [13] enable record-and-replay among syscall events to resolve their dependencies. Similarly, PalanTir [14] deploys hardware processor tracing (using Intel PT [28]) to track program control flow, subsequently applying static taint analysis to resolve dependencies among system call logs. MARSARA [15] also leverages PT to validate the partitioned dependencies within system call logs. However, none of these systems aim to investigate control-flow *during* system calls, which requires a new approach (§4).

The other major recent focus of audit systems has been to ensure tamper-protection of logs against compromised systems (i.e., kernels). There are two main techniques employed: (a) leveraging efficient cryptographic constructions [22–24] to preserve the integrity of logs and (b) leveraging trustworthy environments either on the same machine (e.g., using Arm TrustZone [26]) or an external device [25] to preserve both integrity and availability of logs. The goal of these systems is only to preserve the coarse system call logs, which are generated by the kernel auditing software stack.

3. System and Threat Model

This work targets deployment on enterprise machines, where our attacker's goal is to compromise the kernel.

Auditing scope. Like prior auditing research, we assume that IT administrators will establish a set of *high-risk programs* as the target auditing scope (e.g., through internal risk assessment [29]). For example, a typical scope includes network-facing applications, such as web servers [30], script downloaders, and execution engines. Attackers will complete prerequisite attack steps (e.g., initial foothold establishment) and compromise one such userspace program. Through this program, we assume that *the attacker gains the ability to execute any system call with any parameters at any time*.

Attacker capabilities. The attacker is capable of launching *control-flow hijacks* during system calls. The requirements for such attacks are (a) unpatched kernel vulnerabilities and (b) the ability to decipher kernel memory layout and find gadgets (e.g., using KOOBE [31] and RetSpill [6]). By satisfying such requirements, attackers can bypass the default kernel security mitigations and redirect kernel execution to *chain arbitrary kernel functions* for privilege escalation. As soon as the attacker compromises the kernel, they will tamper with log generation mechanisms to hide traces of their attack [23]. After compromise, the attacker will leverage stealth techniques [32] (e.g., attack delegation) to evade detection and investigation. Thus, logs recorded after kernel compromise are not useful for investigation [25, 26].

Assumptions and out-of-scope. This work exclusively focuses on control-flow hijacking attacks—data-only attacks are out of scope. We also assume that the machine's hardware (e.g., CPU, motherboard, and DRAM) and firmware are correct. Hence, we exclude kernel compromise through micro-architectural defects [33] and fault injection (e.g., rowhammer attacks [34]). In addition, we assume that the OS is vulnerable but initially benign (i.e., written by honest developers).

4. Approach

Our goal is to design an audit system that captures fine-grained information regarding control-flow hijacks in the logs produced before a machine is compromised. To achieve this, we propose the approach of **tracing *anomalous* and *unlikely* control-flow during a system call**.

Anomalous control-flow refers to functions that should never be executed during a specific syscall's benign execution. Such executions can be found by traversing control-flow graphs (CFGs) produced by static analysis from the syscall entry point, and detecting those that are unreachable.

Unlikely control-flow refers to execution of functions that the CFG says are reachable, but are (1) potentially a false positive due to the imprecision in static analysis (e.g., over-approximated indirect jump resolution), and are thus "truly-anomalous" in such cases, or (2) belong to niche pathways that are seldom triggered under normal execution.

Fig. 2 illustrates the key concepts in our approach. We identify a set of ***reference behavior***, representing functions whose execution is *neither anomalous nor unlikely*, i.e., those **likely to occur during a syscall** (❶). Any control-flow transfer away from those functions is considered anomalous (❷). Upon such divergence, we log all anomalous control-flow transfers until execution returns to the reference behavior (❸). The rest of this section describes the rationale of our approach (§4.1) and its challenges (§4.2).

4.1. Rationale

The naive solution is to record *all* kernel control-flow during a syscall. Unfortunately, this incurs a significant slowdown and produces an overwhelming number of logs. We empirically evaluated this using two common programs, *Nginx* and *Redis*. Specifically, to trace control-flow during system calls, we individually leveraged (a) hardware-assisted tracing (using Intel Processor Tracing or PT [28]), which records control-flow transfers as hardware-generated packets, and (b) software tracing (using `ftrace` [35]), which uses compiler instrumentation to trap and log executed kernel functions. We deployed both to comprehensively understand the overhead using software and hardware approaches. In terms of workload, we used Redis-benchmark (100K requests with 1:1 key-value `SET:GET`) for *Redis* and ApacheBench (1K requests for a default file) for *Nginx*. We employed concurrent client connections ranging from 1 to 64 (see §8 for detailed machine specifications).

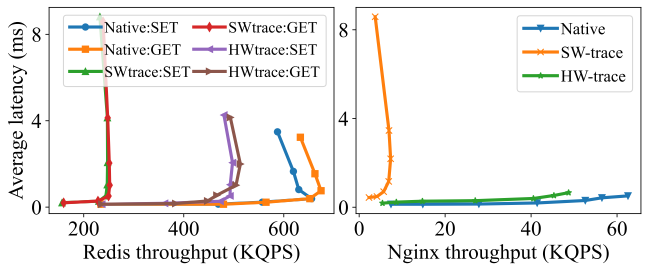


Figure 3: Runtime overhead of the strawman solution (indiscriminately tracing all kernel functions of each syscall).

Fig. 3 shows performance overhead incurred by the naive solution. Compared to native, software tracing slows Redis by 57.7% and Nginx by 85.9%. While hardware tracing is considered efficient [14, 36], it still imposes non-negligible overheads of 20.6% and 27.6%, respectively, given that it has to persist the high volume of trace data without any data drops (which is necessary for forensic analysis [37]). Moreover, full control-flow tracing also produces an enormous volume of logs—exceeding 133.6 MB/s (Nginx) and 78.1 MB/s (Redis)—up to 11.5× more than produced by typical audit systems (§9). This requires significant storage and processing power to adequately parse through all produced logs and detect potential control-flow hijacks.

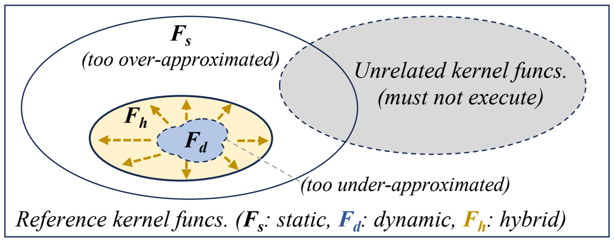


Figure 4: Illustration of using different approaches to identify a certain `syscall`'s reference kernel functions.

One way to address the aforementioned problems would be to record only *anomalous* control-flow based on static analysis. However, static analysis yields *significant over-approximation, causing precision problems* (F_s, Fig. 4). Due

Listing 1: A unlikely (niche) path in syscall `write`.

```
struct page *__alloc_pages(gfp_t gpt...) {...
   page = get_page_from_freelist(...);
   if (likely(page))
      goto out;
   /* Unlikely path triggered */
   page = __alloc_pages_slowpath(...);
out:...
   return page;
}
/* Dynamic call path (very infrequent) */
write --> __x64_sys_write() ... -> __alloc_pages()
   -> __alloc_pages_slow_path() ...-> shrink_zones()
```

to kernel complexity and the pervasive indirect jumps, identifying a syscall's reachable functions statically is difficult. For instance, even using one of the state-of-the-art analyses [38], we found over 45.8% and 68.6% functions can be reached from `read` and `write` syscalls. Such over-approximation would cause us to neglect logging considerable behaviors in anomalous control-flow hijacking (false-negative logging).

To capture a broader range of *truly anomalous* behaviors, it is also important to log *unlikely* control-flow transfers during system call execution (§4). This would mitigate the imprecision in the aforementioned static analysis.

Even when unlikely paths are benign, logging them remains practical since they are niche and rarely executed, adding minimal overhead with little risk of excessive logging. List. 1 shows a concrete example of such an unlikely and benign code path. Our empirical profiling reveals that the `write` syscall invokes `shrink_zones` in fewer than 3.3% of executions. This is expected, as `shrink_zones` is part of a fallback activated in the page allocator's slow path, which is triggered only when the kernel is under high memory pressure. Under normal conditions, such functions are rarely executed; including them in the control-flow logging scope would introduce minimal overhead (and false-positive logging). Thus, we only include "likely" paths into a syscall's reference behavior (§4).

4.2. Challenges

Given the findings in the previous section, we find it advantageous to log anomalous *and* unlikely control-flow during syscalls. However, as we describe in the subsequent paragraphs, this approach presents two non-trivial challenges.

Challenge-1: Identifying reference behavior for syscalls. Prior research in domains like kernel specialization discovers reference behavior for syscalls using dynamic workload profiling *only* [18–20]. However, this has severe problems (F_d, Fig. 4) in terms of under-approximation and scalability.

Under dynamic profiling, reference behaviors are derived by recording functions exercised by workload-specific inputs. However, even within a single program, syscall behaviors may vary due to complex kernel state. For instance, profiling Nginx even under *identical workloads* (same as §4) shows convergence only after ∼15 runs (Fig. 5). Moreover, different applications and workloads may behave differently under the same system call. For example, `read` may be used

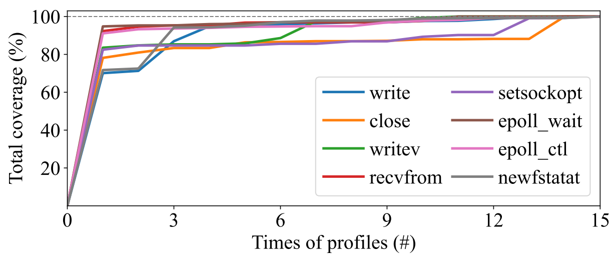


Figure 5: Dynamically profiled sysall behaviors using the same workload of *Nginx*, accumulated by profile times.

for filesystem access, network I/O, or both, depending on program behavior [19]. It is hard to profile all audit programs for deployment. Thus, per-program under-approximation triggers superfluous logging (false-positive).

