Validating the Integrity of Audit Logs Against Execution Repartitioning Attacks

Carter Yagemann
Georgia Institute of Technology
Atlanta, Georgia, USA

Mohammad A. Noureddine
Rose-Hulman Institute of Technology
Terre Haute, Indiana, USA

Wajih Ul Hassan
University of Illinois
Urbana-Champaign, Illinois, USA

Simon Chung
Georgia Institute of Technology
Atlanta, Georgia, USA

Adam Bates
University of Illinois
Urbana-Champaign, Illinois, USA

Wenke Lee
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

Provenance-based causal analysis of audit logs has proven to be an invaluable method of investigating system intrusions. However, it also suffers from dependency explosion, whereby long-running processes accumulate many dependencies that are hard to unravel. Execution unit partitioning addresses this by segmenting dependencies into units of work, such as isolating the events that processed a single HTTP request. Unfortunately, we discover that current designs have a semantic gap problem due to how system calls and application log messages are used to infer complex internal program states. We demonstrate how attackers can modify existing code exploits to control event partitioning, breaking links in the attack and framing innocent users. We also show how our techniques circumvent existing program and log integrity defenses.

We then propose a new design for execution unit partitioning that leverages additional runtime data to yield verified partitions that resist manipulation. Our design overcomes the technical challenges of minimizing additional overhead while accurately connecting low level code instructions to high level audit events, in part with the use of commodity hardware processor tracing. We implement a prototype of our design for Linux, MARSARA, and extensively evaluate it on 14 real-world programs, targeted with expertly crafted exploits. MARSARA's verified partitions successfully capture all the attack provenances while only reintroducing 2.82% of false dependencies, in the worst case, with an average overhead of 8.7%. Using a new metric called Partitioning Attack Surface, we show that MARSARA eliminates 47,642 more repartitioning gadgets per program than integrity defenses like CFI, demonstrating our prototype's effectiveness and the novelty of the attacks it prevents.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Software and application security**.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484551

KEYWORDS

auditing, execution unit partitioning, processor tracing

ACM Reference Format:
Carter Yagemann, Mohammad A. Noureddine, Wajih Ul Hassan, Simon Chung, Adam Bates, and Wenke Lee. 2021. Validating the Integrity of Audit Logs Against Execution Repartitioning Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3484551

1 INTRODUCTION

The complexity of interactions within modern computers makes it difficult to detect, prevent, and reverse unwanted system changes, such as in the case of an intrusion. A promising method of understanding suspicious events is causal analysis, in which system audit logs are transformed into a *data provenance graph* that encodes causal dependencies and historical relationships between subjects (processes) and objects (files, sockets, etc.) [5, 29, 33, 38, 42, 47, 52]. The resulting provenance graph can then be used by human analysts or monitoring tools for intrusion detection [3, 14, 68], forensic investigation [4, 34, 42, 52, 56, 57, 60, 85], and more [2, 19–21, 34, 95].

However, due to the noisey and complex nature of system interactions, provenance graphs are not always sufficient for investigating suspicious activity. Specifically, long-running processes can accumulate causal dependencies over time that become increasingly difficult to unravel; referred to as the *dependency explosion problem* [79] (a.k.a. false provenance). For example, consider a web server handling many requests in parallel. Due to the interwoven system calls invoked by multiple threads, data provenance will falsely conclude that all the files read during a request are causally related to all the currently connected remote IP addresses, which is excessive. However, multi-threading is not the only source of false provenance. Even in a single-threaded web server, a request response will link back to all previously handled requests, even though no actual data flow between the most recent request and prior responses occurred.

To address dependency explosion, the research community has proposed *execution unit partitioning* (EUP) [36, 48–50, 56–58]. In EUP, audit log events are grouped at the sub-process level, subdividing a monolithic long-running process into autonomous units of work that are easier to trace in the graph. *Signatures* for identifying where to place partitions are typically generated during an offline profiling phase and may be encoded in several ways, such as a state machine of regular expressions to be matched against the audit

log [49]. Continuing the web server example, a unit would be the code that processes a single request-reponse pair and the signature is the sequence of system calls and/or application level logs that the code emits. For example, the code might be expected to open a socket and record an access log entry with the source IP address, time, and requested URL at the start of its handling routine. Once a system call closes the socket, this marks the end of that unit. In this way, the data provenance system can distinguish between requests, correctly identifying which objects were accessed or modified on their behalf. In short, EUP is what makes data provenance viable for auditing real-world production systems.

However, all existing EUP solutions [36, 48–50, 56–58] make a dangerous implicit assumption, which we are the first to point out. Namely, they assume that *if the audit log events match the expected signatures, the underlying application must be performing the expected execution.* Ensuring this in real-world settings requires complete user program integrity, otherwise a low level bug (e.g., overflow, use-after-free) giving rise to *emergent execution* [9, 18, 75] or *out-of-bounds writes* [40] can produce erroneous signature matches. This in turn can add and remove partitions, reintroducing false dependencies and severing legitimate ones. Potentially, this would make it possible for the attacker to hide their steps from investigators while also framing innocent parties.

Would real-world adversaries be motivated to perform such an attack on EUP-enabled systems? Unsurprisingly, attackers already tamper with audit logs to cover their tracks [31, 43, 65, 72, 80]. Tampering is so prevalent that 72% of incident responders have encountered it during real investigations [15, 23], to which numerous *log integrity* defenses have been proposed [7, 24, 32, 37, 41, 44, 45, 55, 62, 66, 67, 71, 73, 74, 91, 92]. However, to our knowledge, all past solutions focus solely on an *offline* threat model, with tampering occurring *after* events are written to the log and are resting on a storage device. This is a distinctly different threat to what we just described, where changes to the user application's *online* execution yields frustratingly incorrect analysis results.

In this work, we are the first to present two avenues for *online* tampering designed to frustrate provenance analysis without violating traditional notions of log integrity. At a high level, the first technique, *spoofing*, attempts to inject fake log events into the runtime by either maliciously invoking event-emitting code or by tampering with write buffers via an arbitrary write primitive (e.g., format string vulnerability). The second technique, *delaying*, introduces memory corruptions with deferred repercussions, allowing the current unit to finish normally, whereas a subsequent unit (with no discernible causal relationship to the prior) resumes the attack. To demonstrate practicality, we show how to create working examples starting from real-world CVE vulnerabilities.

In response to this new threat, the obvious solution would *seem* to be the deployment of known control flow integrity (CFI) techniques. However, we surprisingly discover that CFI can only prevent a subset of EUP-targeted attacks, specifically those built on control hijacking. Even then, depending on how subtle the hijack is (e.g., overwriting a code pointer to an arbitrary address versus another valid function), the overhead of enforcing sufficiently fine-grained CFI can be upwards of 47% [39]. Conversely, when data-only exploits are leveraged, prevention exceeds CFI's scope [40].

Seeking a different solution, we propose a new defense to *validate* the placement of partitions. Specifically, given knowledge about the kinds of events certain parts of the code should yield (data flow), and their expected orderings (control flow), our solution compares runtime execution traces to audit logs to ensure consistency. If the attacker tries to change the ordering with control flow bending, or inject fake event data from another part of the program, our defense will detect the discrepancy, disregarding the resulting events during partitioning to preserve the integrity of the provenance graph.

However, designing a solution around this idea raises several technical challenges. First, our system has to accurately determine which event sequence to expect for a given execution. Fortunately, rather than having to consider all possible executions, our system can focus on just the ones used offline to generate EUP signatures. Any program paths outside this scope were not intended by the EUP algorithm to yield partitions in the first place. To accomplish this, we propose a binary analysis that combines concrete execution traces with symbolic analysis.

Next, our solution has to collect the necessary additional runtime information to perform validation while minimizing additional overhead compared to prior (insecure) work. To this end, we propose a design that is compatible with the hardware processor tracing (PT) available in commodity processors[1], which a kernel driver can securely control. We then overcome the challenge of connecting low level instruction sequences collected with PT to high level audit log events to accurately perform validation.

To evaluate our design, we implement a prototype for Linux, MARSARA[2], and extensively evaluate it on 14 real-world programs using expertly crafted exploits. MARSARA accurately partitions all the attack provenances while only reintroducing 2.82% of false dependencies, in the *worst case*, with an average performance overhead of 8.7% over traditional auditing frameworks. We also create a new metric for measuring the vulnerability of user programs to EUP attacks, **P**artitioning **A**ttack **S**urface (PAS), and show that MARSARA removes 47,642 more gadgets than CFI on our real-world programs, on average per program. To promote further exploration of solutions to the new *online* log integrity problem, we have open sourced our code and data.[3]

2 BACKGROUND & MOTIVATION

Consider an Nginx web server with several worker processes, hosting a music website that the attacker aims to steal from. He starts by triggering CVE-2013-2028 using a maliciously crafted HTTP request, originating from the IP address `x.x.x.x` in Figure 1. This causes a buffer overflow within one of the worker processes, allowing him to inject shellcode and corrupt a code pointer. However, instead of corrupting any code pointer arbitrarily to point at the shellcode, he cleverly overwrites a particular event handler[4] that he knows the worker will not use to complete his request. Consequently, his HTTP request completes with no anomalous system calls or application messages. We call this novel setup a *delay attack*, which we elaborate on in Section 3.

