PalanTír: Optimizing Attack Provenance with Hardware-enhanced System Observability

Jun Zeng*
National University of Singapore
junzeng@comp.nus.edu.sg

Chuqi Zhang*
National University of Singapore
chuqiz@comp.nus.edu.sg

Zhenkai Liang
National University of Singapore
liangzk@comp.nus.edu.sg

ABSTRACT

System auditing is the foundation of attack provenance to investigate root causes and ramifications of cyber-attacks. However, provenance tracking on coarse-grained audit logs suffers from false causalities caused by dependency explosion. Recent approaches address this problem by increasing provenance granularity using execution partitioning or record-and-replay techniques. Unfortunately, they require program instrumentation and/or impose an unaffordable overhead, which is not practical in deployment.

In this paper, we present PalanTír, a provenance-based system that enhances *system observability* to enable precise and scalable attack investigation. Leveraging hardware-assisted processor tracing (PT), PalanTír optimizes attack provenance in system-call-level audit logs by recovering instruction-level causalities via taint analysis based on PT traces. To reduce the scope of taint analysis and simplify the complexity of taint propagation, PalanTír statically profiles program binaries to identify instructions causally relevant to audit logs and pre-summarize their taint propagation logic at the coarse granularity of basic blocks. Our evaluation against real-life cyber-attacks shows PalanTír's efficiency and effectiveness in attack scenario reconstruction. We also demonstrate that PalanTír can scale to large applications (e.g., Nginx and Sendmail) compiled from upwards of 463,510 lines of C/C++ code.

CCS CONCEPTS

• **Security and privacy** → **Systems security**.

KEYWORDS

Attack Provenance; System Auditing; Processor Tracing

ACM Reference Format:
Jun Zeng, Chuqi Zhang, and Zhenkai Liang. 2022. PalanTír: Optimizing Attack Provenance with Hardware-enhanced System Observability. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560570

1 INTRODUCTION

Large enterprises (e.g., Equifax [3] and Twitter [4]) are often the targets of advanced cyber-attacks, resulting in personal information leakage or fraudulent access to financial services. Attackers have also become more stealthy by carrying out the attacks in longer time spans. For example, the Solarwinds hack went undetected for more than nine months and leaked sensitive data from government agencies and technology companies [8].

In order to understand the motivations behind cyber-attacks, a promising method is to reconstruct *attack provenance* [16, 34, 42], where a *provenance graph* is built upon system-call-level audit logs to investigate how an attacker gains access to a system and what damages are inflicted. Specifically, a provenance graph encodes dependencies among system entities, i.e., processes, files, and network sockets. Once a compromised system entity is detected, security analysts can traverse the graph to discover the root cause of the attack and its ramifications [52, 53].

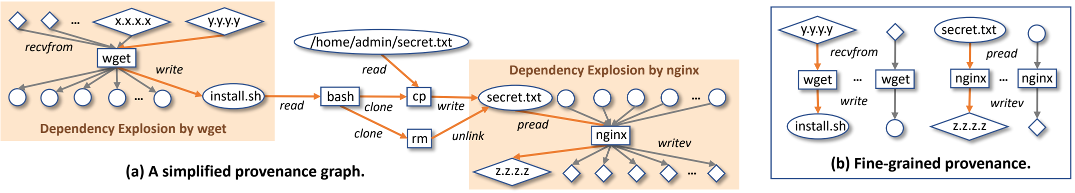
However, provenance analysis based on system-call logs can only track coarse-grained information flows in systems, conservatively assuming the output of a process dependent on *all* its preceding inputs. Consequently, provenance-based attack investigation usually suffers from the *dependency explosion problem* [55], especially for long-running processes. For example, Nginx is a widely-used web server in which network requests are handled by a worker process. Suppose a `nginx` process receives multiple requests for `index.html` and `secret.txt`. It is challenging for an analyst to distinguish which request(s) corresponds to `secret.txt` since it depends on all the preceding requests.

To address this problem, prior work separates information flows in a process, e.g., by partitioning a process into autonomous execution units [55, 64, 65]. Subsequently, only audit logs within a unit are considered causally relevant. However, most solutions for execution partitioning require application instrumentation to mark unit boundaries, presuming permissions to modify software programs, which is not practical for deployment in production environments [63]. Besides calling for instrumentation, recent studies [41, 89, 90] leverage developer built-in application logs to identify execution units. Nonetheless, the quality of application logs is not always reliable due to the lack of concrete logging specifications and guidelines [31, 59], which may lead to false-positive and false-negative execution units. Another line of research adopts record-and-replay systems [46, 47] to record non-deterministic program executions that are later replayed to conduct instruction-level dynamic information flow tracking (i.e., *taint analysis* [72]). Unfortunately, record-and-replay systems also require instrumentation and commonly impose an unaffordable slowdown for multi-threaded programs running on multi-cores [24].

The fundamental question to optimize attack provenance is how to enhance *system observability* [82] — a measure of how well the internal states of a system can be inferred from knowledge of its external outputs. The key to high observability is differentiating data flows in processes to achieve fine-grained provenance tracking on

[*] Co-primary authors.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560570



(a) A simplified provenance graph.

(b) Fine-grained provenance.

Figure 1: Watering-hole attack.

audit logs. However, collecting necessary runtime information of a process for data-flow separation can result in heavy performance overhead. A practical solution needs to incur a low overhead, require no program modification, and work with commodity systems.

In this work, we present PalanTír, the first attack investigation system that leverages *hardware-assisted processor tracing (PT)* to optimize attack provenance, which efficiently records a process's execution trace and effectively recovers instruction-level data flows. By combining system-call-level audit logs and instruction-level PT traces, PalanTír enables precise attack forensics with low overhead. Moreover, PT is a hardware feature enabled in modern CPUs [1, 5] that does not need any software instrumentation.

While PT can enhance system observability for attack provenance, several challenges exist in designing a practical provenance tracker. First, even though PT allows fully reconstructing program executions at the instruction level, optimization is needed to avoid the prohibitive overhead in the instruction-level taint analysis. Second, PT traces across processes are regularly interleaved, and their orders would vary based on dynamic program executions, making it difficult to correlate audit logs with PT traces.

To overcome the first challenge, our key intuition is to identify the tasks that can be pre-computed, reducing the computational complexity in instruction-level provenance tracking. Towards this end, PalanTír performs static analysis on application binaries to determine instructions that may taint system calls or be tainted by system calls. Therefore, PalanTír can selectively taint instructions in PT traces only pertaining to audit logs of system calls. To further improve efficiency, PalanTír pre-summarizes taint propagation logic for instructions at a coarser granularity of basic blocks. As for the second challenge, we develop a kernel module to intercept the PT hardware and associate execution traces with the corresponding process IDs (`PID`). To bridge the gap between PT traces and audit logs, PalanTír extracts `syscall` instructions from PT traces and sequentially aligns audit logs with them. Thereafter, by propagating taints among `syscall` instructions, PalanTír is capable of tagging audit logs with instruction-level provenance.

We implement PalanTír and evaluate its effectiveness in optimizing attack provenance on four real-world cyber-attacks simulated in a testbed environment. Additionally, we adopt 15 commonly-used Linux applications to investigate the scalability of PalanTír's static binary analysis. Experimental results show that PalanTír efficiently captures fine-grained provenance that reconstructs forensically accurate attack scenarios. Besides, PalanTír's static analysis scales to complicated applications compiled from upwards of 463,510 lines of C/C++ source code. We also demonstrate that PalanTír incurs an average runtime overhead of 4.5% on SPEC CPU 2006 benchmarks and 3.7% on popular Linux applications.