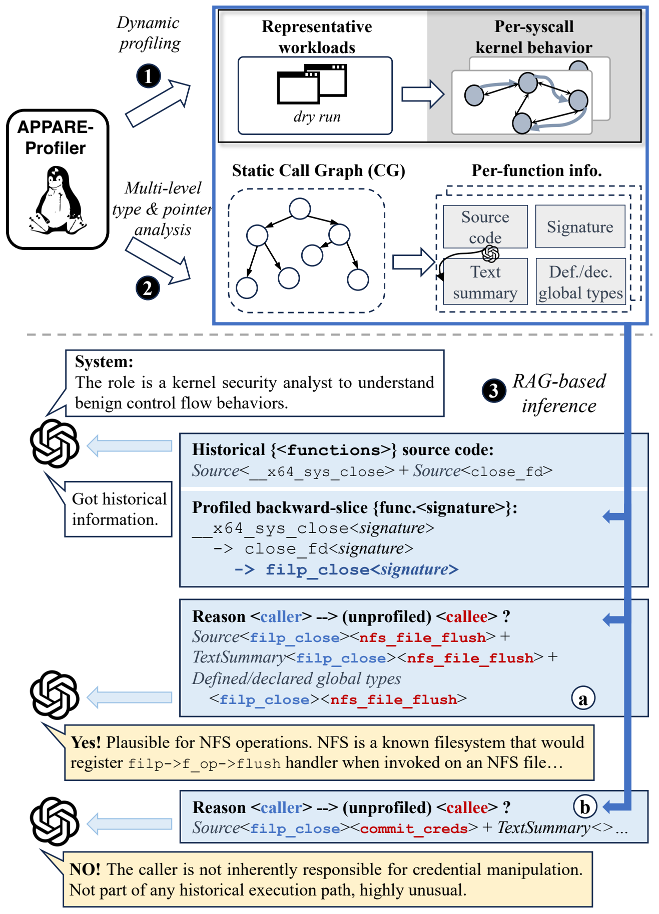
Challenge-2: Tracing control-flow securely and selectively. Once a suitable reference behavior is identified, any control-flow divergence from this behavior must be traced. Effective tracing must be *selective and secure* within syscall contexts, but achieving this is non-trivial. First, selectivity is difficult to enforce efficiently using existing software (`ftrace`) or hardware (PT) mechanisms. While both tracing mechanisms can be toggled on/off at runtime, they require the audit system to track each function executed within the kernel (e.g., to find anomalies). This defeats the initial motivation to selectively trace. Second, control-flow traces must be resilient against tampering to ensure accurate investigation. This is challenging because an advanced attacker could trick the vulnerable kernel (even before full compromise) to disable tracing or invalidate in-memory trace buffers [27, 39]. This limitation applies to both `ftrace` and PT-based logging.

5. Appare

Appare is a framework that efficiently records anomalous and unlikely control-flow traces to enable the investigation of attacks involving sophisticated kernel control-flow hijacks. To address outlined challenges (§4), Appare implements two key aspects: (a) a system call reference behavior identifier based on the augmentation of hybrid (static-dynamic) analysis with semantic reasoning (§5.1) and (b) a virtualization-aided processor tracer for secure, selective tracing of arbitrary kernel functions during execution (§5.2).

5.1. Augmented Reference Behavior Analysis

Appare derives generalizable syscall reference behavior (F_a, Fig. 4) by combining dynamic profiling and static analysis (i.e., *hybrid* analysis) with semantic reasoning. This augmented analysis is performed by our toolchain–Appare-Profiler–through a three-stage workflow (Fig. 6). In the first stage, the profiler takes a *small set* of representative workloads and executes them to generate dynamic syscall traces (❶, §5.1.1). In the second stage, the profiler constructs a comprehensive kernel knowledge base using



System:
The role is a kernel security analyst to understand benign control flow behaviors.

Historical {<functions>} source code:
Source<`__x64_sys_close`> + *Source*<`close_fd`>

Profiled backward-slice {func.<signature>}:
`__x64_sys_close`<*signature*>
`-> close_fd`<*signature*>
`-> filp_close`<*signature*>

Reason <caller> --> (unprofiled) <callee> ?
Source<`filp_close`><**`nfs_file_flush`**> +
TextSummary<`filp_close`><**`nfs_file_flush`**> +
Defined/declared global types
<`filp_close`><**`nfs_file_flush`**>

Yes! Plausible for NFS operations. NFS is a known filesystem that would register `filp->f_op->flush` handler when invoked on an NFS file…

Reason <caller> --> (unprofiled) <callee> ?
Source<`filp_close`><**`commit_creds`**> + *TextSummary*<>…

NO! The caller is not inherently responsible for credential manipulation. Not part of any historical execution path, highly unusual.

Figure 6: Overview of augmented reference behavior analysis: (❶) representative workload profiling, (❷) kernel knowledge base construction, and (❸) RAG-based code-semantic inference to explore uncovered functions. LLM inference (prompt) examples (ⓐ, ⓑ) are for demonstration purposes.

a compiler-based static analyzer, code extractor, and text summarization components (❷, §5.1.2). In the last stage, the profiler leverages this knowledge base and collected traces to classify potential unprofiled function candidates as *likely* or *unlikely* using semantic reasoning (❸, §5.1.3).

For control-flow reasoning, Appare employs a Retrieval Augmented Generation (RAG)-based code semantics inference approach [40], using a large language model (LLM) to (a) retrieve relevant extracted code semantics and (b) infer likely but unprofiled control-flow transitions, thereby expanding dynamic syscall profiles. Recent studies have demonstrated that LLMs can reason effectively about source-level code semantics and function behavior [41–43], achieving state-of-the-art results in tasks like indirect call prediction [44]. Building upon these insights, Appare evaluates whether an unprofiled function could plausibly (*likely*) continue a syscall's observed control flow.

Please note that Appare's inference approach does not compromise precision with respect to the kernel's control-flow graph (*CG*). The inference process only considers functions that are valid control-flow targets—i.e., functions corresponding to actual edges in the CG.

5.1.1. Dynamic representative workload profiling.

APPARE-PROFILER gathers kernel function execution traces triggered by individual syscalls of representative workloads. We define representative workloads as those that involve feature-rich OS interactions, including filesystem access, inter-process communication (IPC), device I/O, and networking. Representative workloads employed throughout our experiments are detailed in §6, and our empirical study (in §7.1) shows that a small set of workload profiles can generalize to typically audited real-world programs.

The profiler generates traces by executing representative workloads and capturing all functions invoked during syscalls. The collected syscall execution traces provide concise and targeted seeds ($F_d \rightarrow F_h$, Fig. 4) for subsequent reference set expansion. Crucially, the seed-based design restricts the scope of RAG-based LLM inference, eliminating the need to encode the full kernel callgraph, a task that is impractical due to LLM context-length limitations [44, 45].

It is important to ensure workload profiling remains free from adversarial interference, as otherwise the resulting initial reference behavior set may be contaminated with "anomalous" control flows. To this end, APPARE's profiling is carried out in a controlled environment (e.g., isolated virtual machines) and is conducted offline prior to production deployment. In practice, this profiling phase is a *one-time* effort per-kernel version. We expect IT administrators (in enterprises where auditing is deployed) to integrate such profiling into their software testing pipeline.

5.1.2. Static kernel knowledge base extraction.

To facilitate semantic inference, APPARE constructs a static kernel knowledge base encompassing both the structure of kernel control flows (i.e., a static call graph CG) and the semantic information of functions. The generated CG guides the LLM in identifying potential yet *unobserved control-flow paths adjacent to dynamically profiled seed functions*, while associated code information offers essential semantic context to enhance reasoning accuracy.

APPARE utilizes type- and pointer-based analysis techniques [38, 44] to generate the static CG with future opportunities to incorporate more sophisticated methods for further precision improvements [46, 47]. For each kernel function, APPARE extracts the following information:

1) *Function source code and signatures*;
2) *Textual semantic summary of relevant functions*, generated by the querying LLM from source code;
3) *Global kernel context* such all declared or defined global object types (structures). This enhances the LLM's ability to discern high-level semantic relationships, a capability validated by recent studies [44, 45].

The aforementioned semantic information significantly guides the LLM toward precise inference. According to our validation evaluation (§7.2), providing such comprehensive context can improve code inference accuracy by over 14%.

Algorithm 1: Syscall reference set identification.

```
1  Function EnhanceReferenceSet
2      Input: A syscall's profiled behavior set ($\mathcal{F}_i$), static call graph $CG$, maximum hops $N$
3      Output: Enhanced syscall reference set $\mathcal{F}_e$
4      /* Derive a dynamic call graph */
5      $CG_f \leftarrow$ GenerateSubGraph ($CG, \mathcal{F}_i$)
6      /* Sort $CG_f$ nodes in reversed topology order */
7      $\mathcal{F}_{sorted} \leftarrow$ ReverseTopoSorting ($CG_f$.nodes)
8      $\mathcal{S}_{queried} \leftarrow \{\}$
9      $\mathcal{F}_e \leftarrow \mathcal{F}_i$
10     foreach $F \in \mathcal{F}_{sorted}$ do
11         $\mathcal{S}_{frontier} \leftarrow \{F\}$
12         foreach hop in 1...$N$ do
13             $\mathcal{S}_{next} \leftarrow \{\}$
14             foreach $L \in \mathcal{S}_{frontier}$ do
15                 /* Get (unprofiled) callee<$V$> of caller<$L$> */
16                 while $V \leftarrow$ PopCallTargets($CG, L$) do
17                     if $\neg CG_f$.hasEdge($L, V$) and $(L, V) \notin \mathcal{S}_{queried}$ then
18                         /* LLM-based reasoning (Fig. 6) */
19                         ANS $\leftarrow$ QueryLLM($CG_f, L, V$)
20                         if ANS then
21                             /* Enlarge the referential set */
22                             $\mathcal{F}_e \leftarrow \mathcal{F}_e \cup \{V\}$
23                             $\mathcal{S}_{next} \leftarrow \mathcal{S}_{next} \cup \{V\}$
24             $\mathcal{S}_{frontier} \leftarrow \mathcal{S}_{next}$
```

5.1.3. LLM-driven code semantics inference.

To predict *likely* unprofiled functions within syscall executions, APPARE applies a RAG approach by querying the LLM with semantically-related code context and static CG traversal. Algo. 1 (and Fig. 6) illustrate the workflow.