[1] Available in Intel®, AMD®, and ARM® processors.
[2] Monitor Application Runtimes, Stop Arbitrary Repartitioning Attacks.
[3] https://github.com/carter-yagemann/MARSARA
[4] ngx_http_process_request_line

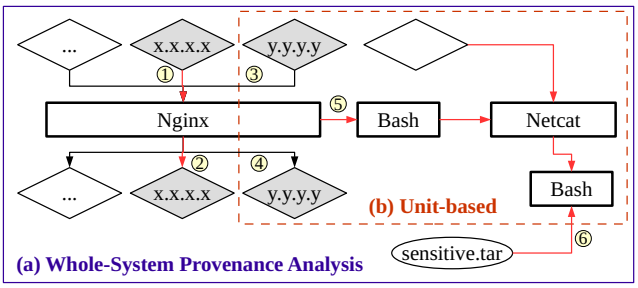

Figure 1: Motivating example. The attacker sends a request ① that produces a seemingly normal response ②. However, it has actually employed a delay to trigger the payload ⑤ during a benign request ③ to exfiltrate a sensitive file ⑥, which is further obfuscated using spoofed log messages.

Later, a request from a benign IP, `y.y.y.y`, is received, causing the worker to access the corrupted code pointer and execute the shellcode. It starts by reading sensitive local files into a buffer. However, instead of immediately transmitting the data back to the attacker's server, it first writes several forged log entries into Nginx's access and debug logs to make it look like the current request has ended. This is another novel attack technique, which we coin *spoofing* and also elaborate on in Section 3. With the spoofed messages inserted, the shellcode transmits the buffer of sensitive data back to the attacker and then the worker resumes normal operation.

2.1 Existing Defenses & Limitations

Intrusion Detection & Prevention. Several aspects of the motivating attack make it difficult to detect or prevent at the onset. First, the initial exploit does not emit any anomalous system calls or application-layer events, rendering host-based defenses reliant on them ineffective. Obfuscation makes it impractical to detect the payloads on the network, and the shellcode may no longer be in memory by the time a symptom of the attack is observed. The corrupted code pointer requires fine-grained CFI to detect because its legitimate value is calculated dynamically during runtime and the necessary instrumentation can yield upwards of 47% execution overhead [39].

Whole-System Provenance Analysis. Whole-system provenance tools [5, 29, 33, 38, 42, 47, 52] record system call level events to establish causal dependencies between objects and subjects, resulting in a provenance graph. Figure 1 (a) shows the provenance graph for our motivating attack scenario *without EUP*. While the attacker's IP address is contained in the provenance graph, we also see the false dependency problem described in Section 1, where every open socket is associated with the exfiltrated data, making it inconclusive which connection instigated the attack and which request delivered the exploit and payload. At the same time, every file Nginx touched since its startup (e.g., configurations, temporary files) is also linked to the attack, making it inconclusive what was exfiltrated. In short, human analysts and automated systems do not have a clear picture for answering their forensic questions.

Unit-based Provenance Analysis. EUP [36, 49, 51, 56, 58, 59] attempts to solve this dependency explosion problem by partitioning the execution of a long-running process into autonomous *execution units* in order to provide more precise causal dependency graphs. While EUP is very useful when the adversary is oblivious to how it works, the delay and the spoofing attacks in our motivating example exploit it to further obfuscate what occurred.

Figure 1(b) shows the result. The delay attack successfully partitions away the request from the attacker (`x.x.x.x`), causing `y.y.y.y` to appear as the origin point of the attack. Additionally, the spoofing employed by the shellcode causes the reading of sensitive files to be partitioned separately from its transmission, obfuscating what was actually exfiltrated.

It may be tempting to argue that if the corrupted worker could be identified, then all these problems would be solved, however this is not the case. Since Nginx reuses workers across requests, simply following its PID will wrongly associate unrelated events from prior and future requests, reintroducing false dependencies.

2.2 Insights & Lessons Learned

From the above discussion of the motivating example, we observe that data provenance systems that only analyze traditional audit log events will never be able to verify that the recorded, seemingly normal, patterns were emitted by normal program execution, and not by delay or spoofing attacks. Conversely, systems like CFI that rely purely on low level control flow will never be able to answer forensic questions that consider the data contents of reads and writes. Furthermore, we demonstrate in Subsection 3.3 that data-only attacks can also leverage delays and spoofing, which is outside CFI's scope to handle.

Instead, our solution is to leverage execution tracing and knowledge gathered during the offline profiling for EUP to recognize the manipulative events introduced by the attacker. In this example, knowing that the worker processing requests executed a program path (due to the delay attack) that was never seen during profiling indicates that it should not be isolated into its own partition. Subsequently, recognizing that several log messages originated from a previously unknown code location (the shellcode), indicates that they should not be considered during partitioning, preventing the attack from separating the sensitive file reads from network sends.

In Section 3, we elaborate on how these novel delay and spoofing techniques can empower existing exploits to hinder provenance analysis. In Section 4, we formalize the threat model based on our attack techniques and then our proposed defense is presented in Section 5.

3 EXECUTION REPARTITIONING ATTACKS

We propose a novel set of techniques for augmenting existing exploits to hinder defenses and forensic tools reliant on data provenance. Our techniques enable exploits to achieve their original goal while simultaneously obfuscating the true sequence of attack events from defenders, making it harder to determine where the attack originated from and what was done to the victim system. The techniques can be divided into two categories, *spoofing* and *delays*, which manipulate the audit events emitted from the target application prior to them being recorded by the auditing framework. Consequently,

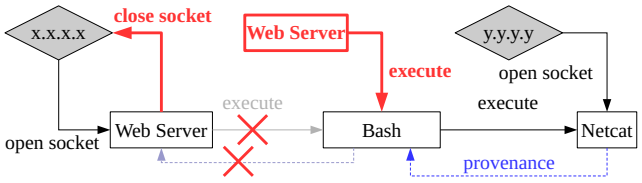

Figure 2: High level example of augmenting an exploit with spoofing to thwart data provenance. By adding a close socket system call, the call to execute Bash is partitioned into a different unit, isolating it from the attacker's exploit.

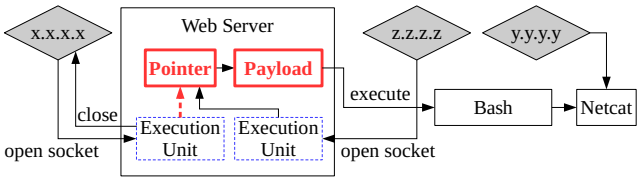

Figure 3: High level example of augmenting an exploit with delaying to thwart data provenance. By corrupting a code pointer, rather than directly executing the payload, a different unit can be exploited into triggering the next stage.

these techniques cannot be detected with traditional log integrity defenses [7, 24, 32, 37, 41, 44, 45, 55, 62, 66, 67, 71, 73, 74, 91, 92], which only detect changes after the logs are committed to storage.

3.1 Spoofing Attacks

Spoofing entails generating artificial system calls and application log messages in order to forge the necessary audit log events to satisfy an EUP signature. Typically, the attacker's exploit begins in the middle of an execution unit, with events linking the unit back to an ingress point. Figure 2 shows this for a web server example, with an open socket system call linking the current unit to the attacker's IP address.

Suppose the payload for the exploit is designed to start a reverse shell connected to a remote machine controlled by the attacker, thereby granting them access into the system. If the payload were triggered immediately, data provenance would trivially associate the resulting execute and open socket system calls to the current execution unit. Consequently, a system or human analyst wanting to investigate any of these events can recover the entire sequence using data provenance. For example, if the Netcat process is examined, a backward provenance query will reveal the attacker's IP address and the request used to compromise the web server. Similarly, a forward query will reveal the remote server used to issue commands and any data it exfiltrated.

What would happen if the payload closed the initial socket *before* invoking the execute system call? As it turns out, most existing EUP algorithms for data provenance will mark this as the end of the current execution unit and partition all subsequent audit log events into a new unit, as reflected in Figure 2.[5] With the call to execute Bash now in a new unit, the previously described data provenance query will not include the attacker's IP address, nor contain the request carrying the exploit and payload. In summary, with just one added system call, the attacker has thwarted the ability for data provenance to recover the full attack sequence.

While spoofing is conceptually straightforward, signatures can require many events, all of which have to be spoofed in the correct order to successfully match a signature. Continuing the previous example, for a real server like Nginx, simply closing a socket is not sufficient. There are also dozens of debug messages that have to be spoofed to create a valid signature. In Section 6, we evaluate an exploit that uses CVE-2009-4769 to target `httpd`'s `tolog` method to conduct a successful attack.

Format string bugs warrant special mention, as they are particularly powerful for spoofing. For example, CVE-2012-0809 in `sudo` can be exploited to yield any string starting with the prefix "sudo: ", making it very flexible for matching signatures. Interpreters that allow scripts to specify format strings (PHP: CVE-2015-8617, CVE-2016-4071) are also ripe for abuse in this manner.

3.2 Delay Attacks

Rather than forging fake events to create a partition, the attacker can alternatively augment their exploit to intentionally delay the manifestation of certain actions to later execution units, covertly spanning partitions in a way that will not be reflected in the data provenance. Figure 3 visualizes this at a high level, reusing the web server as an example. Rather than directly executing the payload, which would causally link the attacker's IP address and request to the resulting reverse shell, the exploit instead corrupts a code pointer to point to the payload and then exits normally. When a subsequent (benign) request causes the corrupted pointer to be dereferenced, it will inadvertently trigger the next stage of the attack with no audit log events linking it back to the attacker's request. This not only decouples the attacker from the payload, but also frames a benign IP address as being the ingress point.