In summary, we make the following contributions:

- We propose a novel idea of optimizing attack provenance by enhancing system observability based on hardware-assisted processor tracing (PT).
- We design a scalable binary analyzer to statically identify instructions causally related to audit logs and summarize their taint propagation logic, and a practical provenance tracker that efficiently collects PT traces and performs selective taint analysis for fine-grained provenance tracking.
- We implement PalanTír and evaluate it against real-world cyber-attacks and commonly-used Linux applications. The results show that PalanTír achieves high effectiveness and efficiency in attack investigations and scales to complicated applications.

2 BACKGROUND & MOTIVATION

In this section, we illustrate the problem of *dependency explosion* in provenance-based attack investigation using a real-world cyber-attack, *watering-hole attack* [70]. Then, we present our insight into using hardware-enhanced observability to address the problem.

2.1 Running Example

Watering-hole attack is a widespread cyber-attack that targets large enterprises [2]. Instead of directly breaking into well-protected enterprise networks, it compromises less-secure websites frequently visited by the enterprise's employees.

Attack Scenario. Consider a server administrator in an enterprise who one day receives requests for software installation. To facilitate software management, the enterprise maintains an internal mirror that archives a list of sources for software packages. After going through the mirror, the administrator uses `wget` to download the requested software in bulk. Unfortunately, one of the sources has been compromised by an attacker, in which the original installation script `install.sh` is replaced with a malicious one. Without being aware of the attack, she then runs the malicious `install.sh` that copies `/home/admin/secret.txt` to the folder `/usr/share/nginx/html` hosted by Nginx. This folder serves as the content source of a public web server that handles daily network requests. Therefore, the attacker can access the sensitive file by visiting the server. To hide the attack footprint, `install.sh` will delete `/usr/share/nginx/html/secret.txt` after data exfiltration.

2.2 Attack Investigation

After gathering the latest cyber threat intelligence reports, the administrator discovers an indicator of compromise (connection to `z.z.z.z`) in the server, indicating forensic evidence of potential intrusions. To understand attack scopes, the administrator parses audit logs generated by a system auditing framework (Linux Audit [10]) into a provenance graph. By traversing the graph forward and backward, she can identify the root cause of the attack and its

Table 1: Comparison of solutions to dependency explosion.

Provenance System	Data Granularity	App Instru.	App Logging	Training Run
BEEP [55]	Unit	✓	✗	✓
ProTracer [65]	Unit	✓	✗	✓
MPI [64]	Task	✓	✗	✗
RAIN [46]	Instruction	✓	✗	✗
RTAG [47]	Instruction	✓	✗	✗
MCI [54]	Task	✗	✗	✓
UISCOPE [89]	Task	✗	✓	✗
OmegaLog [41]	Task	✗	✓	✗
ALchemist [90]	Task	✗	✓	✗
PalanTír	**Instruction**	✗	✗	✗

ramifications. Intuitively, provenance analysis serves to connect separate attack steps, reconstructing the overall attack scenario for investigation [52]. Figure 1(a) illustrates a simplified provenance graph resulting from the backward tracing based on `z.z.z.z`, in which nodes represent system entities, while rectangles, ovals, and diamonds denote processes, files, and sockets, respectively. Edges indicate dependencies (i.e., system calls) among system entities with the direction of information flow.

Dependency Explosion. Unfortunately, due to coarse-grained logging at the system-call level, provenance tracking on audit logs has to assume that the output of a process depends on all its preceding inputs [55], even though there exists no actual causality. This leads to the *dependency explosion* problem, especially for long-running processes that accumulate dependencies over time. For example, by backtracking `install.sh` in Figure 1(a), the administrator identifies 101 incoming network requests (e.g., `x.x.x.x` and `y.y.y.y`) in its ancestry, making it inconclusive where the malicious package comes. Even worse, the administrator will find hundreds of local files in the ancestry of `z.z.z.z`, making it inconclusive whether or which sensitive file is exfiltrated.

Limitations of Existing Solutions. To mitigate the dependency explosion, recent work strives to partition a process into finer-grained execution units [41, 55, 64]. For example, `nginx` can be decomposed based on iterations of its network request handling loops. Alternatively, MCI [54] trains a causal model based on audit logs with ground-truth provenance and adopts it to predict true causalities for incoming logs. Elsewhere in the literature, record-and-replay systems [26] have been extended to track fine-grained provenance — recording and replaying non-deterministic program executions and instrumenting programs to propagate taints from sources to sinks (i.e., system calls of audit logs) [46, 47].

However, we identify several drawbacks in existing solutions, which are summarized in Table 1. First, most approaches based on execution unit (or task) partitioning require expert knowledge to identify unit boundaries and mark them with program instrumentation. This indicates that software vendors must ship their applications with instrumentation, which unfortunately is not under consideration by any vender [68]. Recent studies propose to replace instrumentation with built-in application logs crafted by developers [41, 89, 90]. Unfortunately, the quality of application logs can be highly varied due to factors, e.g., the developer's subjective understanding of application runtime behaviors [58, 59]. More importantly, developers typically maintain application logs in a trial-and-error manner [57], limiting their reliability for security solutions. Second, while modeling-based approaches do not require instrumentation or application logs, they suffer from out-of-order audit logs generated by concurrent/cooperating applications. Finally, record-and-replay systems theoretically bring the most significant benefit because instruction-level provenance can be restored by performing taint analysis during application replays. However, they are not widely deployed due to high overhead in both time and storage. In addition, they also require instrumentation for recording non-deterministic inputs and tracking taint propagation.

2.3 Fine-grained System Provenance with Hardware Enhancement

Considering the limitations above, one ideal solution is to incorporate fine-grained provenance into system auditing without application instrumentation/logging. Towards this end, we take inspiration from the recent developments of hardware-assisted processor tracing (PT) that enables collecting transfers of control during program executions. Intuitively, PT forms the basis of non-intrusively recovering applications' execution history and tracking data flow at the instruction level. Thus, we propose to *enrich audit logs with instruction-level details from PT to resolve fine-grained provenance.*

By applying this intuition to our running example, we aim to generate the provenance graph in Figure 1(b). It concisely captures the ancestry (`wget` connecting to `y.y.y.y`) of `install.sh` and the progeny (`nginx` connecting to `z.z.z.z`) of `secret.txt`. Since substantial false causalities (e.g., `wget` connecting to `x.x.x.x`) are removed, we assist analysts in quickly understanding how the attacker gains access to the server and leaks the sensitive file.

While promising, there exist two major technical challenges: (1) efficient taint tracking using PT traces; (2) effective correlation of audit logs and PT traces. We solve these challenges through two design innovations. For efficient taint tracking (Section 4), we first statically analyze application binaries to identify instructions that possibly reach both input (e.g., `read`) and output system calls (e.g., `write`) at runtime. In this way, we can *selectively* propagate taints only for instructions of interest in which fine-grained provenance among audit logs will be resolved. Furthermore, we pre-summarize taint propagation logic for instructions at a coarser granularity (i.e., basic blocks in our design), so that taint analysis on PT traces can be pre-computed and simplified. Note that this paper uses taint propagation logic and tainting logic interchangeably. To correlate audit logs and PT traces (Section 5), we associate PT traces with the corresponding process IDs (`PID`) at runtime. Then, we recover process execution paths and identify the constituent `syscall` instructions. Thereafter, system-call-level logs and instruction-level traces per process can be aligned through system-call sequences, and taint propagation among `syscall` instructions tracks fine-grained provenance among audit logs of system calls.

In summary, the central idea behind our approach is leveraging hardware-assisted PT to restore instruction-level information flows aimed at enhancing system observability for attack provenance.