APPARE first gathers all profiled seed functions, and represents them as a dynamic call graph (CG_f, line 5). It then processes each seed function individually using a bottom-up approach, by applying a reversed topological sort over CG_f (lines 7–9). For each individual seed node (e.g., `filp_close` in Fig. 6), APPARE considers the maximum of its next N depths (hops), by forward traversal on the kernel CG (lines 10-14). All the traversed callees are the potential "likely" candidates (lines 16-18) (e.g., `nfs_file_flush` in Fig. 6). To perform code inference, APPARE prompts the LLM with the semantics information of both the caller and the callee, as well as the historical context information from the dynamic call graph CG_f. To do so, APPARE takes a backward-slice (from the seed node) of the CG_f, inputting the structure of this backward-slice (see **profiled backward-slice** in Fig. 6), as well as all functions' source code within the backward-slice. Then, APPARE completes the prompt by inputting retrieved code semantics information of the caller and callee (§5.1.2, also shown as ⓐ in Fig. 6). Last, APPARE queries the LLM (lines 18-23), expanding the syscall's reference functions according to the inference decision.

5.2. Virtualization-Aided Processor Tracing

APPARE employs a security monitor executing at the hypervisor privilege mode, called APPARE-LOGGER, to support efficient, secure, and selective tracing. Such *virtualization-based* monitors are widely adopted for kernel security by industry [26, 27]. For example, Windows 11 enables a trusted hypervisor by default to support Virtualization-Based Security (VBS) [48], which defends against page-remapping attacks and verifies loaded kernel drivers. Similarly, Android/Linux introduced protected KVM (pKVM) to provide equivalent protection [49]. APPARE can be integrated with these existing security mechanisms to support robust kernel auditing.

At a high-level, APPARE-LOGGER enforces **restricted kernel code views depending on the executing syscall** to trap and selectively trace control-flows. In addition to leveraging Processor Tracing (PT), this is achieved by using three widely-available hardware primitives:

- *Memory view protection*—Extended Page Tables (EPT) and IOMMU—to restrict kernel and device access to protected or non-reference regions. CPUs today, by default, use them to isolate VMs from the host.
- *Historical branch counters*—Last Branch Record (LBR)—to efficiently restore partial historical contexts before anomalous control flow. Like PT, such features are also common in modern CPUs for branch profiling.
- *Synchronized hardware timestamp counter*—instructions like `RDTSCP`—to ensure time can be unified across the entire system and propagated to all (system call and control-flow) logs for chronological correlation (§6).

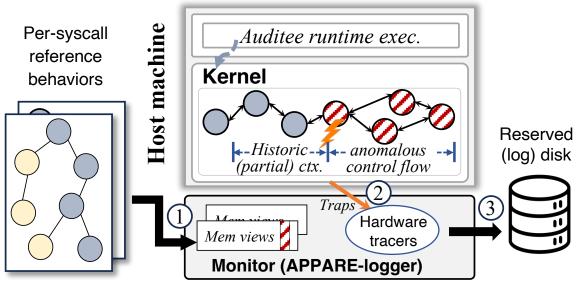


Figure 7: Overview of virtualization-aided processor tracing.

Fig. 7 shows the workflow of APPARE-LOGGER. Initially, the logger configures EPT interfaces based on the generated syscall reference behaviors to establish per-syscall kernel memory views for the OS (①, §5.2.1). During an audited program's execution, anomalous control-flow triggers automatic traps to the logger, which toggles (a) PT to enable selective tracing and (b) LBR to restore partial historical contexts (②, §5.2.2). The historical contexts are helpful to further investigate whether anomalous control flow occurs due to false-positive or true attack behaviors (§7.3). All logs are securely stored in protected memory and disk regions, while tracing interfaces are protected against OS-level interference (③, §5.2.3). Finally, analysts retrieve these logs for further investigation (using APPARE-PARSER, detailed in §6).

5.2.1. EPT-based restricted code view maintenance.

At system initialization, APPARE-LOGGER generates *restricted* kernel code pages for each system call, in addition to the normal kernel (*unrestricted*) code pages. Unlike normal (unrestricted) pages, for each syscall, its restricted code pages only contain kernel functions within that syscall's reference behaviors. The remaining (unprofiled kernel functions) are overwritten to undefined instructions (`#UD2`) (▨, Fig. 8a).

APPARE-LOGGER preloads both restricted and unrestricted code pages in reserved memory at boot. It creates multiple EPTs: one mapping the guest's physical addresses to normal (unrestricted) pages, and additional EPTs mapping to per-syscall restricted pages. Thus, the logger generates $n + 1$ EPTs, where n is the number of system calls, and the additional EPT corresponds to unrestricted kernel pages. Each EPT is installed within the *EPT-pointer list* (EPTP-list) of the virtual machine control structure (VMCS) by the logger. This EPTP-list is used to perform fast exitless switching (explained in §5.2.2) for selective tracing.

5.2.2. Trace toggle by view-switching and guard stack.

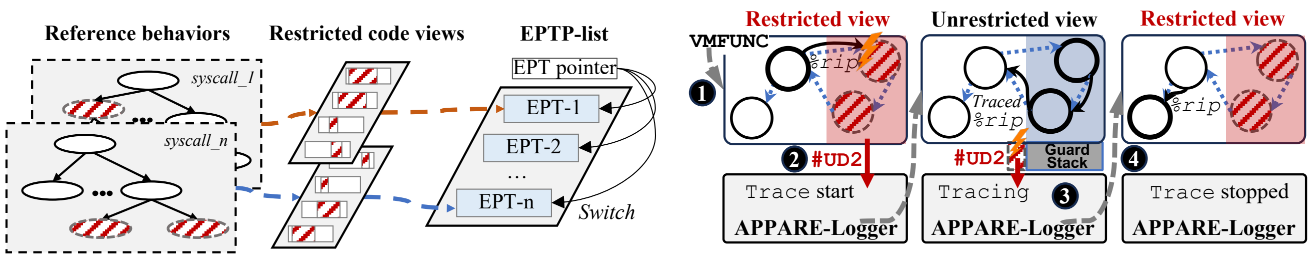
At runtime, APPARE enforces switches of kernel code pages between restricted and unrestricted versions with the kernel's help. When an audited program executes a system call, the kernel executes `VMFUNC` [39, 50] to switch its code-view to restricted pages based on the system call (❶, Fig. 8b). The `VMFUNC` instruction is inserted into the kernel by lightweight compiler-based instrumentation. This instruction is an optimized version of a hypervisor-call (`VMCALL`) that switches to a different EPT (within the EPTP-list) without exiting to APPARE-LOGGER (hypervisor).

Note that, the attacker may try to abuse the `VMFUNC` instruction to switch back to unrestricted code pages. To prevent this, `VMFUNC` is immediately made "inaccessible" in the restricted pages, once it is performed at the system call entry. Specifically, the compiler *always aligns* `VMFUNC` *at the end of a page* using `NOP`. Hence, after its execution, the instruction pointer will move to the next page. This allows the logger to mark the previous (`VMFUNC`-containing) page as inaccessible in the EPTs of restricted code pages [51].

During the execution of a restricted code page, any control-flow transfer to a function outside the reference behavior raises a `#UD2` violation, which is trapped by the logger (❷, Fig. 8b). Upon trapping, the logger sequentially:

1) Generates a log entry to save the timestamp when tracing begins (used later for chronological correlation);
2) Dumps the current kernel's historical branch traces (preceding the anomaly) by using LBR interfaces;
3) Starts tracing control-flow using the PT interfaces;
4) Switches the kernel's code view to unrestricted pages;
5) Resumes kernel execution (with tracing active).

To stop tracing when the anomalous control-flow ends, APPARE-LOGGER employs a guarded stack strategy. Specifically, it replaces the kernel stack via the VMCS's stack pointer (`rsp`) [39] with a new stack frame that (a) does not contain a return address and (b) is preceded by a *guard* frame containing `#UD2` (similar to the concept of "RedZone" [52]). Hence, when anomalous control-flow returns to the calling function, the stack pointer hits the guard frame, and a fault is caught by the logger (❸, Fig. 8b). Note that the attacker can potentially tamper with the new stack, but doing so would only mean that tracing is never disabled. On the caught fault, the logger disables tracing, copies return values between the



(a) Multiple kernel code views maintained by EPTs.

(b) Control flow tracing enabled/disabled by `#UD2`.

Figure 8: APPARE's selective secure control-flow tracing. (a) The monitor maintains restricted code pages that are outside per-syscall reference behaviors (i.e., code replaced by `#UD2` instructions, ▨), by using an EPTP-list (a set of EPTs). (b) At a syscall entry, `VMFUNC` is performed to switch the current EPT to its corresponding restricted view for trapping/logging.

stack frames, and switches the kernel's code-view back to restricted (❹, Fig. 8b). Afterwards, any further anomalous control-flow will again be caught and traced similarly.