However, delays do not always require a memory safety violation. For example, event handling loops in many programs can encounter situations where a task must be deferred and rescheduled for handling at a later time (e.g., because a necessary resource is not yet available). Offline analysis can miss these alternate code paths during profiling, creating unintended delay attack primitives.

3.3 Crafting Real-World Exploits

Based on our techniques of spoofing and delaying, we present 3 working exploits against real-world programs to encompass the techniques an adversary can use to exploit repartitioning attacks. Our exploits are based on known CVEs, extended using our attack techniques to invoke erroneous data provenance results.

CVE-2013-2028. This CVE stems from a bug in Nginx's handling of chunked HTTP requests and can be exploited to cause an out-of-bounds write. We use this to target Nginx with the *delay* technique. Specifically, we exploit the original stack overflow to change two local variables that are then used by the buggy function to perform a write, creating an arbitrary write primitive. We exploit this in turn

[5] The only exception we know of is BEEP [50] because it instruments programs with an explicit "end-of-unit" event, however this can also be spoofed to perform the attack.

to corrupt one of the program's global code pointers, implementing the delay primitive. To simplify the payload, we make the program's heap executable prior to the attack so that the malicious HTTP request can carry its own shellcode. In a real-world setting, the attacker could instead trigger the CVE multiple times to write a ROP chain into memory that corrupts the global pointer.

CVE-2004-0541. This CVE stems from a bug in one of Squid's remote authentication modules, which can be remotely triggered to cause a buffer overflow. Our attack augments exploits for this CVE with the *spoofing* technique. Specifically, we trigger the overflow in its NTLM authentication child process to inject and trigger a ROP chain, which in turn messages the logging daemon via an IPC channel to print arbitrary log strings. We use this spoof primitive to forge the necessary messages to complete a valid EUP signature, ending the current unit and starting a new one, and then trigger the payload, which is now causally disconnected from the attacker.

CVE-2009-4769. This CVE stems from multiple format string bugs in `httpd`, which can be triggered remotely by a HTTP request to perform arbitrary reads and writes. Specifically, the buggy logging procedure is intended to record details pertaining to the incoming HTTP request (timestamp, IP address, requested file, response code). However, by exploiting it with the spoof technique, an attacker can control the write to inject multiple seemingly legitimate entries into the log, thereby partitioning the attack across several bogus execution units with no causal dependencies. The exploit can then trigger a payload using arbitrary writes or leak data back to the attacker without creating a link to the malicious request.

4 THREAT MODEL & ASSUMPTIONS

Defender. The defender's goal is to investigate an intrusion with the aid of a full-system data provenance framework. In order to handle complex real-world long-running programs, it relies on EUP, as is the norm [36, 48–50, 56–58]. Conversely, simple short-lived programs that do not incur dependency explosion can have all their events grouped into a single partition and do not require further consideration for this work. In accordance with prior work [36, 48–50, 56–58], partitioning signatures do not span multiple programs, so each can be analyzed independently. We assume kernel integrity and correct ordering of audit data, which are standard prerequisites in all full-system auditing [36, 48–50, 56–58]. We only consider EUP attacks and note that our proposed solution is compatible with existing approaches to offline tamper-evident logging [7, 24, 32, 37, 41, 44, 45, 55, 62, 66, 67, 71, 73, 74, 91, 92].

CFI has some capacity to coincidentally reduce the EUP attack surface by limiting the range of unexpected *control* behaviors a program can exhibit. To account for this, we define a metric for quantifying attack surface reduction in Subsection 4.1 and perform a comparison between CFI and our solution in the evaluation. Our findings show that our design offers more protection than CFI, *against EUP attacks*, across all 14 evaluated real-world programs, eliminating 47,642 additional delay and spoof gadgets per program.

Attacker. The attacker's primary goal is to take control of a target program in order to gain a foothold into the victim's system. For brevity, we will consider a production server environment where the attack surface is an internet accessible service, such as a HTTP server. Since the attacker expects the defenders to be using an auditing framework that allows for data provenance, he is motivated to augment the attack with the techniques described in Section 3 to make it as difficult as possible to uncover his activities.

The minimum prerequisite for the attacker to succeed is one vulnerability in the target program that enables control flow hijacking or arbitrary write, along with knowledge of the EUP algorithm being used and a copy of the target program so he can know the partitioning signatures in advance. However, to demonstrate the strength of our proposed defense, we will consider a significantly more powerful adversary who has a complete local copy of the victim system and access to an arbitrary read vulnerability in the target program, granting him complete knowledge of the remote program's state and the ability to refine his attack to work on the first try, guaranteed. By demonstrating that our defense is able to correctly recover the complete attack provenance of this powerful adversary, we also demonstrate the ability to handle weaker, more realistically constrained attackers.

4.1 Quantifying EUP Attack Surface

In order to quantify the surface for EUP attacks and facilitate objective comparisons between defenses, we propose a new metric called **P**artitioning **A**ttack **S**urface (PAS). The intuition behind PAS is to quantify how many audit-event-producing sites (e.g., system calls, application log writing procedures) are reachable from any point in the program based on the policy being enforced by integrity defenses. The more sites that are reachable from the current point in the execution, the more events an attacker can choose from to match a signature.

To measure PAS in real-world programs efficiently, given a graph model representing the enforced policy, we define audit-event-producing sites as nodes that invoke either a system call or write library function (e.g., `printf`). Thus, for each node n in policy N and node e in the set of audit-event-producing nodes E, PAS is defined as:

$$\frac{\sum_{n \in N, e \in E} r(n, e, \{E - e\})}{|N|} \tag{1}$$

where r is a function that returns 1 if e is reachable from n without going through any other node in E (i.e., $\{E - e\}$) and returns 0 otherwise. This check is relevant because going through another node in E produces a side-effect that the attacker does not desire. Ultimately, higher PAS values reflect a weaker defense that grants greater flexibility to the attacker.

5 DESIGN & IMPLEMENTATION

The high level idea of MARSARA is to use control flow data and knowledge of event-producing code locations (i.e., what messages or system call parameters they can produce) to validate unit signature matches. Figure 4 shows our proposed design, which similar to prior work in EUP [36, 49, 58] consists of an offline profiling phase, an online auditing phase, and a post-forensic analysis.

During offline profiling, MARSARA records and analyzes PT traces of the target program, using a binary symbolic analysis, to identify important control and data flows along with possible starting points for execution units (Subsection 5.2).

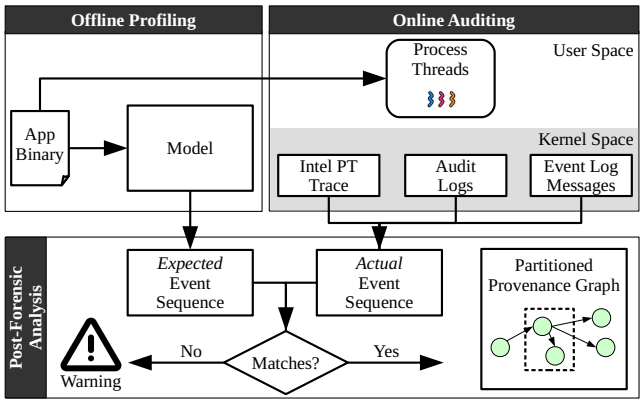

Figure 4: MARSARA architecture overview. An offline profiling phase yields a model of expected program behavior, which is used alongside execution traces and audit logs collected during online auditing to perform verified partitioning in post-forensic analysis.

During online auditing, MARSARA records the program's execution and stores it alongside the traditional audit log of system calls and application log messages (Subsection 5.3).

Lastly, **during post-forensic analysis**, MARSARA compares the recorded trace against the resulting audit log events to validate each occurring event (Subsection 5.4) and then uses these verified events to determine where to place partitions, yielding verified execution units (Subsection 5.5).

At first glance, this approach may seem too restrictive and false positive prone (i.e., rejecting of valid events) to be usable in real-world systems, however it works because:

(1) The cost of a false positive is low, merely reintroducing an unnecessary dependency back into the data provenance.
(2) Since all EUP work is based on offline profiling [36, 49, 58], no such system can guarantee that signatures are complete in the first place, and yet have demonstrated value in making data provenance usable for real-world systems [34].

Ultimately, MARSARA is effective if it preserves attack provenances while having a false dependency reduction and performance comparable to previous (insecure) systems.

In this work, we focus on demonstrating the ability for MARSARA to ensure integrity using verified events and execution unit signature matches, as opposed to proving that our EUP algorithm is the most accurate. Readers interested in the latter topic should refer to OmegaLog [36], which implements and evaluates a similar partitioning strategy (without integrity verification).

5.1 Intel Processor Trace

Before diving into the phases of MARSARA, it is important to understand how PT works, since we intentionally design our solution to be compatible with it for better performance. PT enables MARSARA to securely audit the basic blocks executed by user space programs and can be controlled with a kernel driver, which we implement as part of MARSARA. For brevity, we will focus on how Intel's implementation of PT works, which is the architecture supported by our prototype, however our design can be generalized to other PT implementations as well.