3 DESIGN OVERVIEW

3.1 Threat Model

In this work, we consider an attacker whose primary goal is to manipulate or exfiltrate sensitive information present in a system. To achieve this goal, the attacker may install malware, inject a backdoor, or exploit vulnerabilities of running applications. Similar to previous research on attack provenance reconstruction [39, 40, 43, 91, 92], we consider OS kernel as part of our trusted computing base. Hardware trojans and side-channel attacks are beyond our scope as they are invisible in system audit logs. We also assume that cyber-attacks are launched after our approach is deployed — system monitoring has started before the initial compromise. Although an attacker can escalate privileges to corrupt OS, we assume that he/she has no way of manipulating previous logs that have recorded the evidence of privilege escalation.

3.2 Palantír Architecture

Figure 2 presents a high-level overview of our approach, Palantír. It receives audit logs from Linux Audit [25] alongside PT traces from Intel PT [5] and generates observability-enhanced provenance graphs for attack investigation. Palantír consists of three main phases: static binary analysis (Section 4), runtime monitoring (Section 5), and attack provenance analysis (Section 6).

In static analysis, Palantír first lifts application binaries into intermediate representations and constructs their control flow graphs (*CFGs*) and call graphs (*CGs*). Then, it traverses the *CGs* to identify functions that potentially reach `syscall` instructions at runtime. To minimize the scope of taint analysis, Palantír further excludes the functions beyond the lowest common ancestors of `syscall` instructions. In this way, we safely remove instructions causally irrelevant to audit logs of system calls from the analysis. Finally, Palantír summarizes tainting logic per basic block in *CFGs* that is later queried for taint propagation. Our taint summarization is based on abstract interpretation [23], a widely used technique in static binary analysis [21, 51, 76], where concrete program states are subsumed into *abstract domains*, and program semantics are analyzed with *abstract semantics*.

At runtime, Palantír captures instructions executed by applications as PT traces and associates them with `PID` through a Linux kernel module. Simultaneously, Palantír also collects the corresponding audit logs produced by a kernel-space audit framework.

The key idea behind our attack investigation is to *use instruction-level PT traces to resolve fine-grained provenance for system-call-level audit logs so that coarse-grained dependencies are refined before causal analysis*. Specifically, Palantír recovers program execution paths from PT traces, identifies `syscall` instructions as taint sources and sinks, and selectively performs taint propagation by querying taint summaries derived from the static analysis. Once the taint propagation is done, Palantír correlates PT traces and audit logs through system-call sequences, constructs a whole-system provenance graph from audit logs, and tags dependencies with tainted `syscall` instructions. Afterward, given a symptom of an attack, analysts can precisely connect attack steps by traversing the fine-grained provenance graph.

4 STATIC BINARY ANALYSIS

Palantír's static analysis serves to profile application binaries. At a high level, Palantír traverses a binary to identify the scope of taint analysis and generate summaries of taint propagation logic per basic block. Specifically, Palantír adopts angr [78] as the front-end to lift an application's binary into VEX IR [77] and construct its *CFG* and *CG*. From the *CG*, Palantír first identifies functions causally related to system calls of audit logs to refine the *CFG* for taint summarization (Section 4.1). Then, it performs a context-, path-, and field-sensitive data flow analysis to summarize taint propagation logic at the granularity of the basic block (Section 4.2). Finally, Palantír stores taint summaries into an in-memory database (Redis [12]) for fast access.

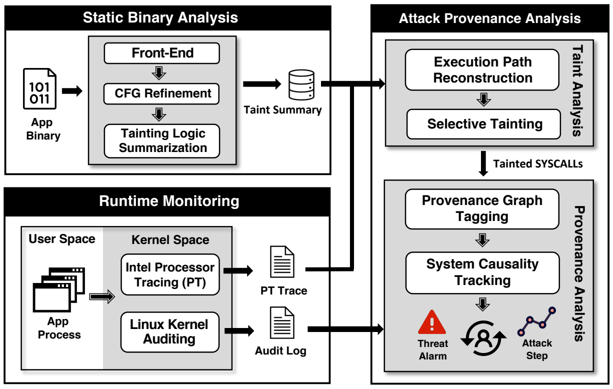


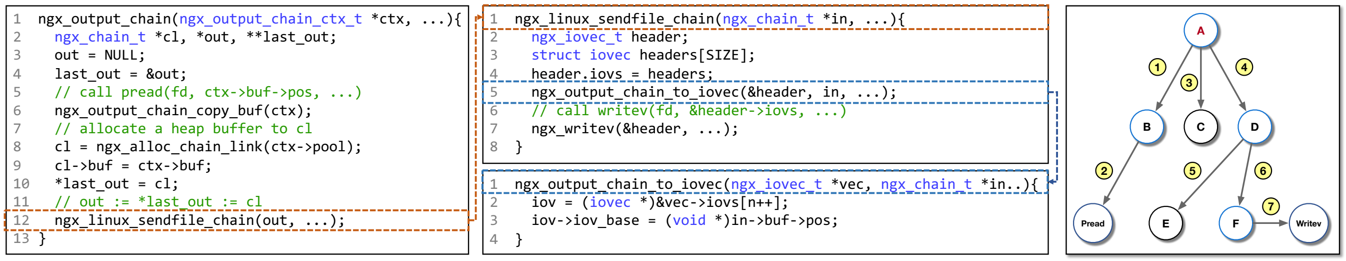
Figure 2: Overview of Palantír architecture.

However, static binary analysis commonly suffers from prohibitive overhead and false positives [35, 93], making it difficult to be both scalable and precise. In what follows, we describe how Palantír ameliorates its static analysis to achieve high scalability and precision. For clarity, we use source code to explain our analysis procedures, although the real analysis is based on binary. We perform the analysis at the binary level rather than the source level for two reasons: (1) PT records binary instructions executed by CPUs. Unfortunately, it requires non-trivial efforts to convert the analysis result from source code back to binary due to open problems, e.g., compiler optimization. (2) binary analysis is agnostic to programming languages and compilers and can be applied to proprietary software where source code is usually unavailable.

4.1 CFG Refinement

Palantír recovers fine-grained provenance by performing taint analysis on program execution traces collected by PT hardware. One naïve approach is to propagate taints on the full PT trace. However, the volume of a PT trace is typically large, imposing a high overhead in taint analysis. Chen et al. observe that only a fragment of a program contributes to taint analysis by introducing taint sources, propagating taints, or reaching taint sinks [20].

Inspired by this observation, we propose to refine the scope of taint analysis on a PT trace by filtering functions that do not affect taint propagation among system calls. To illustrate, consider tracking fine-grained provenance for a web server. Clearly, only the functions relevant to handling network and file operations should be included. The remaining functions (e.g., network scheduling) can be safely skipped without sacrificing provenance precision. Towards this end, Palantír seeks to traverse a program's *CG* to locate functions causally related to system calls of audit logs. We call such functions *audit-sensitive* ones. It is worth noticing that not all system calls of audit logs are of interest to provenance tracking (see [9] for details). For example, process initialization constantly loads read-only resources, but such resources are "dead ends" from



```
ngx_output_chain(ngx_output_chain_ctx_t *ctx, ...){
  ngx_chain_t *cl, *out, **last_out;
  out = NULL;
  last_out = &out;
  // call pread(fd, ctx->buf->pos, ...)
  ngx_output_chain_copy_buf(ctx);
  // allocate a heap buffer to cl
  cl = ngx_alloc_chain_link(ctx->pool);
  cl->buf = ctx->buf;
  *last_out = cl;
  // out := *last_out := cl
  ngx_linux_sendfile_chain(out, ...);
}
```

```
ngx_linux_sendfile_chain(ngx_chain_t *in, ...){
  ngx_iovec_t header;
  struct iovec headers[SIZE];
  header.iovs = headers;
  ngx_output_chain_to_iovec(&header, in, ...);
  // call writev(fd, &header->iovs, ...)
  ngx_writev(&header, ...);
}
```

```
ngx_output_chain_to_iovec(ngx_iovec_t *vec, ngx_chain_t *in..){
  iov = (iovec *)&vec->iovs[n++];
  iov->iov_base = (void *)in->buf->pos;
}
```

(a) Code Snippets

(b) Call Graph

Figure 3: A running example of taint logic summarization. (a) Simplified code snippets denote how `nginx-1.20.2` handles HTTP(S) requests. (b) Call Graph of the code snippets. A: *ngx_output_chain*; B: *ngx_output_chain_copy_buf*; C: *ngx_alloc_chain_link*; D: *ngx_linux_sendfile_chain*; E: *ngx_output_chain_to_iovec*; F: *ngx_writev*.