5.2.3. Protected trace configurations and regions.

APPARE ensures the integrity and availability of traces throughout end-to-end collection. This is achieved by preventing the untrusted OS from **(a)** manipulating or disabling tracing during anomalous control-flow; **(b)** tampering with the traced PT packets or the LBR branch information in-memory; **(c)** overwhelming the system to overflow traced PT packets in memory before they are persisted; or **(d)** corrupting logs after they have been persisted on storage.

APPARE-LOGGER leverages virtualization features [39] to ensure **(a)** and **(b)**. Specifically, the logger configures a virtualization trap on writes to all PT configuration-related model-specific registers (e.g., `IA32_RTIT_CTL`) to prevent their misconfiguration by the untrusted kernel. Such a trap is also applied to all LBR interfaces (e.g., `IA32_DEBUGCTL`) and recorded branch information (e.g., `MSR_LASTBRANCH_*`). The logger additionally leverages EPT and *PT-passthrough* to secure the traced control-flow packets in memory. In particular, the logger reserves *per-core* physical memory regions for trace packets using EPT. Per-core division avoids concurrency delays. Only the PT hardware (on each core) is allowed to access its own per-core reserved region. This is achieved by configuring PT (using its model-specific registers) to bypass EPT and write only to allocated physical regions using the Top of Physical Address (ToPA) buffering feature [39].

To ensure **(c)**, APPARE-LOGGER configures the PT hardware to generate a *non-maskable* Performance Monitor Interrupt (NM-PMI) [14, 53] when a *per-core* PT buffer is full. This NM-PMI is exclusively handled by APPARE-LOGGER, which then leverages a standard dual buffering and asynchronous persistence strategy [26]. APPARE-LOGGER distributes a background thread to persist the *previous* buffer, while switching PT to generate packets to the *next* buffer. In rare scenarios when both buffers are full, APPARE-LOGGER waits for a buffer to be persisted before resuming.

Finally, for **(d)**, APPARE-LOGGER leverages standard hypervisor mechanisms to reserve a protected log storage disk. Specifically, the hypervisor omits mappings of the log device's MMIO in the EPT, causing any guest access attempts to trigger EPT violations. When PCI passthrough is enabled, the hypervisor configures the IOMMU (VT-d) to restrict DMA to guest-assigned devices only, thereby ensuring that protected log disks cannot be accessed by the guest [26, 54].

TABLE 2: Base software framework and the source code lines we added or changed for different APPARE components.

Component	Base	Version	SLoC
APPARE-PROFILER	FTrace/LLVM/Ollama	–/15.0/0.9	1.7k
APPARE-LOGGER	Linux-KVM	5.19.0	2.5k
APPARE-PARSER	Distorm [55]	3.3	1.5k

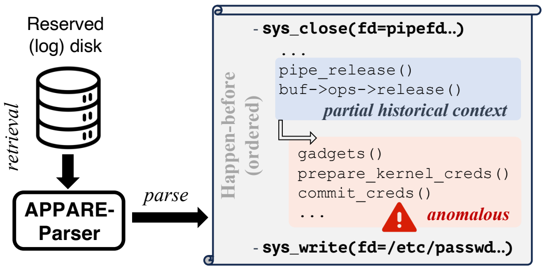
6. Implementation

We implemented a prototype of APPARE for Linux operating system kernels running on the Intel x86-64 architecture. Tab. 2 shows our prototype's lines of code.

APPARE-PROFILER. The profiler first uses Linux's built-in tracer (`ftrace` [35]) to generate control-flow profiles for different system calls executed by a program. It inserts a profiler routine at the start of each kernel function, which records the function's execution during a system call. We adopt `ftrace` since it is natively supported in modern Linux, making it easy to use. Although `ftrace` is not performance-optimal, this choice is acceptable because profiling is performed offline and is not performance-critical. Then, the profiler leverages LLVM to extract a kernel knowledge base and employs Ollama [56] to deploy the Qwen3-32b LLM for its RAG-based hybrid augmentation. Specifically, we employed MLTA [38], a state-of-the-art approach to generate the kernel static call graph. This can be further refined using more advanced techniques (e.g, KallGraph [46]) when they become available. We chose Qwen3-32b given that it is one of the state-of-the-art open-sourced LLMs with coding and reasoning capabilities [57]. Any other advanced large language models can be applied to APPARE.

APPARE-LOGGER. The logger is implemented by extending KVM. In principle, a lightweight single-host-based virtualization approach [58–61] is more suitable, instead of using a full hypervisor like KVM. We used KVM (like prior research [26]) to easily implement aspects like reserved log storage. During initialization, APPARE-LOGGER reserves memory for creating EPTs with different kernel code views and per-core PT buffers to collect control-flow traces (§5.2). The logger configures the system to always enable LBR (only at kernel-mode execution, by editing `MSR_LBR_SELECT`). The logger also incorporates a *memory scanner* to decipher the runtime kernel code page layout and store it for later analysis. This is required because traces collected by PT must be correctly translated into specific kernel functions for analysis (next heading). For this to work, the kernel layout must be saved after its code self-update (e.g., `text_poke`) and randomization (i.e., KASLR) at boot-time [36].

APPARE-PARSER. This component is designed to retrieve and parse the logs for investigation. The result is a temporal graph of events following *happens-before* semantics that contains nodes for syscall audit logs, anomalous control-flow (recovered to function/basic block-level), alongside its partial historical control-flow contexts (shown below).



```
- sys_close(fd=pipefd..)
  ...
  pipe_release()
  buf->ops->release()
    gadgets()
    prepare_kernel_creds()
    commit_creds()
    ...
- sys_write(fd=/etc/passwd..)
```

The parser relies on Distorm [55] to disassemble the loaded kernel code layout and decode PT packets to basic blocks and functions. We applied the *basic block pointer pages* strategy in Griffin [62] to enable fast control flow recovery. Once APPARE-generated logs are fully recovered, the analyzer leverages the common protected timestamp installed within them to unify them and the native syscall audit logs. Administrators may use graph-based threat hunting [63] and trace-based root-cause analyzers [53] to determine anomalies and attack origins. Security analysts may also perform manual analysis (as demonstrated in §7.2).

Limitations. Our prototype does not currently support the guard stack (§5.2). Instead, we invoke a `VMCALL` at the end of syscalls or context switches to ask APPARE-LOGGER to stop PT (if enabled). Note that this does not favor our evaluation; rather, it slightly increases the performance overhead.

APPARE does not currently handle hardware interrupts. All interrupt-handling functions are added to each syscall's reference set. Thus, there are no restrictions on interrupt behaviors. Our prototype cannot detect anomalous control flow within those contexts (further shown in §7.3). We discuss potential solutions in §10.

7. Security Evaluation

In this section, we first individually demonstrate the system's logging capability in terms of anomalous kernel control-flow. Then, we further validate our analysis using case studies on real-world exploits.

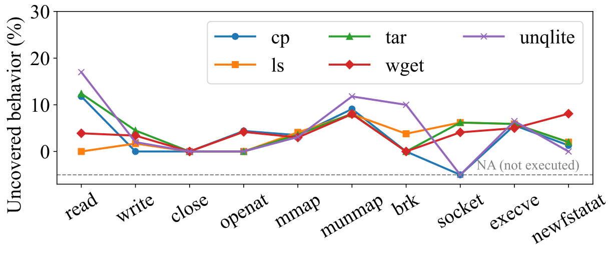


Figure 9: The extensibility of representative workloads (§7.1) to other real-world programs. For each program shown, we report runtime syscall behavior divergence (uncovered) based on profiled behaviors from representative workloads. We listed frequently executed syscalls across selected programs.

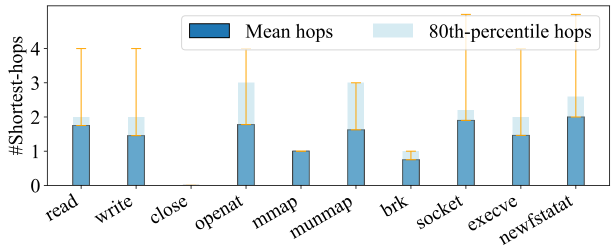


Figure 10: Mean and 80th-percentile of the minimum hop distance to the unprofiled functions (executed by programs in Fig. 9), measured by forward-traversing the kernel CG from profiled functions (via representative workloads in §7.1). We listed frequently executed syscalls across selected programs.

7.1. Settings: Reference Behavior Identification

APPARE's logging capability is grounded in its syscall reference behaviors. This section describes the configurations and variants used to generate those reference behaviors.

Representative workload settings. Those workloads are profiled to generate a set of seed functions (§5.1.1). We rely on the following representative workloads (through all experimental evaluations in this paper):

- Linux Test Project (LTP) [64]: OS developers-maintained common syscall test suite, offering a broad syscall corpus.
- Nginx and Redis: complex server-side applications with diverse behaviors on the host machine (workloads in §4).

The reason is that those workloads have feature-rich interactions with the OS. To show this, we empirically profiled their syscall behaviors and evaluated how well they generalize to other real-world programs. As shown in Fig. 9, the representative workloads cover over 91% of syscall behaviors across the selected programs on average, with at most 17% of runtime functions uncovered. We discuss other profiling strategies in §10.