When a program for which MARSARA has a model is loaded for execution, it configures Intel PT to trace the execution. The MARSARA kernel maintains per-thread trace buffers, redirecting PT's data output appropriately during context switches. Anytime a branching or indirect control transfer instruction occurs, PT records an event packet with the outcome. For branches, the packet is a single taken-not-taken (TNT) bit, whereas for indirect transfers (indirect call, indirect jump, and return), the target instruction pointer (TIP) is recorded. The Intel PT hardware automatically applies compression to the written packets to conserve space.

At the start of execution, the MARSARA kernel driver takes a snapshot of the program's executable pages and then any additional pages loaded into memory afterwards (e.g., `mmap`) are also captured and recorded. This also includes dynamically generated code, such as just-in-time (JIT) compilation. The resulting *sideband* data consisting of the initial snapshot, subsequently mapped executable memory, and context switch events, are interwoven with the PT data in the thread buffers to yield a linear stream of data.

Each stream contains all the necessary data to recover the program's execution, down to individual instructions, with the help of a disassembler. However, as we will explain in Subsection 5.2, not every instruction needs to be recorded for auditing, so to conserve space we distill the instruction sequences using kernel worker threads into relevant events and metadata centered around basic blocks. Since the PT data is not needed until the post-forensic investigation phase, the workers process data asynchronously to minimize overhead.

Intel PT is only configurable in the root CPU privilege level using model specific registers (MSRs) and writes directly to physical memory. This allows the kernel to prevent all user space programs from reading or tampering with the trace. It also bypasses CPU caches, eliminating potential side channels and effects on the program's performance. When the trace buffer is almost full, a non-maskable interrupt (NMI) is raised, allowing the contents to be flushed without any data loss. As a result, systems leveraging Intel PT have demonstrated low performance overheads (under 7% [27, 39, 90]) and are capable of offering strong security integrity guarantees [27, 90].

5.2 Offline Profiling

In the offline phase, we propose to overcome the challenge of accurately determining a program's control and data flows by using a combination of concrete traces and symbolic analysis. Specifically, MARSARA reads a target binary and generates a model of the program consisting of the possible paths between application log events, systems calls, and function/loop heads. Formally, given a binary b, MARSARA generates a graph $G =< V, E >$ where V is a subset of b's basic blocks, and $E = V \times V$ is a set of edges such that $(u, v) \in E$ if there exists a path from u to v in b's control flow. System call and application log event nodes then get annotated with regular expressions defining their possible data values, calculated using binary single-path symbolic execution over the profiled traces. We use angr [76] for our Linux prototype.

Algorithm 1: Model generation in MARSARA

```
Func BuildModel
    Inputs  : Binary b
    Outputs : Model G
    F ← GetLoggingProcedures(b)
    V ← ⋃_{f∈F} GetCallSites(b, f)
    V ← V ∪ b.libc_calls ∪ b.function_heads ∪
      b.loop_heads ∪ b.function_returns
    foreach v ∈ V do
        v.rva ← CalculateRVA(b, v)
        if v is log call site then
            v.logstring ← GetLogFormatString(b, v)
        else if v is loop head then
            v.is_infinite_loop ← HasNoExitEdges(b, v)
    E ← {(u ∈ V, v ∈ V) | ∃ path u → v in b}
```

Algorithm 1 shows the steps to produce a model in more detail. First, MARSARA identifies the set $\mathcal{F}$ of logging procedures that produce application level messages. Then, using a first pass on the binary's CFG, derived from profiled execution traces, MARSARA captures the basic blocks that end in a call to any function in $\mathcal{F}$. Next, MARSARA collects all basic blocks that correspond to heads of functions/loops and blocks that lead to system calls. In practice, we find that applications rarely make direct system calls, relying instead on standard libraries (e.g., `libc`) that expose equivalent user APIs. To account for this, MARSARA also collects all calls to functions in `libc` and analyzes them to determine the possible system calls they can emit.

To accurately map these basic blocks to events received from PT and audit logs, MARSARA needs to collect further metadata about them. MARSARA first tags each node $v \in V$ with its corresponding type: log, system call, function head, loop head, standard library call. Then, MARSARA calculates the node's relative virtual address (v.`rva`), which corresponds to v's offset from the binary's base virtual address. RVAs allow MARSARA to recognize addresses reported by PT, which are absolute addresses affected by address space layout randomization (ASLR). For each node v that is a call site to a logging procedure, MARSARA uses symbolic execution to produce constraints that are then recorded as the log message's format specifier (v.`logstring`). This is essentially a regular expression of all messages this code location is expected to produce. Finally, to be able to identify execution units (Subsection 5.5) during the later post-forensic analysis phase, MARSARA marks all function and infinite loop heads. We consider such nodes to be possible candidates for starting new execution units since they often correspond to event-handling routines. While this is a heuristic, it has been well studied and considered reliable, appearing in many prior EUP systems [36, 49, 58].

5.3 Online Auditing

At runtime, MARSARA leverages PT to capture low level execution events alongside traditional audit logs of system calls and application level log messages. PT provides a hardware-enforced record of the program's control flow, application log messages reflect data flow, and system calls capture OS events. We pick these sources because they are generated by different layers of the environment (hardware, application, kernel) and are *correlated*. This provides MARSARA a rich perspective from which to verify consistency.

Hardware Processor Trace. Pure software solutions for recording runtime execution suffer from high performance overhead and weak security guarantees. PT is a hardware mechanism designed to address this by efficiently and securely capturing instructions as they are executed in the CPU. Intel's implementation has been included in their processors since 2015, making it a prevalent feature in most computing environments. Although we use Intel's implementation (Subsection 5.1) in MARSARA, our design generalizes to other PT hardware as well.

Application Layer Events. At runtime, audited programs are loaded with an instrumented standard library that augments the `write` call, as is typical of prior EUP designs [36]. In addition to writing to the original destination, the new call also forwards messages to the framework used to record system calls. Most standard auditing frameworks (e.g., `auditd`) provide an API with this functionality. To simplify the segmentation of messages during post-forensic analysis, the instrumented write also appends the process/thread IDs and current timestamp to the sent messages.

Although the event logging frameworks used by user space programs are diverse and heterogeneous, the vast majority rely on standard runtime libraries (e.g., `libc`) to efficiently write logs while preserving portability across systems. MARSARA takes advantage of this to capture log messages that indicate various states in the execution units. A more detailed discussion of supporting heterogeneous logging frameworks is presented in prior work [36] and we discuss our prototype's compatibility with other programming languages with alternative standard libraries in Section 7.

System Calls. Recording for system calls and their parameters are provided by the auditing frameworks MARSARA integrates with, which also include an API for MARSARA to forward application log messages into. For our prototype, we use Linux Audit.

5.4 Signature Match Validation

During the post-forensic analysis phase, MARSARA performs two tasks, starting with cross-validation of events received from PT with those from the audit logs, based on the model generated offline in Subsection 5.2. This yields *validated* audit events that will then be used to produce *verified* execution unit partitions, which we describe in Subsection 5.5.

Algorithm 2 formalizes our cross-validation matching. It takes three inputs: the generated model G, a PT trace $\mathcal{T}$, and an audit log $\mathcal{A}$ of system calls and application log messages. For each event e received from the PT trace, MARSARA first determines if it is a system call event or a code block event. If it is a system call, MARSARA extracts e's call number and checks that it matches the number on the next event received from the audit log. If the two numbers do not match, then the event is invalid and discarded.

Next, if the system call originates from a code block that is either in `libc` or the application's binary, MARSARA obtains the corresponding node in G that matches the event node's RVA. It then validates if the path observed so far matches at least one known

Algorithm 2: MARSARA's trace validation algorithm.

```
1  Func ValidateTraces
       Inputs  : Model G, PT Trace T, Audit Trace A, binary b
       Outputs : Validated Events Path P, Warnings W
2      W ← {Φ}, P ← {Φ}
       /* ω is the last matched node                    */
3      ω = Φ
4      foreach event e ∈ T do
5          if e is system call then
6              a ← GetNextEvent(A)
7              if e.syscall_num = a.syscall_num then
8                  P ← P ∪ {(e, a)}
9              else
10                 W ← W ∪ {(e, a, critical)}
11             if e.object ∈ {libc, b} then
12                 u ← GetNodeByRva(e.rva)
13                 ω ← ValidateEANode(e, u, a)
14         else
15             u ← GetNodeByRva(e.rva)
16             ω ← ValidateEANode(e, u, Φ)
17 Func ValidateEANode
       Inputs  : PT event e, nodes ω, u, Audit event a
       Outputs : Last matched node
18     match ← e is application log event ∧
         MatchLogString(a.logmessage, u.logstring)
19     if match ∨ (e is code block) then
20         if (ω, u) ∈ E then
21             P ← P ∪ {(e, a, u)}
22             return u
23         else
24             if ℓ(u) ∈ {function head} ∨ ℓ(ω) ∈ {function return}
                 then
25                 W ← W ∪ {(e, a, u, low)}
26                 P ← P ∪ {(e, a, u)}
27                 return u
28             else
29                 W ← W ∪ {(e, a, u, critical)}
30                 return Φ
31     else
32         W ← W ∪ {(e, a, u, critical)}
33         return Φ
```

signature. Non-system call PT events (i.e, loop heads, function heads, and returns) are treated in a similar manner.