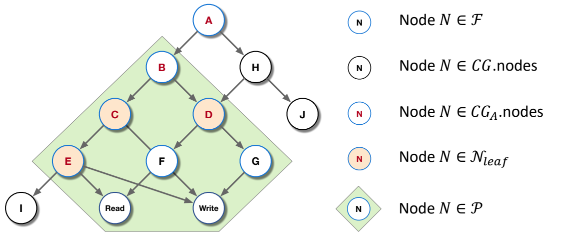

A
B
H
C
D
J
E
F
G
I
Read
Write
N Node $N \in \mathcal{F}$
N Node $N \in CG$.nodes
N Node $N \in CG_A$.nodes
N Node $N \in \mathcal{N}_{leaf}$
N Node $N \in \mathcal{P}$

Figure 4: A Running example for CFG Refinement.

the perspective of attack forensics [79]. Therefore, we filter them out before traversing the *CG*.

Given a *CG* from our front-end, PalanTír first transforms it to a directed acyclic graph by building a depth-first searching tree upon the *CG* and removing all back edges. After that, PalanTír computes two transitive closures for functions of input system calls (i.e., `read`, `recv`) and output system calls (i.e., `write`, `send`), respectively. Nodes in a transitive closure denote the functions, called syscall-reachable functions $\mathcal{F}$, that can arrive at either input or output system calls. An intuitive way to identify functions that emit system calls is searching for `syscall` instructions in a *CFG*. However, we make an observation that a program usually does not make direct system calls but utilizes standard libraries that provide equivalent functionalities. Therefore, PalanTír also extracts all function calls to `libc` and analyzes the possible system calls they can invoke. By further calculating the intersection of the two transitive closures, we obtain an *audit-sensitive call graph* CG_A, in which a node reaches both input and output system calls. Next, PalanTír extracts all leaf nodes $\mathcal{N}_{leaf}$ from CG_A and recognizes their lowest common ancestors as the starting functions of tainting scopes. Finally, we establish a tainting scope $\mathcal{P}$ with a starting function and its succeeding syscall-reachable functions $\mathcal{F}$. Figure 4 illustrates an example of the CFG refinement procedure, and we present its detailed implementation as an algorithm in [9].

4.2 Tainting Logic Summarization

Once CFG refinement is completed, PalanTír performs data flow analysis to generate taint summaries. Here, tainting logic can be summarized at different granularity: instruction, basic block, or function. As PT records control flow transfers at runtime, a PT trace can be viewed as a sequence of basic blocks. Therefore, we adopt the basic block as the primitive granularity for taint summarization so that PalanTír can seamlessly map taint summaries to PT traces for taint propagation. To explain the taint summary, we use the following x86-64 snippet (in Intel syntax) for illustration.

```
;input rbp:rbp0, rsi:rsi0, rcx:rcx0
    mov rbx, rbp                          T(rax) := T(rbp0) | T(rsi0)
    lea rax, [rbx + rsi]                  T(rsi) := T([rbp0 + 0x80])
    mov rsi, [rbp + 0x80]                 T(rbx) := T(rbp0) | T(rcx0)
    add rbx, rcx
```

$T(reg)$ denotes the taint tags of *reg*, and *reg*0 indicates the initial taint state of *reg*. Intuitively, a taint summary presents how taint propagates within a basic block based on its input state.

Before going into detail, we first introduce the challenges to achieving scalable and precise taint summarization as follows: (1) Identify memory alias for uninitialized variables. Since CFG refinement has removed functions beyond the scope of taint analysis, our summarization usually starts at an uninitialized state where global structures and/or function parameters are unknown; (2) Infer memory reference for nested structures. Data dependencies are likely nested in structure pointers, demanding data flow tracking across pointer-rich structures; (3) Reduce computational overhead. Static analysis is known to over-approximate program behaviors, leading to an explosion in the number of variable values during computation; (4) Track dependencies for out-of-scope functions. Tainting scopes only include functions that reach system calls in a *CG*, but out-of-scope functions may also contribute to taint analysis (e.g., propagating taint sources) among system calls.

To illustrate these challenges, we provide a running example from Nginx in Figure 3. Suppose function A (`ngx_output_chain`) is the starting point of a tainting scope, whose argument `ctx` is a structure pointer unknown to PalanTír. If PalanTír cannot identify the memory that `ctx` refers to, the data dependency will be lost when function A calls function B (`ngx_output_chain_copy_buf`). Additionally, since function B invokes a `pread` system call to store requested HTTP resources in `ctx->buf->pos`, PalanTír must be able to track the data flow within the nested pointer from `ctx` to `pos`. Otherwise, PalanTír misses the dependency from `pread`. Moreover, function C (`ngx_alloc_chain_link`) is beyond the scope of taint analysis because it cannot reach any system call. However, function C allocates a heap buffer to `cl` later used by a system call `writev` to send out data. Therefore, skipping this function in taint analysis will lose the dependency between `pread` and `writev`.

To address these challenges, our solution is to initialize unknown variables with symbolic values and enable multi-level pointer dereference on them in the analysis of nested structures. For out-of-scope

$$\begin{array}{lccl}
\text{Symbols} & \mathbb{Y} & = & \mathit{symbol} \mid c_1 * \mathbb{Y} \quad c_2 \mid [\mathbb{Y}] \ \ (c_1, c_2 \text{ are integers}) \\
\text{Abstract Variables} & \mathbb{I} & = & \mathbb{Y} \mid \top \mid \bot \\
\text{Abstract Regions} & \mathbb{A} & = & \mathsf{Heap}(\mathbb{Z}) \quad (\mathbb{Z} : \text{integer set}) \\
 & & \mid & \mathsf{Global}(\mathbb{Z}) \\
 & & \mid & \mathsf{Stack}(\mathbb{Z}) \\
 & & \mid & \mathsf{Symbol}(\mathbb{Z}) \\
\text{Taint Tags} & \mathbb{U} & = & T(\mathit{regs}) \mid T(\mathit{syscalls}) \mid T(\mathbb{A}) \\
\text{Abstract Values} & \mathbb{V} & = & \mathbb{I} \times 2^{\mathbb{A}} \times 2^{\mathbb{U}} \\
\text{Register Map} & \mathbb{R} & = & \mathit{regs} \rightarrow 2^{\mathbb{V}} \\
\text{Memory Map} & \mathbb{M} & = & \mathbb{A} \rightarrow 2^{\mathbb{V}} \\
\text{Execution Contexts} & \mathbb{C} & = & [cs_0, ..., cs_i] \mid ... \ \ (cs_i : \text{i-th call site}) \\
\text{Abstract State} & \mathbb{S} & = & \mathbb{R} \times \mathbb{M} \times \mathbb{C}
\end{array}$$

Figure 6: Abstract Domains.

functions, we track their data flow dependencies within a limited call depth to balance the trade-off between soundness and scalability. More importantly, we design PalanTír's static binary analysis upon abstract interpretation [23], which is developed to be flow-, context-, and field-sensitive.