RAG-based inference settings. Given the aforementioned profiled seed functions, APPARE employs its RAG-based LLM inference to traverse a maximum N hops of the kernel static call-graph CG and incorporate unprofiled but likely functions (§5.1.3). An overly large N not only significantly

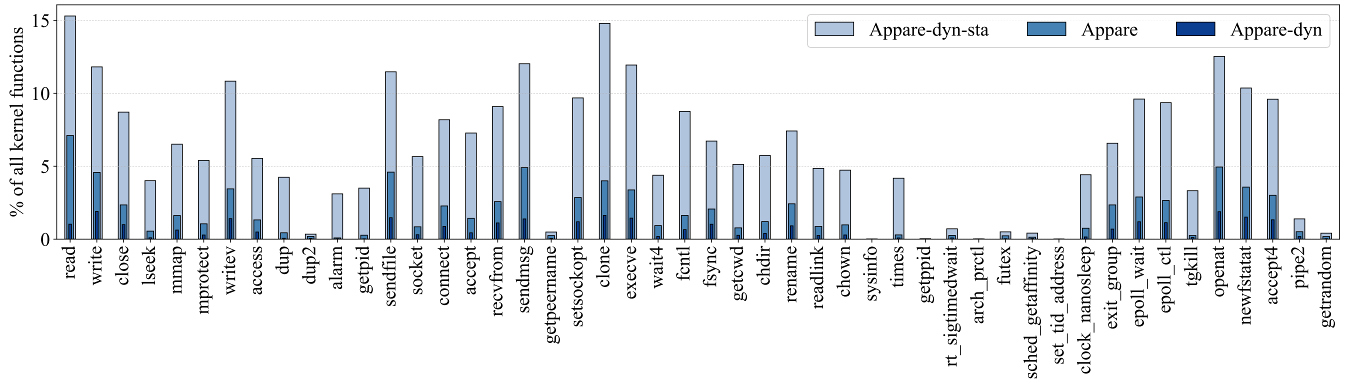


Figure 11: Percentage of identified reference behavior functions amongst all kernel functions for each system call.

enlarges the reference behavior set (potentially introducing false-negative logs), but also incurs high LLM inference cost due to CG traversal path explosion. To address this, we empirically studied the distribution of uncovered functions during real-world programs' (same as programs in Fig. 9) syscall execution based on profiled behaviors. As shown in Fig. 10, uncovered functions are reachable from profiled ones within 2 hops on average across syscalls. Furthermore, over 80% of reachable functions are approximately $\leq$2.23 hops from profiled ones. Therefore, to balance security (i.e., approximating reference behaviors) and inference cost, we set APPARE's maximum traversal depth (N) to 2 hops.

Variant settings. To quantify our approach's effectiveness, we evaluated three APPARE variants for reference behavior generation: (a) dynamic profiling only (**APPARE-dyn**), (b) our approach in §5.1 (**APPARE**), and (c) naive static expansion: traverse the kernel's CG within N $(= 2)$ depths from dynamically profiled functions and include every function encountered without semantics inference (**APPARE-dyn-sta**). We also evaluated their performance in §8.2.

7.2. Reference Behavior-based Logging Capability

Reference behavior statistics. APPARE leverages hybrid augmentation to find reference functions for syscalls (§5.1). We analyzed how much of a control-flow logging capability is retained by APPARE through this approach, and performed ablation studies on different variant settings.

Fig. 11 shows the statistics of syscall reference behaviors. Across all syscalls, APPARE identifies reference behaviors covering merely $0.01\% - 7.11\%$ of kernel functions (1.73% on average). Even in the worst case, the `read` syscall's reference behaviors contain 7.11% (2974/41837) of kernel functions. Moreover, by augmenting dynamic profiles to mitigate its under-approximation, APPARE expands reference coverage by 2.83$\times$ compared to APPARE-dyn. Last, compared to naive static expansion without semantics inference (**APPARE-dyn-sta**), APPARE reduces the reference-set size by up to 97.2% (75.1% on average), thereby maintaining significantly better anomaly detection capability. Performance evaluations in §8 further show that this reference behavior scope reduction only adds small overhead.

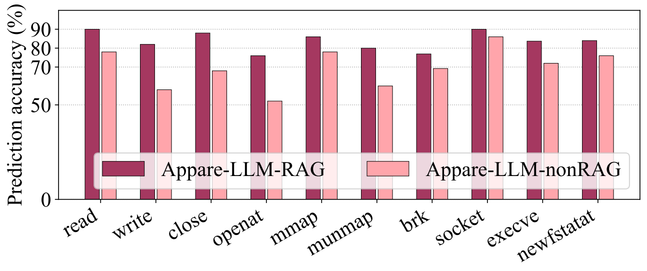


Figure 12: Validation accuracy of APPARE's LLM-driven reference behavior inference.

Takeaway. For control-flow hijacks, after leveraging an initial vulnerability, the attacker must (a) use ROP/JOP gadgets to prepare payloads, and (b) execute critical kernel functions to complete exploits (e.g., obtain root access through functions). They will evade APPARE's logging if (and only if) they can find both within the tiny scope of functions $(0.01\% - 7.11\%)$ accessible within the system call reference behaviors. By RAG-based code semantics reasoning, such a scope is also significantly reduced (by up to 97.2%) compared to one without semantics reasoning.

Inference accuracy validation. As APPARE leverages LLM to reason about the code semantics and enhance the reference behaviors, it is important to understand the accuracy. Given that there is no ground-truth for the inference behaviors, we used the dynamically profiled functions for validation. We selected 10 syscalls and, for each, randomly sampled 50% of the "caller $\rightarrow$ callee" pairs observed in their profiled traces. We queried the LLM to determine whether each sampled caller $\rightarrow$ callee is plausible given its dynamic context.

To understand the benefit of RAG-based LLM inference, we compared two prompting strategies: supplying the LLM with both the extracted kernel knowledge base and trace context (**APPARE-LLM-RAG**, Fig. 6), and providing only the caller and callee source code without historical or semantic context (**APPARE-LLM-nonRAG**).

Fig. 12 reports the results. RAG-based inference achieves 83.7% validation accuracy versus 69.7% without RAG. These results align with prior work [44], indicating that contextual code knowledge (e.g., historical call-stack summaries and the

TABLE 3: APPARE's logging capabilities in kernel exploits.

PoC CVE	Exploited Functions
V1: 2021-26708	dw_dma_initialize_chan (✓)
V2: 2021-4154	switch_task_namespaces (✓); prepare_kernel_cred (✓) commit_creds (✓)
V3: 2021-22555	switch_task_namespaces (✓); prepare_kernel_cred (✓) commit_creds (✓)
V4: 2021-42008	__request_module (✓)
V5: 2021-43267	__request_module (✓); regcache_mark_dirty (✓)
V6: 2022-0185	find_task_by_vpid (✓); prepare_kernel_cred (✓) commit_creds (✓)
V7: 2022-0995	prepare_kernel_cred (✓); commit_creds (✓)
V8: 2017-6074	native_write_cr4 (✓); prepare_kernel_cred (✓) commit_cred (✓)
V9: 2022-25636	prepare_kernel_cred (✓); commit_creds (✓)
V10: 2023-2598	call_usermodehelper_exec (✓); queue_work (✓)
V11: 2022-1015	__do_softirq (✗); commit_creds (✗)

function's global summaries) is crucial for inference. Future work could boost accuracy by using a more powerful LLM (than the current QWen3-32b).

7.3. Exploit Case-Study and Validation

This section further analyzes APPARE's logging capability using case-studies on real-world exploits.

Attack setup and payload category. We randomly selected a total of 11 real-world CVEs, accompanied by their corresponding PoCs, from public exploit repositories. The selected vulnerabilities contain various types, including stack and heap out-of-bounds (OOB) write, use-after-free, and double-free. Tab. 3 shows the concrete list of CVEs. We used syscall reference behaviors generated from §7.2.

Exploit logging analysis. Based on our selected vulnerabilities and their real-world PoC exploits, our analysis (Tab. 3) shows that APPARE can log in-memory behavior of anomalous control-flow (from PT) in the vast majority of exploits (except for V11).

We found that none of the kernel functions used in these exploits were within APPARE's reference behaviors of the corresponding syscall(s) during the exploit, and thus, all control-flow hijacks can be logged. This is expected: the commonly used gadget functions for exploits, like `prepare_kernel_creds` and `commit_creds`, are only used in certain scenarios (e.g., during kernel thread creation). However, as our prototype ignores interrupt contexts (§6), exploits within such contexts (like V11) evade detection. We discuss addressing this limitation in §10.

In contrast with APPARE, existing auditing solutions (e.g., Auditd [11] and OmniLog [26]) capture only syscall entries rather than function-level information. As a result, they do not report *any* of the functions in Tab. 3.

Case study 1: CVE-2021-4154. To validate our analysis with APPARE's prototype, we reproduced a PoC attack that leveraged CVE-2021-4154. This exploit manipulates sensitive objects and gains control-flow hijacking capabilities, which are then used to compromise the kernel. Due to the different memory and object layouts among different kernel and compiler versions, the reproduced resulted in a kernel crash instead of fully compromising the kernel. Note, however,

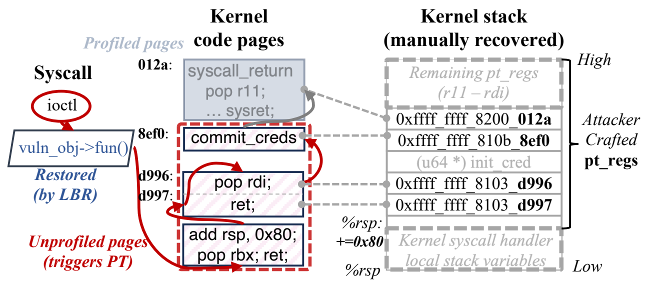


Figure 13: Case study of RetSpill's control flow hijacking [8] with APPARE enabled (red control flows are logged).

that crashing is a common indication of a successful kernel exploit [31, 65]. Nevertheless, before the crash, APPARE successfully logged kernel functions in the exploit payloads.