To check for path validity, MARSARA keeps track of the last matched node in the current observed trace. If the newly matched node u is an application log node, MARSARA extracts the node's format specifier (u.logstring) from the model, and confirms that it matches the concrete message recorded in the audit log. If a discrepancy is found, the match is invalidated.

When the log matching succeeds, or alternatively, if u is simply a code block, MARSARA checks if there exists an edge $(\omega, u) \in E$ between the last matched and current node. If it exists, MARSARA considers the path to be valid and updates that last matched node to be u. If a discrepancy is found, it is invalidated.

Warning Types. When MARSARA detects invalid events, it records warnings of two severity levels: low and critical. Currently, warnings are intended only to provide verbosity so we can empirically evaluate MARSARA's accuracy. They do not need to be considered by investigators and we leave the possibility of using them to aid in investigations to future work.

The severity is based on what kind of discrepancy is detected in the model. In benign experiments where no attack is occurring, if a direct code branch causes a warning, it is ranked low because this is due to a missed path during offline profiling and can be resolved using more data. Recall that all prior work also relies on offline profiling and therefore cannot guarantee completeness.

Conversely, if the inconsistency (in benign experiments) arises from indirect transfers (indirect jump, indirect call, return), it is ranked critical since this is a limitation in the symbolic analysis used during offline profiling. This represents a limitation that cannot be resolved with more data, which is why we differentiate it from low warnings. Fortunately, as we demonstrate in our evaluation, these are rare, meaning that our design is effective overall.

5.5 Execution Partitioning

MARSARA's partitioning logic relies on the observation that developers of long-running processes create log messages for the important events in each execution unit's lifecycle. For example, for a web server that handles user requests, it is customary for developers to log the user's request at the start of each unit. Such log messages often reside at the start of an event-handling function (typically a function pointer) or an infinite loop, which is why our binary analysis in Subsection 5.2 labeled them explicitly.

However, determining which log messages signal the start of a new execution unit without semantic analysis of the message's content is a challenging task. To overcome this, we combine information about loops and functions from the offline profiling phase with runtime information about log messages to uncover the heads of execution units.

As discussed in Section 5.2, MARSARA assigns each code block v with a label $\ell(v)$ indicating whether v is an infinite loop or the head of a function. Such blocks become candidates for starting new execution units. MARSARA keeps a running count of the number of times a log messages has been encountered in a priority queue. The intuition behind this approach lies in the observation that application developers, in an effort to reduce the performance overhead of logging, restrict the log messages to important events, the most important of which is the servicing of a new input. Therefore, the log message at the top of the priority queue (i.e., the one with the largest count) likely corresponds to the head of an execution unit. Every time that message is encountered, MARSARA performs a backward search in the current trace and identifies the closest code block that is either an infinite loop head, or the head of a function with no incoming edges in the model. MARSARA then creates a new execution unit starting from that block and adds all subsequent events to the new unit.

6 EVALUATION

We evaluate MARSARA with an emphasis on answering the following research questions:

Table 1: Performance, accuracy, and storage overhead of MARSARA. Time captures the seconds to analyze and validate events. Baseline storage corresponds to running the Linux Audit framework and application log tracking without MARSARA. The low warnings are categorized by the model edge type for additional granularity.

Program	Model		Total Events	Time (sec)	Warnings					Storage (MB)	
					Low			Critical	FPR		
	Blocks	Edges			Forward Edges	Backward Edges	Other			Baseline	MARSARA
cupsd	4,768	32,521	15,592	0.109	1	20	0	0	0.13%	0.218	0.067
HAProxy	28,837	188,422	69,009	0.241	131	264	11	5	0.59%	0.141	0.244
httpd	7,087	25,465	419,532	1.237	187	226	0	6	0.09%	0.433	1.613
lighttpd	5,680	24,862	508,707	0.967	179	134	5	3	0.06%	0.277	2.436
memcached	38,427	200,041	4,282	0.082	82	32	7	0	2.82%	0.300	0.219
nginx	15,675	99,924	175,239	0.511	265	326	2	0	0.33%	0.310	0.722
postfix	146,296	476,904	2,968	0.043	28	2	5	2	1.24%	0.898	0.010
Proftpd	10,918	70,767	3,050,246	15.214	305	229	4	0	0.01%	0.630	11.181
Redis	28,881	161,294	2,681,711	21.357	334	416	3	0	0.02%	0.483	15.007
squid	32,516	109,804	116,100	0.436	170	118	2	0	0.24%	0.652	0.583
thttpd	32,725	203,385	12,589,818	22.361	48	17	6	4	0.00%	0.206	33.681
Transmission	7,045	27,765	173,705	0.397	236	154	80	1	0.27%	0.282	0.031
wget	6,979	49,028	17,624	0.048	74	63	1	0	0.78%	0.095	0.088
yafc	3,621	18,981	31,170	0.318	60	39	5	2	0.33%	0.114	0.105

(1) What is MARSARA's accuracy when validating the integrity of partitions? We measure its accuracy in terms of the number of warnings generated over benign inputs in 14 real-world programs and show that only 2.82% of false dependencies are reintroduced at worst.
(2) How much does MARSARA reduce the vulnerability of programs to EUP attacks compared to CFI alone? We measure PAS for the same real-world programs while being protected by MARSARA, shadow stack, and function-level CFI. MARSARA removes 47,642 more gadgets per program.
(3) Can MARSARA prevent execution repartitioning attacks based on the techniques from Section 3? We attack several programs using expertly crafted exploits and find that MARSARA successfully preserves the full attack provenance.
(4) What is the cost of MARSARA's forensic analysis? We measure the overhead for the real-world programs and the SPEC CPU 2006 benchmark compared to a standard auditing framework and find it to be 8.7%, on average.

Experimental Setup. We evaluate MARSARA using 14 popular real-world applications. These programs have frequently been used to evaluate prior work [36, 49, 50, 58, 59], justifying their inclusion. We use the default configurations and generate workloads with standard benchmark tools, such as Apache Benchmark [26]. We also evaluate against the SPEC CPU 2006 benchmark, with full workloads, for direct comparison with prior work.

For practical binary CFI defenses, we consider shadow stack and function-level policies, which are realistic to enforce without source code. Shadow stack prevents control flow hijacking from arising via corrupted return pointers whereas function-level CFI additionally enforces that indirect calls and jumps must target the start of a valid function. More accurate policies have been proposed, but have not seen real-world deployment due to requiring source code, being incompatible with mechanisms like stack unwinding, and/or having overheads upwards of 47% [39].

We conduct our tests on a server-class machine with an Intel Core(TM) i7-6700K CPU @ 4.00GHz and 16GB of memory, running Debian 10. Audit logs are collected using Linux Audit with rules covering the most commonly used system calls, such as `read`, `write`, and `execve` (23 in total).

Definition of Errors. For the purposes of this evaluation, a *false positive* is defined as a legitimate audit event that is accidentally detected during MARSARA's integrity check, yielding a warning, and a *false negative* is a spoofed or delayed event that is not. In terms of the resulting provenance graph, a false positive *may* introduce a false dependency edge whereas a false negative *may* remove a true dependency edge.[6]

Calculations. Overhead is calculated as $(P - B)/B$ where B is the baseline performance value and P is the value with the evaluated system enabled. False positive rate (FPR) for Table 1 is calculated as the sum of all warnings divided by total events. We do not report the time to produce models since this is only done once per program during the offline phase.

6.1 Partition Validation Accuracy

Table 1 shows the performance and accuracy of MARSARA's analysis for validating the execution partitions. As expected from Algorithm 2, the time to validate is linear to the number of events recorded. In the largest observed case (12 million events, `thttpd`), MARSARA analyzes and validates the trace in less than 30 seconds. This is reasonable since verification is only required once per trace and is not performed until an investigation occurs (i.e., the post-forensic analysis phase).

We also report the number of events yielding false positive warnings (FPs) during verification. For 8 of the 14 applications,

[6]Notice that if a false positive happens to be a true dependency, the graph is unaffected, and if a false negative fails to forge a signature match, the graph is also unaffected.

Table 2: PAS for several real-world programs and defenses.

Program	ICTs	None	SS	Func.	MARSARA
cupsd	4,017	19.42	8.62	8.59	**8.33**
HAProxy	13,155	2.49	2.18	2.18	**2.11**
httpd	1,779	40.00	12.14	12.02	**9.41**
lighttpd	2,858	0.18	0.13	0.13	**0.13**
memcached	797	2.82	1.10	1.02	**0.89**
nginx	3,997	0.80	0.37	0.37	**0.28**
postfix	848	16.00	10.28	9.75	**9.42**
Proftpd	34,830	2.18	0.82	0.81	**0.72**
Redis	28,047	7.09	5.37	5.34	**5.06**
squid	18,412	353.00	196.03	181.11	**123.94**
thttpd	1,198	1.02	0.14	0.14	**0.11**
Transmission	17,507	2.89	1.83	1.82	**1.75**
wget	16,594	6.71	0.85	0.71	**0.64**
yafc	8,590	0.85	0.64	0.63	**0.62**
Average:	**10,902**	**32.53**	**17.18**	**16.04**	**11.67**

MARSARA reports no critical FPs, meaning that the symbolic analysis used during the offline profiling phase works well on the evaluated programs. For the remaining programs, the FPs are <6, highlighting only a few troublesome model edges.