4.2.1 *Abstract Domain*.

Figure 6 shows the abstract domains used in our static analysis. We present an abstract variable ($i \in \mathbb{I}$) as a *symbol* ($y \in \mathbb{Y}$) that denotes either a symbolic or concrete variable. Following the strategy used in *under-constrained symbolic execution* [29, 74, 75], we create symbolic variables for unknown function parameters and/or global structures. We also enable linear and dereference operations on them to generate new symbols. For example, the structure pointer `ctx` and its first member `ctx->buf->pos` in Figure 3 are presented as symbols ctx_s and $[[ctx\_s]]$ (i.e., $[[ctx\_s + 0x0] + 0x0]$), respectively.

Our abstract value ($v \in \mathbb{V}$) is composed of an abstract variable ($i \in \mathbb{I}$), a set of abstract regions ($A \subseteq \mathbb{A}$), and a set of taint tags ($U \subseteq \mathbb{U}$). Formally, we define it as a triple $v := \langle i, A, U \rangle$: First, the abstract variable i represents the variable that one register or memory cell contains. Note that, i can represent either a concrete variable, when $i = y$ ($y \in \mathbb{Y}, y = c_2$), or a symbolic one. Second, each abstract region ($a \in A$) represents the type of a memory cell (e.g., stack and heap) and its location that abstract value v can point to. Specifically, $\mathsf{Stack}(o)$ denotes a stack frame at the offset o, $\mathsf{Heap}(o)$ denotes a heap object at the offset o, $\mathsf{Global}(o)$ denotes a global cell at the address o, and $\mathsf{SymLoc}(y, o)$ denotes a symbolic memory address at the offset o of the symbolic base address y. Third, a taint tag ($u \in U$) represents the data flow dependency within abstract value v. Specifically, tag $T(reg \in regs)$ denotes v's data flow from the register reg, while $T(s \in syscalls)$ from the system call s and $T(a)$ from the abstract region a.

Recall that our static analysis summarizes taint propagation logic at the basic block granularity. Thus, each abstract state ($S \subseteq \mathbb{S}$) denotes a program state for a basic block under a specific execution context. S maintains a register map ($R \subseteq \mathbb{R}$) and a memory map ($M \subseteq \mathbb{M}$) that contain abstract values. The context ($C \subseteq \mathbb{C}$) describes the execution context during the static analysis. Formally, we define an abstract state as a triple $S := \langle R, M, C \rangle$. Note that PalanTír's static analysis inevitably over-approximates program behaviors, and thus each register or memory cell usually contains a set of valid/invalid abstract values. Similarly, an abstract value can include a set of valid/invalid abstract regions to point to.

$$\mathcal{G}(S, \mathsf{CONST}(c)) = \begin{cases} \{\langle c, \{\mathsf{Global}(c)\}, \phi \rangle\} & \text{if} \quad c \in \mathit{data\_section} \\ \{\langle c, \phi, \phi \rangle\} & \text{otherwise} \end{cases} \qquad (c \in \mathbb{Z})$$

$$\mathcal{G}(S, \mathsf{BinOP}(e_1, e_2)) = \mathsf{OP}(S, e_1, e_2) \quad \text{where } \mathsf{OP} \in \{\mathsf{ADD}, \mathsf{SUB}, ...\}$$

$$\mathcal{G}(S, \mathsf{GET}(reg)) = \begin{cases} S[0](reg) & \text{if} \quad reg \in S[0] \\ \mathit{AssignVals}(S, \mathit{ToSymbol}(reg), reg) & \text{otherwise} \end{cases}$$

$$\mathcal{G}(S, \mathsf{LOAD}(e)) = \bigcup_{v \in \mathcal{G}(S,e)} \{\mathit{LoadMem}(S, a) \mid a \in v[1]\}$$

$$\mathit{LoadMem}(S, a) = \begin{cases} S[1](a) & \text{if} \quad a \in S[1] \\ \mathit{AssignVals}(S, [\mathit{ToSymbol}(a)], a) & \text{otherwise} \end{cases}$$

$$\mathit{AssignVals}(S, y, t) = \{\langle y, \{\mathsf{SymLoc}(y, 0)\}, \{T(t)\} \rangle\}$$

(a) Evaluation of expressions: $\mathcal{G}(S,\ e) \rightarrow V$.

$$\mathcal{H}(S, \mathsf{PUT}(reg, e)) = S[0] \leftarrow S[0][reg \rightarrow \mathcal{G}(S, e)]$$

$$\mathcal{H}(S, \mathsf{STORE}(e_1, e_2)) = S[1] \leftarrow \mathit{Update}(S[1], \{v[1] \mid v \in \mathcal{G}(S, e_1)\}, \{v \mid v \in \mathcal{G}(S, e_2)\})$$

$$\mathit{Update}(M, A, V) = \begin{cases} M[a \xrightarrow{strong} V] & \text{if} \quad A = \{a\} \\ M[a_1 \xrightarrow{weak} V]...[a_n \xrightarrow{weak} V] & \text{if} \quad A = \{a_1, ..., a_n\} \end{cases}$$

$$\mathit{InitState}(S, R, M, C) = S \leftarrow \left\langle \begin{array}{l} R[r \rightarrow \bigcup\limits_{r \in \mathbb{R}} \bigcup\limits_{v \in R(r)} \{\langle v[0], v[1], \{T(r)\} \rangle\}], \\ M[a \rightarrow \bigcup\limits_{a \in \mathbb{M}} \bigcup\limits_{v \in M(a)} \{\langle v[0], v[1], \{T(a)\} \rangle\}], \\ C \end{array} \right\rangle$$

$$\mathcal{H}(\mathsf{Call}(S, f, cs)) = \mathit{Summarize}(\mathit{InitState}(S, S[0], S[1], S[2]), f, S[2] \circ cs) \ \ (cs\text{: callsite})$$

(b) Evaluation of statements: $\mathcal{H}(S,\ stmt)$.

Figure 7: Abstract Semantics.

4.2.2 *Abstract Semantics*.

Figure 7 shows our abstract semantics defined based on VEX IR. We use $X[k]$ to represent the (k)-th element of tuple X. For example, $v[1]$ of an abstract value v returns its abstract region set A.

First of all, we define $\mathcal{G}(S, e) \rightarrow V$ to evaluate an expression e given an abstract state S and return an abstract value set V, as shown in Figure 7a. For a constant expression $\mathsf{CONST}(c)$, its return V only contains a single abstract value v whose taint tag set $v[2]$ is empty. Therefore, evaluating a constant expression is simple: we check if the constant c is in the data section. If so, $v[1]$ is assigned as a set of one global address or otherwise an empty set.

Our design of evaluating register read (GET) and memory load (LOAD) expressions is the key to achieve multi-level pointer dereference for a symbolic memory address (SymLoc). After looking up the current S, we determine if the target register reg or memory cell a is in S's register or memory map, respectively. If so, we directly return the corresponding abstract value set in the map. Otherwise, we assign a new symbolic abstract value to the reg or a. For example, in Figure 3, suppose the parameter `ctx` of function A is passed by a register reg. When reading the reg, we assign a new abstract value v as it is uninitialized (not in S). More specifically, we first create a *symbol* using $ToSymbol(reg)$ to represent its abstract variable $v[0]$. For clarity, we call the created symbol ctx_s. Then, v's taint tag set $v[2]$ is assigned as $\{T(reg)\}$, denoting that its taint flow derives from reg. As `ctx` is a pointer, we set v's region set $v[1]$ as $\{\mathsf{SymLoc}(ctx\_s, 0)\}$. By doing so, $v[1]$ represents a symbolic region that supports dereference operations. For example, dereferencing `ctx->buf` (`ctx`'s first member) is to evaluate a memory load expression whose target memory cell is $\mathsf{SymLoc}(ctx\_s, 0)$.