Case study 2: RetSpill [6]. To show the effectiveness of APPARE, we reproduced RetSpill's crafted attack scenario, which exploits the kernel with Linux's KCFI deployed [8]. Current KCFI implementations enforce control flow only at a coarse granularity, allowing any indirect call or jump to target any valid function entry point. Therefore, attacks like RetSpill would hijack the control flow to a controlled function entry point. It then never relies on the ability to hijack forward-edge control flow again, but uses data spilled on the kernel. We enabled APPARE and manually analyzed the logs (results in Fig. 13). In such a scenario, a vulnerable kernel driver's `ioctl` interface allows overwriting a function pointer to a malicious gadget. Before triggering that function pointer (by an `ioctl` syscall), the adversary spills all the gadgets (a sequence of target instructions and kernel functions) onto registers (`pt_regs`), which will be pushed to the kernel stack at the target syscall entry. According to our analysis in Fig. 13, all gadgets and target functions (`commit_creds`) were outside of the reference set and thus captured. By inspecting the recovered historical contexts (from LBR), it also reveals that the anomalous paths are triggered by an indirect function pointer of `vunl_obj->fun()`.

7.4. Threat to Validity

APPARE's RAG-assisted reference behavior augmentation can introduce false-positive functions into the reference set. Attacks composed entirely of functions that lie inside syscalls' reference sets would evade detection. LLM-based inference may propose functions that are semantically related but never executed in the actual syscall context. For example (List. 2), syscall `execve` loads an executable file into the memory for new process execution. It invokes `load_elf_binary` and thus legitimately exercises file-backed memory mapping (`ext4_file_map`). It should not execute backend-specific mapping routines (e.g., GPU backend or *procfs* mappers). However, APPARE's LLM augmentation also adds these functions, and therefore over-approximates the reference set. The false-positive enlarges the set of "allowed" operations. An adversary can chain functions

Listing 2: An false-positive inference example of LLM-RAG. Green blocks: true-positives; yellow blocks: false-positives.

```
/* for syscall execve() */
__x86_sys_execve -> /* true-positives */
  do_execveat_common ->
    bprm_execve ->
      load_elf_binary ->
        elf_map ->
          ... ->
            ext4_file_map |
            i915_gem_mmap | /* false-positives */
            proc_reg_mmap
```

that remain within the reference set (across multiple syscalls) to construct an exploit that bypasses APPARE's detection.

8. Performance Evaluation

This section describes APPARE's performance under different settings. All experiments were conducted on an Intel Xeon Gold 6430 server. The machine was configured with 128GiB memory and 512GB NVMe SSD storage. For RAG inference, we employed a Nvidia H100 NVL GPU with 94GiB memory to deploy the LLM locally.

We configured our target virtual machine (running the audited operating system and applications) with 8 virtual processors (vCPUs), 8GiB memory, and a 100GB virtualized (virtio) storage disk. The remaining resources of the server were used to generate client workloads (e.g., send user requests from the host to the Nginx webserver running within the guest VM) in a saturated manner.

Evaluation settings. All APPARE-related settings (reference behavior generation) and variants (including **APPARE**, **APPARE-dyn**, and **APPARE-dyn-sta**) are aligned with §7.1.

To compare APPARE with existing system call-based audit systems, we reproduced and employed the state-of-the-art **OmniLog** [26] and Linux's default **Auditd** [11]. Since OmniLog is designed for a different kernel version, we ported it to our kernel version. Our port does not include OmniLog's proposed compression method [26], but note that doing runtime compression increases performance overhead. Since APPARE depends on conventional audit systems to generate syscall logs, we also measured the overhead of **OmniLog+APPARE** to show a setting where all logs are captured (and also protected against the kernel).

We evaluated all aforementioned settings within the virtual machine with device (storage) passthrough. This is a common practice for virtualization monitor-based systems to avoid the impact of device emulation [26].

8.1. Micro-Benchmarks

Reference behavior inference time overhead. We measured APPARE's per-syscall inference time (§5.1.3) with a kernel call-graph traversal depth of N=2 (§7.1). On average, APPARE requires 2.44 hours to generate the reference behavior for one syscall. Fig. 14 lists the ten slowest syscalls,

TABLE 4: APPARE-LOGGER operation cost breakdown.

Runtime operation	Detailed tasks	Cycles
System call entry		
Switch EPT	Execute `VMFUNC` in entry function	310
Anomalous control-flow		
Enable PT on #UD2	Handle `#UD2`, write `IA32_RTIT_CTL`	3020
Disable PT on return	Handle fault, write `IA32_RTIT_CTL`	3075
Switch full buffer	Handle NM-PMI, switch buffers	4964

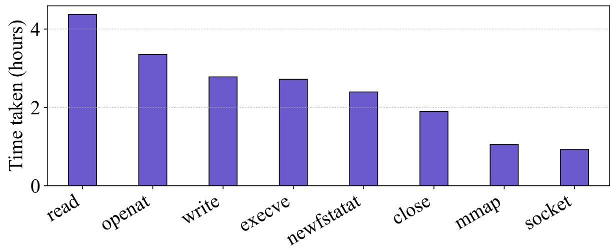


Figure 14: Top-10 inference time-consuming syscalls during augmented hybrid reference behavior analysis (§5.1).

whose inference times range from 0.9 to 4.4 hours, due to their complexity in the kernel call graph. Nonetheless, this is a one-time effort (§5.1) performed per kernel.

Runtime operation cost breakdown. APPARE executes several runtime operations to trace anomalous control-flow during a system call (§5.2). To measure their cost, we wrote a custom benchmark that executed the `getpid` system call in a tight loop for a million iterations to count the average. During each system call, we artificially enabled anomalous control-flow tracing by restricting a fixed kernel code page. We measured all times using `RDTSCP`.

Tab. 4 shows the results. At a system call entry, a `VMFUNC` is performed to enforce kernel function and object profiles (310 cycles). When the kernel executes the restricted code page, it raises a `#UD2` trap to the logger to start the processor trace (PT). This takes 3020 cycles, with most of the time spent handling the `#UD2` trap (~2400 cycles). Afterwards, another `#UD2` round trip is executed to notify the APPARE-LOGGER to disable PT. The sequence of steps is the same as enabling PT; hence, this roughly takes the same amount of time (3075 cycles). Note that while toggling processor trace (PT) is relatively expensive, under well-generated reference behaviors, these operations happen infrequently (§8.2). Hence, their impact is amortized.

Memory consumption breakdown. The logger consumes memory to (a) maintain restricted kernel code pages and track anomalous control-flow (§5.2.1), (b) set up multiple EPTs (Fig. 8a), and (c) store processor trace (PT) packets.

APPARE-LOGGER requires 9.6MiB of memory to keep restricted code pages. Moreover, we reserved 4GiB of memory for multiple EPT tables. However, during runtime experiments, we observed that the actual amount of the required memory is less than that. In the future, to cut down EPT memory budgets, APPARE could reuse the same EPT entries for multiple versions of EPTs (instead of performing deep copies). Additionally, for the PT buffers, APPARE reserves per-core dual-buffer TOPA regions of 64MiB.

TABLE 5: Selected real-world programs and their workloads.

Application	Workload Description
Nginx	Default 4 worker threads; tested with the `ApacheBench` of 10K requests for a 10KB file and 12 client concurrency.
Redis	Default 16 databases; tested by `Redis-benchmark` of 1M sets and 1M gets for 32 bytes data with 1000 clients.
Memcached	Default settings; tested by `Memaslap` of 1M set and 1M get for 32 bytes data with 50 clients.
7zip	Phoronix benchmark: `pts/compress-7zip`.
OpenSSL	Phoronix benchmark: `pts/openssl`.

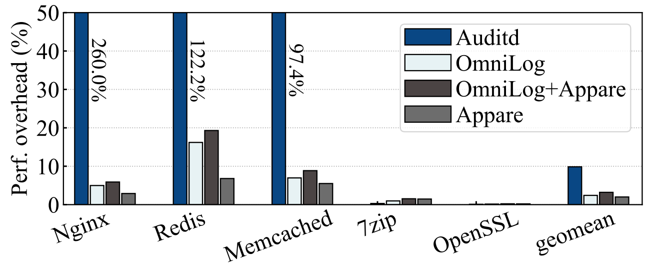


Figure 15: APPARE runtime overhead compared with existing audit systems. The number of syscalls per-sec (from left to right): 63247, 35323, 137006, 743, and 99. The percentage of control-flow violations (#`UD2`) among all syscalls from left to right: 2.1%, 5.6%, 8.7%, 5.3%, and 6.6%.

8.2. Real-World Programs

Settings. We chose five real-world applications to generate profiles: web server (*Nginx*), key-value stores (*Redis, Memcached*), compression software (*7zip*), and cryptographic computation (*OpenSSL*). These applications have also been evaluated by prior auditing research [25, 26] for enterprise computers. Tab. 5 lists the real-world programs and their well-known workloads. We ran each experiment 5 times and present average results for each application.

Results. Fig. 15 presents the observed runtime overhead and statistics. The geometric mean overhead of **APPARE** across all programs is 2.0%. For high-performance programs such as Memcached, Redis, and Nginx, APPARE incurred higher overheads, but this still ranged only from 3.0% to 6.8%. This slightly increased overhead comes from the high system call event throughput of these programs, as well as runtime violations related to control flow. Nonetheless, given APPARE's augmented reasoning approach to derive reference behaviors (§5.1), control-flow violations (logging) happen infrequently. Thus, the overall overhead is modest.

Comparing APPARE's overhead with other system call-only audit systems, we observe that its overhead is comparable to the state-of-the-art (**OmniLog**), whose geometric mean overhead was 2.4%, while remaining significantly faster than the deployed **Auditd** (geomean was 9.6%). Combining APPARE and OmniLog gives us an audit system that tracks all system calls and control-flow traces, and such a system (**OmniLog+APPARE**) still only incurred a geometric mean overhead of 4.3%, with a worst-case of 19.3%. For reference, this is still significantly faster than the overheads incurred by Auditd (e.g., up to 260% for Nginx).