FPs occur mainly for two reasons: due to limitations in binary symbolic analysis and inaccuracies in reporting system calls. In some cases, MARSARA detects system calls that do not map back to nodes in the model. For example, in Transmission, unexpected `openat` system calls are recorded. Investigation reveals that the function `tr_variantToFile` makes a call to the `libc` method `mkstemp`. However, when examining the model, we did not find a node for this method, indicating that symbolic execution was not able to analyze it. We further investigated the source code for `mkstemp` in `glibc` and observed that it is replaced by the compiler with a function called `__gen_tempname`[7]. These kinds of optimizations are not currently handled by the verification algorithm, but will be addressed in future versions.

We also report the number of events yielding low severity warnings, which arise in direct branches not covered by the profiling traces we collected during the offline phase. For additional clarity, we categorize these into forward graph edges (calls, jumps), backward edges (return), and other (unexpected audit log events). The evaluated programs yield between 10 and 600 low warnings, which we explain the impact of next.

Since this experiment does not contain any exploits, all generated warnings are false positives, i.e., legitimate events wrongly detected by MARSARA's integrity check. This is presented in the table as FPR, calculated as the number of warning-producing events (low and critical) divided by the total number of events. In all cases, FPR is 2.82% or lower. Recall that if a false positive pertains to a false dependency, it will be preserved in the resulting provenance graph as an edge rather than being removed during partitioning. A false positive detection of a true dependency is of no consequence, since it would not have been removed anyway. Consequently, FPR is also the maximum number of false dependencies that can be reintroduced into the graph. For example, if the ideal partitioned provenance graph for a given query contains 1,000 dependencies (edges), the resulting graph with a FPR of 2.82% could contain up to 1,028 edges (28 false dependencies), presenting little difference to analysts or downstream systems. In short, MARSARA almost completely preserves the false dependency reduction of prior (insecure) EUP techniques with the added benefit of integrity.

[7] Observed in `glibc/misc/mkstemp.c` at line 33.

6.2 Partitioning Attack Surface Reduction

Table 2 presents MARSARA's PAS for the real-world programs compared to the unprotected binaries and several practical binary CFI policies, along with the number of indirect control transfers (ICTs) in each program. Recall from Subsection 4.1 that smaller values equate to greater protection against EUP attacks.

Across all measured programs, MARSARA's PAS is better than any of the CFI defenses. Since most programs contain over 1,000 ICTs, even small reductions in PAS are significant. For example, MARSARA reduces Proftpd's PAS by 0.09 versus function-level CFI, which over 34,830 ICTs equates to eliminating 3,134 events that an attacker could otherwise leverage to spoof EUP signatures. In the simpler programs, the benefits are more modest. For example, `lighttpd` gains little added protection from MARSARA, or function-level CFI for that matter, due to not having any indirect calls or jumps. The biggest benefit is observed in Squid, where its modular design presents the opportunity for MARSARA to reduce PAS by *57.17* over function-level CFI, eliminating over *1,052,614* event gadgets. On average, 47,642 additional gadgets are removed compared to function-level CFI. In short, MARSARA successfully eliminates thousands (and sometimes millions) of options for an attacker attempting to spoof an EUP signature, even in programs already protected by binary CFI.

6.3 Attack Investigation

To evaluate MARSARA's integrity, we use the expertly crafted exploits described in Subsection 3.3 to attack real-world programs. Specifically, we first run EUP without PT or MARSARA's partition verification (essentially placing partitions as prior systems would, creating a baseline for comparison) to confirm that the exploits produce valid (malicious) signatures for partitioning. As expected, all 3 attacks successfully manipulated prior EUP algorithms into fragmenting the attacker's exploit and resulting symptoms across disjoint partitions. In short, without MARSARA, provenance queries made by investigators will be answered with seemingly legitimate (but actually misleading and incomplete) results.

We then rerun the attacks, now with MARSARA. In the 2 control hijacking cases (CVE-2013-2028, CVE-2004-0541), we observe critical warnings at the point where the exploits redirect control of the execution. For CVE-2009-4769, the critical warning arises because the model reveals, based on the call site to the logging method, that the resulting message in the audit log contradicts the expected format. Consequently, MARSARA does not fragment the attacker's network requests from the rest of the symptoms, yielding complete provenance attack graphs that contain all the relevant events. For example, for CVE-2013-2028, which pertains to our motivating example originally visualized in Figure 1, MARSARA's partition includes both the events pertaining to `x.x.x.x` and `y.y.y.y`. *In short, this experiment yields no false negatives.*

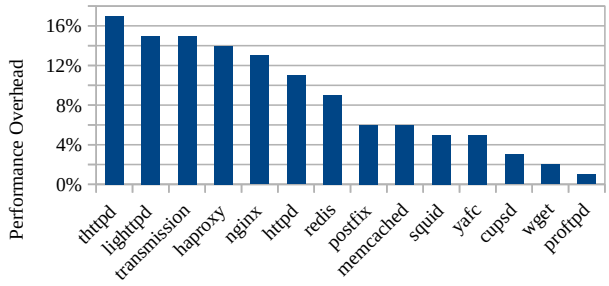

Figure 5: Performance overhead for the real-world programs. The average is 8.7%.

6.4 Runtime & Space Overhead

Real-World Programs. We report the storage requirements for MARSARA's analysis in the last two columns of Table 1. Our baseline represents the amount of compressed data needed to store the events generated by the Linux Audit framework. We compare that to the amount of extra storage (also in compressed form) that MARSARA requires for PT.

For 9 of the 14 applications we evaluated, MARSARA's storage requirement is in the same order as the baseline (e.g., 1.6 MB for 500K events in the case of `httpd`). However, for each Linux Audit trace, the corresponding PT trace can be discarded after MARSARA's validation is completed. This renders the PT storage overhead as only a temporary cost.

For 3 applications (`thttpd`, `Proftpd`, and `Redis`), a large number of PT events are generated, requiring significantly more temporary storage. Investigating further, we discover that MARSARA reports on events pertaining to several loop blocks engaged in "busy-waiting" behavior for initializing large arrays. For example, `thttpd` creates an array for storing all the possible file descriptors (1024 in our evaluation environment) and then initializes each element to −1. Consequently, every time this code block is executed, PT records a path consisting of 1024 blocks, significantly increasing the number of events generated. We discuss possible solutions to PT's storage requirements in Section 7.

Figure 5 shows MARSARA's runtime overhead compared to the baseline of Linux Audit framework with no PT event tracking. MARSARA's average runtime overhead is 8.7%, which is consistent with prior PT systems [27, 39, 90]. The overhead observed varies depending on the profiled application's behavior. For example, applications that are mostly IO-bound, such as caching servers (`memcached`, `squid`), file, mail, printing servers (`proftpd`, `postfix`, and `cupsd`), and key-value stores (`redis`) exhibit low runtime overhead, ranging from 1% for `proftpd` to 9% for `Redis`. Conversely, applications that are more CPU-intensive, such as web servers and load balancers, incur a larger overhead (up to 17% for `thttpd`) since PT yields more events. We will consider alternative methods to reduce PT's runtime overhead for CPU-intensive applications in future work.

SPEC CPU 2006. To provide an additional standard benchmark for comparison, we also report the performance overhead of monitoring the SPEC CPU 2006 benchmark programs over all provided workloads, visualized in Figure 6. Across the SPEC programs, MARSARA yields an average performance overhead of 7.21%, which is consistent with the results from monitoring the 14 real-world programs that are typically used in provenance system evaluations. However, we also note that some of the SPEC programs produce noticeably higher overhead due to the amount of PT data they produce. This is to be expected since the benchmark is designed to stress CPUs, making the workloads CPU-bound, whereas the other programs we evaluate are mostly I/O-bound. We believe the non-SPEC workloads are more representative of the programs an EUP attack would target, so we conclude that the SPEC performance results are tolerable.

7 DISCUSSION

Improving Model Accuracy. The current MARSARA prototype relies on binary single-path symbolic execution to generate the model during offline profiling. This results in an under-approximated set of paths. Although we consider improving the state of binary analysis to be outside our scope, several possible solutions exist to improve its accuracy.

For example, because MARSARA already records the full PT trace and system call audit for protected programs, it is possible to use the collected data to guide an offline replay. Specifically, when MARSARA encounters an inconsistency due to a missing edge in the model, an existing record and replay (R&R) system [25, 42, 64] can re-execute the program offline with additional instrumentation (e.g., Valgrind [82]) to detect the presence of memory corruptions and then refine the model appropriately. Although memory-safe R&R is expensive, the cost would be paid in an offline analysis and each newly encountered path would only need to be tested once. In time, the model would converge to the ground truth graph with a priority towards refining execution paths actually observed in real-world executions.

We also note that symbolic execution does not scale to all programs, particularly complicated ones like web browsers. However, by evaluating a prototype that uses application message and system call auditing, designed as an extension of the most recent work, we demonstrate that our approach of using PT and binary symbolic analysis to verify signatures can benefit the security of all EUP-dependent systems, not just our prototype. We also demonstrate that even in its current form, MARSARA protects logs derived from important web services.

Improving Storage Overhead. While most of the tested binaries produce audit logs comparable in size to the baseline system considered in Section 6, we encounter some cases where sizes are an order of magnitude larger. We discover the cause of this phenomenon to be non-blocking event loops (i.e., "busy waiting"), which yield many control flow events of little significance (i.e., checking a flag and then returning to the loop head). This can be addressed as the PT trace is decoded by summarizing loops or using compression tailored to our problem context. Note that decreasing the PT trace size will also benefit performance, since less data has to be processed.