By assigning symbolic abstract values, PalanTír can track data flow dependencies across nested pointers. To be scalable, we design a parameter N_{sym} to limit the maximum depth for dereferencing a symbolic pointer in memory loading. Notice that such a depth limit will cause unsound analysis of memory load expressions. Nonetheless, as demonstrated in our evaluation in Section 8.3, this limit does not sacrifice the precision of attack investigation, meaning that our unsound static analysis is forensically accurate.

Besides that, we define a *binary operation* as BinOP that performs arithmetic or logical operation (e.g., ADD, SUB) given two expressions e_1 and e_2. To evaluate $V = \mathcal{G}(S, \text{BinOP}(e_1, e_2))$, we first compute two abstract value sets $V_1 = \mathcal{G}(S, e_1)$ and $V_2 = \mathcal{G}(S, e_2)$ and then handle the semantics of the binary operation (see [9] for details). Note that the array index is ignored in BinOP when calculating abstract regions of returned abstract values. That is, all the array elements are merged into a single abstract region (i.e., array base). In this way, PalanTír is designed to be array-insensitive, a common practice in static binary analysis [21].

Next, we define $\mathcal{H}(S, stmt)$ to evaluate the statement $stmt$ given the input state S to update S, as shown in Figure 7b. The semantics of register write (PUT) is straightforward: PalanTír evaluates expression e given S to obtain the abstract value set V and then updates the target register in S with V. For memory write (STORE), PalanTír inputs two expressions e_1 and e_2 to evaluate the locations of target memories A and their abstract values V, respectively. To update the memories A in a memory map M with V, we deliberately distinguish between strong and weak updates [27] in Figure 7b.

As our analysis performs forward in a tainting scope, we first initialize each basic block's input abstract state with a join of its predecessors' output states. Then, we initialize taint tags of each abstract value in the input state. After evaluating every expression and statement in a block, we identify all registers and memory cells in the block's abstract state S and extract their taint tag sets U to formulate a taint summary. Later, we dump the taint summary of every block into an in-memory database with the concatenation of its execution context and address ($C \circ block\_addr$) as the key.

In particular, when our forward analysis encounters a function call, it determines whether the target function is within the scope of taint analysis. If so, we initialize an entry state for it and concatenate its callsite address cs and execution context C to start a new analysis procedure. After this analysis, we take a join of the states of all its exits (e.g., returns) as its output state (see [9] for details).

Since out-of-scope functions may also contribute to taint propagation, an idea is to force the analysis to step through every out-of-scope function to ensure the soundness of taint summarization. However, this design inevitably makes it hard to scale due to the bursts of execution contexts. To reside at a sweet spot between soundness and scalability, we present a configurable parameter N_{dep} to decide the maximum call depth for the taint summarization of out-of-scope functions. That is, PalanTír includes out-of-scope function calls within a limited depth for taint analysis.

Finally, to handle external functions (e.g., *libc* APIs), we follow the standard practice [87] to identify functions that invoke system calls (e.g., `fgets`) or allocate memories (e.g. `malloc`) and manually interpret their semantics for tainting logic summarization.

5 RUNTIME MONITORING

At runtime, PalanTír collects audit logs alongside PT traces. Here, we focus on Intel's implementation of PT, but our design can also be generalized to other PT technologies [1].

5.1 Intel Processor Tracing

To record a program's execution history, software-based instrumentation techniques (both static [28] and dynamic [17, 62] ones) suffer from high performance overhead and weak security properties [87]. Therefore, we center our design around Intel PT, a hardware feature that records program executions efficiently and securely.

After a program is loaded for execution, Intel PT generates a stream of packets to encode transfers of control. For example, *taken-not-taken* (TNT) packets log whether conditional branches are taken or not, and *target instruction pointer* (TIP) packets log target addresses of indirect transfers (e.g., indirect calls and returns). To decode PT packets into executed instructions, the decoder requires an additional memory layout of the running program. As such, we design a kernel module in PalanTír to first take the snapshot of a program's initial executable page. Then, we capture executable pages loaded into memory by intercepting related system calls (e.g., `mmap`). Thereafter, we obtain all the necessary information to completely reconstruct a process's execution path.

In order to distinguish PT packets across processes, we maintain execution traces for processes separately by hooking context switches. For example, when a `clone` system call is invoked to create a task (i.e., process or thread), PalanTír will allocate a new trace buffer to store its PT packets accordingly. Once a trace buffer becomes full, Intel PT raises a *non-maskable* Performance Monitoring Interrupt (PMI) that is immediately handled by our kernel module so that the trace buffer can be flushed without information loss. Notice that it is unnecessary to monitor every process using PT as not all processes lead to dependency explosion. As a result, we further hook the `execve` system call to allow analysts to attach PT only on processes of interest (e.g., `nginx`).

To minimize the performance cost, Intel PT writes packets directly to physical memory so that memory translation and CPU caches can be bypassed. For flexibility, we configure Intel PT to store packets in discontinuous memory spaces through the Table of Physical Address. Note that Intel PT can only be configured using the model specific register interface from Ring 0. For example, `MSR_IA32_RTIT_OUTPUT_BASE` specifies the base address of ToPA. This design prevents PT collection from being compromised by user-space processes.

5.2 Linux Kernel Auditing

Audit logs record system calls that a process has requested at runtime. To collect whole-system logs, we adopt Linux Audit, a standard kernel-space monitoring framework widely used in existing forensics systems [41, 42]. Compared to `syscall` instructions recorded in PT traces, audit logs provide additional system-call parameters and return values. Such auxiliary information forms the basis of building dependencies among system entities. For example, the file descriptor in `read` indicates from which opened file to read.

6 ATTACK PROVENANCE ANALYSIS

With taint summaries from static binary analysis and runtime information from Linux Audit and PT, PalanTír recovers instruction-level provenance relevant to audit logs of system calls and constructs a fine-grained provenance graph for attack investigation.

6.1 Taint Analysis

We perform taint analysis on PT traces to capture fine-grained (instruction-level) provenance. Towards this end, we first recover the execution path of a process based on its PT trace and memory layout. The key idea is to linearly disassemble the memory from its entry point and sequentially consult PT packets for non-deterministic branches. For runtime efficiency, Intel PT does not log any control transfers (e.g., direct calls) that have deterministic impacts on execution paths. While straightforward, this design inevitably repeats disassembling the same basic blocks in the memory due to repetitive program executions (e.g., loops). To avoid redundant computations, we implement a fast disassembly lookup technique from Griffin [33]. Specifically, PalanTír first allocates a heap data structure for every code block in the memory. While encountering a block in decoding PT packets, it looks up its data structure to retrieve the disassembled binary. If the block has not been disassembled, PalanTír then disassembles it and stores the result back in its heap data structure. By doing so, PalanTír never wastes time repetitively disassembling the same basic blocks.

In parallel to parsing PT packets, PalanTír identifies `syscall` instructions as sources and sinks for taint analysis. We also infer their system-call numbers by observing the constant integer moved into the `RAX` register. Since this observation mainly occurs within two basic blocks prior to `syscall` instructions, the inference is a trivial def-use analysis that can be done on-the-fly.

Given basic blocks recovered from PT traces, PalanTír first queries their taint propagation logic summarized in Section 4.2. Then, it performs selective tainting that starts from the source of input `syscall` (e.g., `read` and `recv`) and ends at the sink of output `syscall` (e.g., `write` and `send`). To enable taint propagation for multiple sources of system calls, we use a *set* to contain taint tags. Specifically, a new tag will be inserted into the set when an input `syscall` (e.g., `read`) is introduced. Following mainstream taint tools [45, 69], we maintain taint propagation status (aka taint state) in *maps* for registers, memory locations, and symbols.