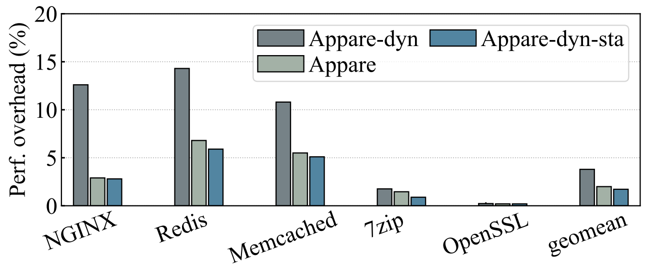


Figure 16: APPARE runtime overhead (by using variations shown in §7.2 to generate syscall reference behaviors).

Fig. 16 illustrates APPARE's performance overhead break down upon variations. Specifically, using pure dynamic profiles as reference behaviors (**APPARE-dyn**) is generally low, with syscall-intensive programs seeing the highest overheads (14.3%). This is due to the significant underapproximation of benign syscall behaviors using dynamic profiling. In contrast, **APPARE** and **APPARE-dyn-sta** impose similar, modest overheads—averaging 2.0% and 1.7%, respectively. These results show that APPARE balances performance with accurate approximation of syscall behaviors, while maintaining a tight reference-behavior scope (§7.2).

Takeaways. APPARE incurs modest performance overhead across real-world applications that is comparable to the state-of-the-art audit systems that are designed to log syscall information only, while logging richer information to help track sophisticated kernel exploits (§7). For control flow logging, APPARE's augmented hybrid approach (§5.1) also effectively balanced the approximation of syscall reference behaviors and runtime logging costs.

9. Storage Evaluation

This section describes APPARE's log entry sizes, and the storage required to maintain them over system call-only logs.

Log entry sizes. We leveraged Auditd as the reference for syscall log storage (on average 330 bytes per log entry). Control flow traces are of variable-lengths, and generated by the hardware during indirect jumps and calls. On average, their length is less than a byte (per packet).

Storage requirement. We used three applications, Nginx, Redis, and Memcached, with the workloads listed in Tab. 5 to evaluate the storage requirement, and compared it with system call logs produced by Auditd. Note that we do not compare with OmniLog, because it uses an efficient compression algorithm, while both APPARE's current prototype and Auditd produce uncompressed logs. A basic block-level pre-summarization technique [14] can further optimize PT storage needs, which we leave as future work.

We found that APPARE's log storage requirement is comparable to Auditd. With selective tracing, control-flow traces do not incur high overhead (unlike prior work [14, 15]). For Nginx, system call logging from Auditd reaches a throughput of 11.6 MB/s, while APPARE produces control-flow traces at a rate of 4.1 MB/s. For Redis, syscall log throughput is 8.6 MB/s, and APPARE produces control-flow traces at a rate of 5.1 MB/s. Finally, for Memcached, syscall logs are generated at 42.0 MB/s, while APPARE produces control-flow traces at 26.0 MB/s.

We were also curious to see how selective anomalous control-flow tracing (§5.2) would compare to (a) full tracing using PT and (b) pure-dynamic profiling, i.e., **APPARE-dyn** (with under-approximation). Our results show that enabling PT for full syscalls results in extremely high throughput, with more than 133.6 MB/s (32.6×), 78.1 MB/s (15.3×), and 170.6 MB/s (6.6×) for Nginx, Redis, and Memcached, respectively. For **APPARE-dyn**, its tracing throughput ranges from 12.5 – 43.5 MB/s. This shows the importance of APPARE's selective tracing design, which significantly reduces the size of control-flow logs—by ∼50% and ∼91% compared against dynamic profiling and tracing-all, respectively.

10. Discussion and Future Work

Platform portability. APPARE's hardware primitives (mentioned in §5.2) are also available on platforms like ARM. In particular, APPARE's control flow tracing can be supported using Embedded Trace Macrocell/Extension. LBR-based historic context recording can be achieved by recent features like the Branch Record Buffer Extension. EPT for selective trace and memory protection can be supported by Stage-2 Page Table. Those features have been adopted by existing work, for both instruction-level tracing for debugging [66, 67], and virtualization-based memory protection [68].

Alternative monitor design choice. Our current design deploys APPARE-LOGGER alongside a trusted hypervisor, thus requiring virtualization extensions. Although such extensions are widely deployed [61], they may introduce non-trivial performance costs in specific circumstances (e.g., upto 30% overheads on memory-intensive workloads despite hardware optimizations [69–71]). An alternative design that avoids such costs would be to deploy an intra-kernel isolation-based security monitor [72–74]. These monitors leverage hardware features such as Intel PKS [75] to create a higher *virtual* privilege for a trusted component in kernel mode, effectively deprivileging the main kernel with generally low overheads. Deployed alongside such a monitor, APPARE can enable selective secure tracing (§5.2) by controlling the MMU interface and page tables to enforce per-syscall restricted views. It can also trap PT-related registers for selective tracing, while instrumenting MMIO/DMA configurations to protect in-memory tracing and persistence.

Alternative profiling strategies. APPARE employs representative workloads (§7.1) as the initial seed for constructing the reference set. Another choice is full system-wide profiling, where all running workloads, including programs and background daemons, are dynamically profiled during (offline) syscall execution to derive reference sets. While this approach may yield a more comprehensive set of executed functions, the current reliance on `ftrace` would incur substantial storage and parsing overhead (§6). We leave the exploration of efficient techniques for whole-system workload profiling (e.g., compiler instrumentation [19]) to future work.

Interrupt context handling. APPARE currently ignores interrupt contexts (§6), but this can be addressed by adopting the same technique used in §5.1, namely, profiling interrupt-related functions and building reference sets for each interrupt context. The hypervisor could then set up a per-interrupt code view by interposing the exception vector entries (IDT). However, due to the timing sensitivity of interrupts, our `ftrace`-based analyzer does not capture executed functions, leading us to omit them in the prototype. One solution could be to leverage a lightweight compiler (LLVM) instrumentation [19]. We leave the development of such mechanisms for interrupt contexts to future work.

Whole exploit lifecycle investigation. While APPARE took a stride toward auditing anomalous control flow during kernel exploits, a complete investigation requires reconstructing the entire exploit lifecycle—from attack preparation to privilege escalation. Real-world exploits often involve preparatory phases such as heap spraying, object reuse, and controlled overwrites that occur long before control-flow hijacking, sometimes spanning multiple system calls. Future work should extend auditing to capture these complicated preparatory steps, including memory layout manipulation and object misuse, to reveal how early data corruption evolves into arbitrary write attack primitives and final privilege escalation.

11. Related Work

Privileged system auditing. Virtualization-based monitors have been leveraged to improve auditing security. BackTracker [9] is the seminal work on leveraging a hypervisor to trap and monitor the important activities (e.g., file operations and syscalls) of a guest OS kernel. More recently, OmniLog [26] presented a generic system architecture to securely collect and persist system call logs, with one of its designs built on a virtualization monitor. There have also been designs that leverage other privileged system components like TrustZone [76–78] and the NestedKernel [79] to protect logs. APPARE's design is inspired by such designs. However, all prior systems deal with securing general (syscall-level) logs from an untrusted kernel, and do not improve the logging granularity to track sophisticated kernel exploits.

Kernel control-flow specialization. APPARE's reference behavior-based control-flow tracing is inspired by runtime application-driven kernel debloating systems [18–20, 80]. In this regard, APPARE is most related to Shard [19], which also leverages restricted code pages to determine when kernel execution is violated. However, these systems have an orthogonal goal: to prevent kernel exploits by reducing the available codebase. This requires them to enable expensive mechanisms like control-flow integrity (CFI) [81] on profile violations, unlike APPARE's lightweight secure hardware tracing. While logging of a violation was explored by FACE-CHANGE [20], it only logged the initial function (which can be circumvented by strong attackers) but not the entire anomaly control-flow path like APPARE.

Hardware control-flow tracing. PT has been widely applied for program diagnoses, such as root cause analysis [53, 82–85], bug reproduction and hunting [86–89]. Gist [84] and Snorlax [85] leverage PT to track thread interleavings to identify race conditions and concurrency bugs. POMP [83] and REPT [82] perform reverse debugging of program failures by tracking information flow on PT. ARCUS [53] further aids in root cause analysis through symbolic execution. Meanwhile, PT-assisted CFI enforcement is another research area. While FlowGuard [90] and PT-CFI [91] leverage PT to accelerate CFI violation detection, Griffin [62] and μCFI [92] adopt it to enforce finer-grained CFI for user programs.

12. Conclusion

APPARE is a fine-grained auditing framework for investigating sophisticated kernel exploits. Using systematic (a) augmented hybrid syscall reference behavior identification and (b) virtualized selective and secure tracing, APPARE traces anomalous control-flow behaviors within the kernel. Our evaluation on real-world workloads and vulnerabilities shows that APPARE can effectively log the in-memory attack control-flow behavior during exploits with low overheads.

Acknowledgment

We thank our shepherd and reviewers for their constructive feedback on improving the paper. We also thank Rahul Priolkar and Kaijie Zhu for their help with experiments and discussion. This research is supported by the National Research Foundation, Singapore, the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>), and the US Air Force Office of Scientific Research (AFOSR) under award number FA9550-24-1-0204. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Research Foundation, Singapore, the Cyber Security Agency of Singapore, or AFOSR.

References

[1] Kernel self-protection. https://www.kernel.org/doc/html/v5.4/security/self-protection.html?highlight=kaslr#kernel-address-space-layout-randomization-kaslr. Online; Accessed 10 January 2025.

[2] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475, 2022. Online; Accessed 10 January 2025.