In a similar vein, while our PT-enabled kernel is capable of tracing programs with dynamically generated code (e.g., JIT in browsers), doing so is likely to yield higher performance overhead as each generated code page has to be captured in the sideband data. We leave these optimizations to future work.

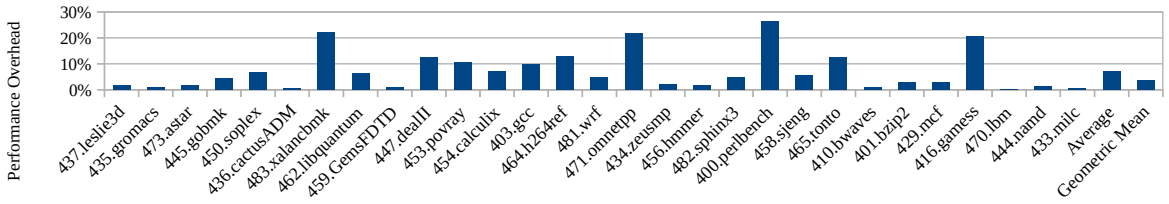

Figure 6: Performance overhead for the SPEC CPU 2006 benchmark. The average is 7.21% and the geometric mean is 3.81%.

Compatibility with Other Languages. MARSARA's reliance on an instrumented `libc` means it will not be able to capture application messages for all possible Linux programs. However, fixating on this detail overlooks two points that are more significant. First, our prototype's EUP signatures contain messages *and* system calls. Even when the former is unavailable due to compatibility, the latter can still be used to identify units of execution, albeit at a coarser granularity. EUP is still valuable in such cases [49, 56]. Second, the purpose for including application messages in our design is to demonstrate the flexibility of our modeling to serve a wide range of analyses that require EUP, not just those reliant on one data source (e.g., system calls).

Compatibility with "At Rest" Integrity. In this work, we focus on protecting log integrity against a novel form of online tampering based on EUP attacks. This is outside the scope of prior work, which focuses on tampering performed to data *at rest* on storage. Our proposed defense complements the protection offered by these past solutions and MARSARA can be extended to incorporate them into a holistic system. For example, solutions based on cryptography can be readily applied to the data produced by MARSARA, thereby adding storage integrity. Similarly, MARSARA can control where data is stored, allowing it to leverage trusted storage solutions like WORM drives or central logging servers.

8 RELATED WORK

8.1 Attack Reconstruction

We are the first work to analyze binary events during system-level provenance collection and solve the challenges associated with protecting the integrity of EUP signature matches. A lot of work has been done to leverage provenance for forensic analysis [4, 34, 42, 49, 50, 52, 56–60, 85], network debugging, auditing and troubleshooting [2, 19–21, 95], alert triage [34, 35], and intrusion detection and access control [3, 14, 68]. MARSARA complements all these systems by offering more secure EUP. Finally, our work also complements the existing EUP systems such as BEEP [50], MPI [58], and MCI [49], which improve post-mortem analysis by solving the problem of dependency explosion.

A large amount of research effort has focused on the generation and use of system call logs in forensic analysis, investigation, and recovery [5, 29, 46, 47, 70, 84]. However, none of the existing work focuses on defending post-mortem analysis against execution repartitioning attacks. Provenance visualization techniques [11, 12] are also proposed to facilitate causality analysis. MARSARA can leverage these techniques to provide provenance graph summaries to admins, accelerating threat investigations.

Several systems [44, 66] have been proposed to detect the tampering of audit logs. Both Custos and SGX-Log use protocols that leverage Intel SGX and cryptographic data structures to protect audit log integrity. Several formats have also been proposed in the literature for storing data in a tamper-evident fashion, such as history trees [24, 71] and hash treaps [71]. These tamper-evident systems only detect if certain entries in the audit log are modified *after being committed*, which is orthogonal to the online threat we model in this work.

8.2 Log Deduplication and Compression

Our work is orthogonal to provenance graph compression and deduplication techniques [17, 22, 86], since they compress the provenance graph instead of defending against EUP attacks. Many approaches [3, 6, 8, 22, 28, 33, 56, 57, 59, 78, 86, 87] are proposed to reduce the size of audit log for long-term storage and to speed up after-the-fact forensic analysis. MARSARA can leverage those techniques to reduce its storage overhead.

LogGC [51] provides offline techniques to garbage collect redundant events that have no forensic value. Similarly, Winnower [33] and Process-centric Causality Approximation [89] both reduce log size by over-approximating causal relations. These techniques can be applied alongside our work to decrease storage overhead. We can also use these approaches to speed up our analysis.

8.3 Control Flow Bending

Control flow bending is the most prevalent way attackers exploit memory corruption vulnerabilities. From the attack perspective, we have seen a rise in sophistication from code injection, to code reuse (e.g., ret2libc [61]), to what is now the predominate exploitation technique: return-oriented programming (ROP) [9, 10, 13, 18, 75, 77]. For defenses, we have seen proposals based on randomization, including ASLR [69], which have been successfully deployed in common OSes. Unfortunately, there is still an ongoing battle between circumvention [30] and better defenses [53, 54].

Another defense is control flow integrity (CFI) [1], which aims to ensure that the program adheres to a predetermined model, thereby reducing the attacker's ability to exploit paths unintended by the developer. Unfortunately, CFI has only seen limited adoption due to conflicts between performance and security. Coarse-grained solutions [93, 94] are fast and compatible with existing programs, but can be bypassed with careful bending [16]. Fine-grained approaches reduce the attack surface [63, 81, 83], but can still be bypassed, require source code, or rely on special hardware for performance [39]. In short, there is no ideal CFI solution to date [88].

In this work, control flow bending is one means by which attackers can conduct EUP attacks, but they can also utilize format string vulnerabilities and other orthogonal classes of bugs. We are the first to propose that online exploitation can explicitly target EUP to hinder forensic investigation. Prior work on bending may evade CFI, but leave the provenance chain intact, posing no hindrance on the attack investigation. Even when CFI is already deployed, MARSARA demonstrates an empirical benefit in terms of PAS.

9 CONCLUSION

This work presents the first formal exploration of online anti-forensic attacks against data provenance leveraging software exploits. We demonstrate that attackers can break the causal links in data provenance graphs used for forensic investigation, and even frame benign subjects, without triggering existing tamper-evident logging defenses. We propose MARSARA to verify EUP signature matches and demonstrate that it resists expertly crafted exploits while reintroducing no more than 2.82% of false dependencies, across 14 real-world programs, with a performance overhead of 8.7%. Compared to CFI, MARSARA removes 47,642 more gadgets per program.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-19-1-2179, N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662, the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031, and the National Science Foundation (NSF) under grants CNS-1750024 and CNS-2055127. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR, DARPA, or NSF.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*.

[2] Adam Bates, Kevin Butler, Andreas Haeberlen, Micah Sherr, and Wenchao Zhou. 2014. Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. In *NDSS Workshop on Security of Emerging Networking Technologies (SENT'14)*.

[3] Adam Bates, Kevin R. B. Butler, and Thomas Moyer. 2015. Take Only What You Need: Leveraging Mandatory Access Control Policy to Reduce Provenance Storage Costs. In *7th Workshop on the Theory and Practice of Provenance* (Edinburgh, Scotland) *(TaPP'15)*.

[4] Adam Bates, Wajih Ul Hassan, Kevin R.B. Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. 2017. Transparent Web Service Auditing via Network Provenance Functions. In *26th World Wide Web Conference (WWW'17)*. Perth, Australia.

[5] Adam Bates, Dave Tian, Kevin R.B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *Proceedings of 24th USENIX Security Symposium* (Washington, D.C.).

[6] Adam Bates, Dave Tian, Grant Hernandez, Thomas Moyer, Kevin R.B. Butler, and Trent Jaeger. 2017. Taming the Costs of Trustworthy Provenance through Policy Reduction. *ACM Trans. on Internet Technology* 17, 4 (sep 2017), 34:1–34:21.

[7] Mihir Bellare and Bennet Yee. 1997. *Forward integrity for secure audit logs*. Technical Report. Computer Science and Engineering Department, University of California at San Diego.

[8] Y. Ben, Y. Han, N. Cai, W. An, and Z. Xu. 2018. T-Tracker: Compressing System Audit Log by Taint Tracking. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 1–9. https://doi.org/10.1109/PADSW.2018.8645035

[9] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*.

[10] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*.

[11] Michelle A Borkin, Azalea A Vo, Zoya Bylinskii, Phillip Isola, Shashank Sunkavalli, Aude Oliva, and Hanspeter Pfister. 2013. What makes a visualization memorable? *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2306–2315.

[12] Michelle A Borkin, Chelsea S Yeh, Madelaine Boyd, Peter Macko, Krzysztof Z Gajos, Margo Seltzer, and Hanspeter Pfister. 2013. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2476–2485.

[13] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*.

[14] Frank Capobianco, Christian Skalka, and Trent Jaeger. 2017. ACCESSPROV: Tracking the Provenance of Access Control Decisions. In *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*.

[15] Carbon Black. 2018. Global Incident Response Threat Report. https://www.carbonblack.com/global-incident-response-threat-report/november-2018/. Last accessed 04-20-2019.