Recall that a single basic block typically has multiple taint summaries under different execution contexts. To propagate taints accurately along with blocks in a PT trace, we must choose their taint summaries according to the underlying context. Towards this end, we use a graph structure to coordinate taint summaries. Each node in the graph stores a set of summaries under the same execution context, and edges present how the context changes through function calls and returns. While propagating taints on a PT trace, PalanTír traverses the graph by following the sequence of function invocations in the trace and retrieves taint summaries of basic blocks by querying their addresses. Meanwhile, PalanTír follows the tainting logic in summaries to update the tag values in the taint state. Notice that PalanTír may fail to locate a block in the graph, indicating that the current block is out of our tainting scopes, i.e., causally irrelevant to audit logs. Thus, PalanTír skips it for taint analysis. When the propagation reaches a sink (e.g., `write`), PalanTír collects the corresponding taint tag to identify where the taint originates, which captures fine-grained provenance.

6.2 Provenance Analysis

Given audit logs on end hosts, PalanTír first parses them into a graph structure called provenance graph [16, 34, 42, 64]. Nodes in the graph represent system entities with a set of attributes (e.g., `PID` and `INODE`) and edges annotated with the timestamp of occurrences describe system dependencies (i.e., system calls). Unfortunately, the graph, constructed from audit logs, is coarse-grained, leading to dependency ambiguity and explosion in attack investigation.

To refine the coarse-grained provenance graph, we aim to integrate the result of taint analysis on PT traces back to audit logs. However, a semantic gap exists between low-level PT traces and high-level audit logs. To bridge the gap, we use system calls as the connection. Specifically, PalanTír first sorts system-call-level logs for individual processes in chronological order. Then, it sequentially aligns these logs representing system calls with per-process `syscall` instructions decoded from PT traces. Afterward, given the result of taint propagation among `syscall` instructions, PalanTír tags the dependencies of system calls in provenance graphs with instruction-level data flows. In this way, when performing backward and forward tracing, PalanTír enables fine-grained provenance tracking on audit logs.

7 IMPLEMENTATION

We implement PalanTír in 10K lines of C/C++ code and 11K lines of Python code. To facilitate future research, we make PalanTír's source code and experimental data publicly available at [9]. Here, we present important technical details.

Binary Analysis. We develop PalanTír's binary analysis atop angr [78]. To resolve indirect calls in binaries, we adopt a type-based approach [71], to identify the number and type of parameters at the callsite and callee functions and then pair them by following the x64 calling convention. The indirect calls are further refined using TIP packets in PT traces that record target addresses of indirect control transfers. Following existing static analyzers [21, 66, 83], we perform loop unrolling to limit the number of loop iterations. Specifically, we iterate every loop twice. Despite sacrificing soundness, this design improves the scalability and precision of our static analysis. Additionally, as PalanTír may follow infeasible program execution paths in static analysis, we implement *opportunistic path sensitivity* from [93] to filter paths with conflicting constraints.

Provenance Collection. We collect whole-system audit logs using Linux Audit with a ruleset covering 32 types of system calls. To facilitate provenance analysis, PalanTír stores a provenance graph built upon audit logs into a local graph database (Neo4j [11]) for visualization. It is worth noticing that not all dependencies in a provenance graph are necessary for the causal analysis of cyber-attacks. To reduce "noisy" dependencies while preserving attack-relevant causalities, we take inspiration from [85] to aggregate dependencies with identical provenance scope.

PT Trace Parsing. We disassemble PT traces using an open-source disassembling library, distorm [7]. To support Transactional Synchronization Extensions (TSX) used in modern applications (e.g., Wget), PalanTír first uses Intel PT to log TSX events when a transaction begins, commits, or aborts. Then, it maintains PT traces to be disassembled for transactions separately. If a transaction aborts, PalanTír frees its trace or disassembles it otherwise.

8 EVALUATION

In this section, we evaluate PalanTír by answering the following research questions (RQs):

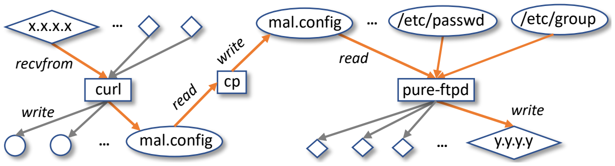


Figure 8: A simplified provenance graph of Data Leakage.

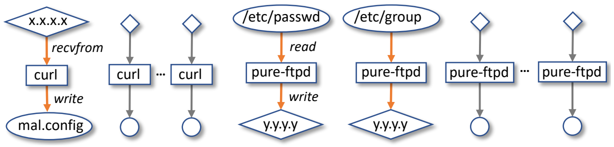


Figure 9: Fine-grained provenance of Data Leakage.

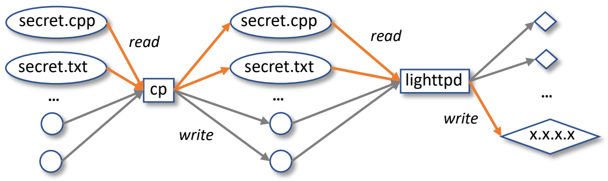


Figure 10: A simplified provenance graph of Insider threat.

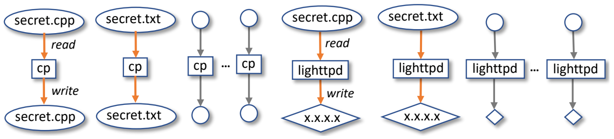


Figure 11: Fine-grained provenance of Insider Threat.

RQ1: How effective is PalanTír at attack investigation?
RQ2: What is the performance of PalanTír's static analysis?
RQ3: To what extent do different design choices in static analysis affect the effectiveness of attack investigation?
RQ4: What is the (runtime/storage/analysis) cost of PalanTír's optimization for attack provenance?

All the experiments are performed on a desktop with Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 16GB physical memory, and a 512GB hard disk drive. The OS is Ubuntu 16.04.6 64-bits LTS.

8.1 Attack Investigation

In this section, we evaluate PalanTír's effectiveness in attack investigation. To do so, we use four real-world attacks (watering-hole attack, data leakage, insider threat, and phishing email) simulated in a testbed environment. Note that the result of the watering-hole attack has been presented in Section 2. For the reason of space, we leave the description of the phishing email in [9].

8.1.1 ***Data Leakage.*** An administrator is asked to configure an FTP server (Pure-ftpd) that will be frequently used for file transfers within an enterprise. For reference, he has downloaded several configuration templates online using `curl`. After going through them, he decides to configure the server based on one of the templates (*mal.config*). However, this configuration lacks appropriate user authentication, allowing public access to arbitrary folders in the FTP server, even a folder that contains sensitive user information (e.g., `/etc/passwd` and `/etc/group`). One day, an adversary accidentally finds that he has permission to access the sensitive files via FTP and then downloads them for personal usage.

After several days, the administrator discovers that a user can read any file on the server. He then starts investigating whether sensitive files are leaked and what causes the misconfiguration of the FTP server. Figure 8 shows the resulting provenance graph generated by a traditional provenance tracking system [52]. It confirms that sensitive files (e.g., `/etc/passwd`) are accessed and the misconfiguration results from a file (`mal.config`) downloaded from unknown websites. However, due to the dependency ambiguity caused by `curl` and `pure-ftpd`, it is difficult for the traditional provenance tracker to determine if the sensitive data are transferred remotely and where the configuration file originates. In contrast, as shown in Figure 9, PalanTír identifies exactly which network sockets send `/etc/passwd` and retrieve `mal.config`, significantly accelerates the process of attack investigation. We also point out that no matter how many network sockets `curl` and `pure-ftpd` have connected, by enabling fine-grained provenance tracking, PalanTír only includes `y.y.y.y` in the progeny of `/etc/passwd` and `x.x.x.x` in the ancestry of `mal.config`.