[3] Cook: Security things in Linux v5.3. https://lwn.net/Articles/804849/.

[4] Elena Reshetova. x86/entry/64: randomize kernel stack offset upon syscall. https://lwn.net/Articles/785484, 2019. Online; Accessed 10 January 2025.

[5] Sami Tolvanen. Kcfi support. https://lwn.net/Articles/893164, 2022. Online; Accessed 10 January 2025.

[6] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Retspill: Igniting user-controlled data to burn away linux kernel protections. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[7] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[8] RetSpill: KCFI_eval. https://github.com/sefcom/RetSpill/tree/main/experiments/kcfi_eval.

[9] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[10] R. Sekar, H. Kimm, and R. Aich. eaudit: A fast, scalable and deployable audit data collection system. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[11] SUSE. Understanding Linux Audit. https://documentation.suse.com/sles/12-SP4/html/SLES-all/cha-audit-comp.html.

[12] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[13] 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 *USENIX Security Symposium (USENIX)*, 2018.

[14] Jun Zeng, Chuqi Zhang, and Zhenkai Liang. Palantír: Optimizing attack provenance with hardware-enhanced system observability. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[15] Carter Yagemann, Mohammad A. Noureddine, Wajih Ul Hassan, Simon Chung, Adam Bates, and Wenke Lee. Validating the integrity of audit logs against execution repartitioning attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[16] Cve-2021-22555 details. https://nvd.nist.gov/vuln/detail/CVE-2021-22555.

[17] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[18] Hsuan-Chi Kuo, Akshith Gunasekaran, Yeongjin Jang, Sibin Mohan, Rakesh B Bobba, David Lie, and Jesse Walker. Multik: A framework for orchestrating multiple specialized kernels. *arXiv preprint arXiv:1903.06889*, 2019.

[19] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Xu Dongyan. Shard: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *Proceedings of the 30th USENIX Security Symposium (Security)*, 2021.

[20] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014.

[21] Microsoft Learn. Event tracing for Windows (ETW), 2021. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-.

[22] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *27th Annual Network and Distributed System Security Symposium, NDSS*, 2020.

[23] 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.

[24] 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)*, 2022.

[25] Adil Ahmad, Sangho Lee, and Marcus Peinado. HardLog: Practical tamper-proof system auditing using a novel audit device. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[26] Varun Gandhi, Sarbartha Banerjee, Aniket Agrawal, Adil Ahmad, Sangho Lee, and Marcus Peinado. Rethinking System Audit Architectures for High Event Coverage and Synchronous Log Availability. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, 2023.

[27] Chuqi Zhang, Jun Zeng, Yiming Zhang, Adil Ahmad, Fengwei Zhang, Hai Jin, and Zhenkai Liang. The hitchhiker's guide to high-assurance system observability protection with efficient permission switches. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[28] Performance Anomaly Detection with Intel Processor Trace and Intel VTune Profiler. https://www.intel.com/content/dam/develop/external/us/en/documents/session1-talk3-844182.pdf.

[29] Tenable Cyber Exposure Study - Application Software Security: Risk Prioritization. https://docs.tenable.com/cyber-exposure-studies/application-software-security/Content/RiskPrioritization.htm.

[30] Guide to auditing web servers. https://download.manageengine.com/products/eventlog/web-server-auditing-guide-download.pdf.

[31] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[32] M. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. Ul Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, 2023.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[34] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[35] ftrace - Function Tracer — The Linux Kernel documentation. https://www.kernel.org/doc/html/v5.0/trace/ftrace.html.

[36] Xinyang Ge, Ben Niu, and Weidong Cui. Reverse debugging of kernel failures in deployed systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[37] Peng Jiang, Ruizhe Huang, Ding Li, Yao Guo, Xiangqun Chen, Jianhai Luan, Yuxin Ren, and Xinwei Hu. Auditing frameworks need resource isolation: A systematic study on the super producer threat to system auditing and its mitigation. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[38] Kangjie Lu and Hong Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 2019.

[39] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. *Volume 3A: System Programming Guide*, 2016.

[40] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[41] Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Jana. Exploiting code symmetries for learning program semantics. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

[42] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025.

[43] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. Large language models for code analysis: Do LLMs really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[44] Baijun Cheng, Cen Zhang, Kailong Wang, Ling Shi, Yang Liu, Haoyu Wang, Yao Guo, Ding Li, and Xiangqun Chen. Semantic-enhanced indirect call analysis with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024.

[45] Yichen Li, Yun Peng, Yintong Huo, and Michael R. Lyu. Enhancing llm-based coding tools through native integration of ide-derived static context. Association for Computing Machinery, 2024.

[46] Guoren Li, Manu Sridharan, and Zhiyun Qian. Redefining Indirect Call Analysis with KallGraph . In *IEEE Symposium on Security and Privacy (SP)*, 2025.

[47] Tianrou Xia, Hong Hu, and Dinghao Wu. DEEPTYPE: Refining indirect call targets with strong multi-layer type analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[48] Linux Virtualization-Based Security (LVBS). https://lpc.events/event/17/contributions/1515/attachments/1353/2717/LPC_2023_LVBS.pdf.

[49] Intel Jason Chen. Supporting TEE on x86 Client Platforms with pKVM. https://www.youtube.com/watch?v=EP9ps_h-WeI.

[50] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[51] Kenichi Yasukata, Hajime Tazaki, and Pierre-Louis Aublin. Exit-less, isolated, and shared access for virtual machines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.

[52] System V ABI. https://wiki.osdev.org/System_V_ABI.

[53] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. ARCUS: Symbolic root cause analysis of exploits in production systems. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[54] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *2012 IEEE Symposium on Security and Privacy (S&P))*.

[55] Powerful disassembler library for x86/amd64. https://github.com/gdabah/distorm, 2021. Online; Accessed 6 April 2022.

[56] Ollama: Get up and running with large language models. https://ollama.com/.

[57] Qwen3: Think Deeper, Act Faster. https://qwenlm.github.io/blog/qwen3/.

[58] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2020.

[59] Adil Ahmad, Alex Schultz, Byoungyoung Lee, and Pedro Fonseca. An Extensible Orchestration and Protection Framework for Confidential Cloud Computing. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Jul 2023.

[60] Bareflank/hypervisor. https://github.com/Bareflank/hypervisor.

[61] Microsoft Learn. Virtualization-based security (VBS), 2020. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.

[62] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[63] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V.N. 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*, 2019.

[64] Linux test project. https://github.com/linux-test-project/ltp, 2024. Online; Accessed 9 January 2025.

[65] Bonan Ruan, Jiahao Liu, Chuqi Zhang, and Zhenkai Liang. Kernjc: Automated vulnerable environment generation for linux kernel vulnerabilities. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, 2024.

[66] Zhenyu Ning and Fengwei Zhang. Ninja: Towards transparent tracing and debugging on ARM. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[67] Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Yajin Zhou, Chao Zhang, and Haipeng Cai. Ncscope: hardware-assisted analyzer for native code in android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[68] Alexander Van't Hof and Jason Nieh. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022.

[69] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[70] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[71] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized numa servers with vmitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[72] Chuqi Zhang, Rahul Priolkar, Yuancheng Jiang, Yuan Xiao, Mona Vij, Zhenkai Liang, and Adil Ahmad. Erebor: A drop-in sandbox solution for private data processing in untrusted confidential virtual machines. In *Proceedings of the Twentieth European Conference on Computer Systems*, 2025.

[73] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. Dope: Domain protection enforcement with pks. In *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023.

[74] Yinggang Guo, Zicheng Wang, Weiheng Bai, Qingkai Zeng, and Kangjie Lu. BULKHEAD: secure, scalable, and efficient kernel compartmentalization with PKS. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[75] Memory protection keys for the kernel. https://lwn.net/Articles/826554.

[76] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, 48, 2015.

[77] Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. Trail of bytes: efficient support for forensic analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[78] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the Arm TrustZone secure world. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.

[79] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[80] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. Kasr: A reliable and practical approach to attack surface reduction of commodity os kernels. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer, 2018.

[81] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[82] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. Rept: Reverse debugging of failures in deployed software. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[83] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Pomp: postmortem program analysis with hardware-enhanced post-crash artifacts. In *USENIX Security Symposium (USENIX)*, 2017.

[84] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[85] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy diagnosis of in-production concurrency bugs. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[86] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[87] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. Automated bug hunting with data-driven symbolic root cause analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[88] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[89] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[90] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient cfi enforcement with intel processor trace. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2017.

[91] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *ACM Conference on Data and Applications Security (CODASPY)*, 2017.

[92] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

The paper introduces APPARE, an auditing framework for tracing anomalous and unlikely control-flow during a system call. Traditional audit logs are typically limited to coarse-grained system call events and thus fail to provide the visibility needed for investigating advanced kernel control-flow hijacking. APPARE tackles this challenge by profiling benign syscall reference behavior and applying semantic reasoning to distinguish anomalous in-kernel control-flow. APPARE uses virtualization-based, tamper-resistant tracing with hardware extensions, ensuring both efficiency and security in capturing detailed execution traces. The authors have built a full prototype and performed a comprehensive evaluation to show that APPARE can effectively log the in-memory attack control-flow behavior. Importantly, the framework maintains low performance overhead across diverse workloads. Overall, APPARE represents a practical step forward in enhancing forensic analysis capabilities for modern kernel security.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) APPARE provides strong tamper-resistance through virtualization extensions and selective logging mechanisms ensures secure and precise tracing.
2) Evaluation demonstrates success on representative workloads and 10 real-world CVEs, with clear performance comparisons against state-of-the-art systems.
3) APPARE maintains low overhead (2.0% slowdown), while providing fine-grained forensic visibility, making it both deployable and valuable in real-world settings.