[16] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium*.

[17] Adriane Chapman, H.V. Jagadish, and Prakash Ramanan. 2008. Efficient Provenance Storage. In *Proceedings of the 2008 ACM Special Interest Group on Management of Data Conference* (Vancouver, Canada) *(SIGMOD'08)*.

[18] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*.

[19] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. One Primitive to Diagnose Them All: Architectural Support for Internet Diagnostics. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 374–388. https://doi.org/10.1145/3064176.3064212

[20] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2015. Differential Provenance: Better Network Diagnostics with Reference Events. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets'15)* (Philadelphia, PA).

[21] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. ACM, New York, NY, USA, 115–128. https://doi.org/10.1145/2934872.2934910

[22] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. 2017. Distributed Provenance Compression. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 203–218. https://doi.org/10.1145/3035918.3035926

[23] Catalin Cimpanu. [n.d.]. Hackers are increasingly destroying logs to hide attacks. https://www.zdnet.com/article/hackers-are-increasingly-destroying-logs-to-hide-attacks/. Last accessed 04-20-2019.

[24] Scott A. Crosby and Dan S. Wallach. 2009. Efficient data structures for tamper-evident logging. In *In Proceedings of the 18th USENIX Security Symposium*.

[25] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. 2014. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 525–540.

[26] Apache Software Foundation. [n.d.]. Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[27] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 585–598.

[28] Ashish Gehani, Minyoung Kim, and Jian Zhang. 2009. Steps Toward Managing Lineage Metadata in Grid Clusters. In *1st Workshop on the Theory and Practice of Provenance* (San Francisco, CA) *(TaPP'09)*.

[29] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference* (Montreal, Quebec, Canada) *(Middleware '12)*.

[30] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*.

[31] Steve Hales. [n.d.]. Last Door Log Wiper. https://packetstormsecurity.com/files/118922/LastDoor.tar. Last accessed 04-20-2019.

[32] Gunnar Hartung, Björn Kaidel, Alexander Koch, Jessica Koch, and Dominik Hartmann. 2017. Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures. In *Proc. of the International Conference on Provable Security*

(ProvSec).

[33] Wajih Ul Hassan, Nuraini Aguse, Mark Lemay, Thomas Moyer, and Adam Bates. 2018. Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS'18)*. San Diego, CA, USA.

[34] Wajih Ul Hassan, Adam Bates, and Daniel Marino. 2020. Tactical Provenance Analysis for Endpoint Detection and Response Systems. In *41st IEEE Symposium on Security and Privacy (SP) (Oakland'20)*.

[35] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *26th ISOC Network and Distributed System Security Symposium (NDSS'19)*.

[36] Wajih Ul Hassan, Mohammad Noureddine, Pubali Datta, and Adam Bates. 2020. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20)*.

[37] Jason E. Holt. 2006. Logcrypt: Forward Security and Public Verification for Secure Audit Logs. In *Proc. of the Australasian Information Security Workshop (AISW-NetSec)*.

[38] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott Stoller, and V.N. Venkatakrishnan. 2017. SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 487–504. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hossain

[39] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1470–1486.

[40] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*. IEEE, 969–986.

[41] IBM Knowledge Center. [n.d.]. Storage and analysis of audit logs. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.sec.doc/doc/c0052328.html. Last accessed 04-20-2019.

[42] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. RAIN: Refinable Attack Investigation with On-Demand Inter-Process Information Flow Tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 377–390. https://doi.org/10.1145/3133956.3134045

[43] JustLinux Forums. [n.d.]. server hacked!! /var/log deleted. how can i trace hacker!?! http://forums.justlinux.com/showthread.php?123851-server-hacked-var-log-deleted-how-can-i-trace-hacker. Last accessed 04-20-2019.

[44] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing System Logs With SGX. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*.

[45] Kent Karen and Souppaya Murugiah. 2006. NIST Special Publication 800-92, Guide to Computer Security Log Management.

[46] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution. In *OSDI*. USENIX Association. http://dl.acm.org/citation.cfm?id=1924943.1924950

[47] Samuel T. King and Peter M. Chen. 2003. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/945445.945467

[48] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 503–515. https://doi.org/10.1145/2872362.2872395

[49] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *Proc. of the 25th Network and Distributed System Security Symposium (NDSS'18)*.

[50] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of NDSS '13* (San Diego, CA).

[51] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: Garbage Collecting Audit Log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. ACM, New York, NY, USA, 1005–1016. https://doi.org/10.1145/2508859.2516731

[52] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS'18)*. San Diego, CA, USA.

[53] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*.

[54] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[55] Di Ma and Gene Tsudik. 2009. A new approach to secure logging. *ACM Transactions on Storage (TOS)* 5, 1 (2009).

[56] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows. In *Proceedings of the 31st Annual Computer Security Applications Conference* (Los Angeles, CA, USA) *(ACSAC 2015)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/2818000.2818039

[57] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. 2018. Kernel-Supported Cost-Effective Audit Logging for Causality Tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 241–254. https://www.usenix.org/conference/atc18/presentation/ma-shiqing

[58] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *26th USENIX Security Symposium*.

[59] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings of NDSS '16* (San Diego, CA).

[60] S. Momeni Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan. 2019. HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/SP.2019.00026

[61] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIG-PLAN Conference on Programming Language Design and Implementation*.

[62] National Institute of Standards and Technology. 2013. NIST Special Publication 800-53 (Rev. 4), Security Controls and Assessment Procedures for Federal Information Systems and Organizations.

[63] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[64] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 377–389.

[65] OccupytheWeb. 2013. How to Cover Your Tracks & Leave No Trace Behind on the Target System. https://tinyurl.com/yygqte9p. Last accessed 04-20-2019.

[66] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. 2020. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20)*.

[67] Riccardo Paccagnella, Kevin Liao, Dave (Jing) Tian, and Adam Bates. 2020. 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 (CCS'20)*.

[68] Jaehong Park, Dang Nguyen, and R. Sandhu. 2012. A Provenance-Based Access Control Model. In *Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST)*. 137–144. https://doi.org/10.1109/PST.2012.6297930

[69] PaX Team. 2003. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[70] D.J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. 2012. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proceedings of the 2012 Annual Computer Security Applications Conference (ACSAC '12)*. Orlando, FL, USA.

[71] Tobias Pulls and Roel Peeters. 2015. Balloon: A forward-secure append-only persistent authenticated data structure. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*.

[72] Rapid7. [n.d.]. Metasploit, the world's most used penetration testing framework. https://www.metasploit.com/. Last accessed 04-20-2019.

[73] Bruce Schneier and John Kelsey. 1998. Cryptographic Support for Secure Logs on Untrusted Machines.. In *Proc. of the USENIX Security Symposium (USENIX)*.

[74] Bruce Schneier and John Kelsey. 1999. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)* (1999).

[75] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*.

[76] Yan Shoshitaishvili, Ruoyu (Fish) Wang, Andrew Dutcher, Christophe Hauser, John Grosen, Chris Salls, Nick Stephens, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. 2017. angr, a binary analysis framework. http://angr.io/.

[77] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*.

[78] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. 2018. NodeMerge: Template Based Efficient Data Reduction For Big-Data Causality Analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1324–1337. https://doi.org/10.1145/3243734.3243763

[79] Dawood Tariq, Maisem Ali, and Ashish Gehani. 2012. Towards Automated Collection of Application-Level Data Provenance. In *4th USENIX Workshop on the Theory and Practice of Provenance*. USENIX, Boston, MA. https://www.usenix.org/conference/tapp12/workshop-program/presentation/Tariq

[80] The MITRE Corporation. 2017. CAPEC-81: Web Logs Tampering. https://capec.mitre.org/data/definitions/81.html. Last accessed 04-20-2019.

[81] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*.

[82] Valgrind Developers. 2017. Valgrind. http://www.valgrind.org/.

[83] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*.

[84] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2017. Fear and Logging in the Internet of Things. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS'18)*.

[85] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Zhen, Wei Cheng, Carl A. Gunter, and Haifeng chen. 2020. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20)*.

[86] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell D.E. Long. 2012. A Hybrid Approach for Efficient Provenance Storage. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management* (Maui, Hawaii, USA) *(CIKM '12)*.

[87] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell D. E. Long. 2013. Evaluation of a Hybrid Approach for Efficient Provenance Storage. *Trans. Storage* 9, 4, Article 14 (Nov. 2013), 29 pages. https://doi.org/10.1145/2501986

[88] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. 2019. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1805–1821. https://www.usenix.org/conference/usenixsecurity19/presentation/xu-xiaoyang

[89] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 504–516. https://doi.org/10.1145/2976749.2978378

[90] Carter Yagemann, Salmin Sultana, Li Chen, and Wenke Lee. 2019. Barnum: Detecting Document Malware via Control Flow Anomalies in Hardware Traces. In *International Conference on Information Security*. Springer, 341–359.

[91] Attila Altay Yavuz and Peng Ning. 2009. BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[92] Attila A Yavuz, Peng Ning, and Michael K Reiter. 2012. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In *Proc. of the International Conference on Financial Cryptography and Data Security (FC)*.

[93] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*.

[94] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*.

[95] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure Network Provenance. In *ACM Symposium on Operating Systems Principles (SOSP)*.