8.1.2 ***Insider Threat.*** An engineer of a technology company is about to quit his job. Before leaving, he plans to steal in-progress projects from a workstation shared with his team. To exfiltrate the data, he first transfers (`cp`) all the project resources (e.g., `secret.cpp`) to a Lighttpd hosting folder. Afterward, he downloads these projects from another low-profile desktop by sending requests to `lighttpd`. Being aware of the deployed intrusion detection system, he also downloads numerous daily documents to disguise the data exfiltration as a regular software development operation.

The company notices that several undisclosed projects (e.g., `secret.cpp`) are posted online and asks its security team to initiate a forensic investigation. Using the traditional provenance tracker [52], the team generates a provenance graph as shown in Figure 10. Without PalanTír's assistance, an analyst would observe that `lighttpd` extensively interacts with local files, including `secret.cpp`, but never knows whether or to which IP address they are leaked. More importantly, the analyst cannot diagnose if any other files are exposed to the same IP address. Figure 11 shows attack causalities tracked by PalanTír. Because PalanTír successfully uncovers the fine-grained provenance for `cp` and `lighttpd`, the analyst can quickly trace the dependencies from `secret.cpp` to `x.x.x.x`, which demonstrates how the insider exfiltrates data. By further performing backward tracking on `x.x.x.x`, the analyst can discover the remaining leaked files (e.g., `secret.txt`). This provenance analysis provided by PalanTír ensures that the company is aware of attack scopes and stops unnecessary worrying.

8.2 Static Analysis Performance

We evaluate PalanTír's static analysis using 15 popular Linux applications, most of which have been widely used in recent work to evaluate attack provenance [41, 54, 64, 87]. The experimental results are summarized in Table 2.

Recall that CFG refinement identifies and eliminates CFG nodes beyond the scope of taint summarization. Despite the cause of unsound analysis, it decreases the size of tainting scopes by 85% on average, substantially reducing the overhead of downstream

Table 2: Results of PalanTír's static binary analysis. Refined CFG shows the size and percentage of our tainting scopes refined from Static CFG. Taint Summary shows the number of basic blocks to be summarized in different execution contexts and their taint (data) flows. CFG-R, TLS, and TSC denote CFG refinement, tainting logic summarization, and taint summary coordination. Memory and Storage present the peak memory consumption during analysis and storage cost of taint summaries.

Program	Binary Size (KB)	Static CFG		Refined CFG		Taint Summary		Time Cost (s)			Memory (MB)	Storage (MB)
		Nodes	Edges	Nodes	Edges	Blocks	Flows	CFG-R	TLS	TSC		
HAProxy	8,395	69,531	116,803	1,783 (2.6%)	2,750 (2.4%)	2,415	4,496	37.8	47.9	0.8	1184.6	3.5
Varnishd	6,066	45,650	93,163	1,322 (2.9%)	2,342 (2.5%)	21,908	75,502	60.1	809.8	9.5	968.6	46.6
Nginx	5,013	30,709	55,691	2,338 (7.6%)	3,795 (6.8%)	7,294	28,890	16.4	506.7	3.5	4246.9	14.3
Ntpd	4,164	31,325	59,421	4,194 (13.4%)	6,497 (10.9%)	12,930	38,993	18.6	1306.6	6.9	7923.5	24.4
Sendmail	3,965	39,204	77,075	1,911 (4.9%)	3,132 (4.1%)	6,219	26,889	22.1	859.2	1.8	3449.4	10.8
Proftpd	3,510	41,140	82,877	1,275 (3.1%)	2,027 (2.4%)	16,265	61,066	24.5	1942.1	10.2	7799.9	38.8
Curl	3,146	20,160	34,981	4,680 (23.2%)	7,702 (22.0%)	16,337	44,938	15.1	1294.0	3.7	3092.8	25.8
Wget	2,234	22,349	41,044	2,186 (9.8%)	3,526 (8.6%)	14,291	51,439	13.6	1566.9	7.7	4669.9	26.5
Httpd	1,214	39,205	67,983	2,920 (7.4%)	4,742 (7.0%)	39,111	119,754	25.6	2652.4	20.4	2599.8	88.0
Lighttpd	1,495	14,453	25,639	156 (1.1%)	220 (0.9%)	235	642	8.9	2.5	0.1	431.8	1.2
Cupsd	1,078	19,294	39,655	2,200 (11.4%)	3,576 (9.0%)	7,508	20,021	9.6	518.7	2.4	4517.7	12.4
Pure-ftpd	513	6,198	10,750	3,367 (54.3%)	5,713 (53.1%)	26,056	88,195	4.0	1298.2	10.6	2925.4	43.9
Cp	360	5,357	9,552	961 (17.9%)	1,522 (15.9%)	1,541	4,537	3.2	76.1	0.1	828.5	3.1
Zip	220	9,155	16,831	553 (6.0%)	894 (5.3%)	20,748	54,552	4.8	3295.5	10.7	9525.2	41.3
Thttpd	105	4,021	7,249	2,409 (59.9%)	4,111 (56.7%)	5,890	22,107	2.2	295.0	2.6	848.8	10.7

tainting logic summarization. As shown in Table 2, the time cost for CFG refinement is stable, always within a minute. In contrast, the overhead for tainting logic summarization is highly varied, ranging from two seconds (for Lighttpd) to an hour (for Zip).

At first glance, it seems counterintuitive that no relationship exists between the binary size and the overhead of taint summarization. For example, the summarization of HAProxy (8395KB) and Lighttpd (1495KB) requires less than a minute, while that of Zip (220KB) takes almost an hour. To gain further insights, we investigate the source code of these applications and discover that PalanTír's performance is affected by two main factors.

First, different ways of invoking system calls in applications significantly affect the size of tainting scopes determined by CFG refinement. Typically, the closer system calls are in a CFG, the smaller the tainting scope will be. The reason is that PalanTír defines the scope of taint summarization by calculating the lowest common ancestors of system calls. For example, in Lighttpd, all the invocations to `read` and `write` system calls appear closely in its CG. As a result, Table 2 shows that PalanTír includes only 1.1% of Lighttpd's CFG nodes in its tainting scope and skips the rest for taint summarization. Second, since PalanTír's taint summarization is context-sensitive, its analysis overhead naturally increases with the growing complexity of execution contexts. For applications (e.g., Zip) with denser CGs, PalanTír usually generates more execution contexts, resulting in higher overhead to summarize individual blocks under diverse contexts.

As a further investigation, we present the source code analysis of Lighttpd, HAProxy, and Zip. Detailed descriptions of HAProxy and Zip are summarized in [9]. The following code snippet of Lighttpd represents the only tainting scope identified by our CFG refinement. This code handles a network request by reading local resources and writing the response to a remote network socket. As the `read` and `write` system calls appear together and both of them are called by the same function *network_write_file_chunk_no_mmap*, CFG refinement only forms a small tainting scope for them.

It is worth noticing that PalanTír's static analysis is a one-time effort for an application. That is, once PalanTír has profiled an application's binary, there is no need to re-analyze it until the application gets updated.

```
  /* lighttpd 1.4.53 */
  static int network_write_file_chunk_no_mmap(server *srv, int fd,
         chunkqueue *cq, off_t *p_max_bytes) {
    chunk* const c = cq->first;
    ...
    if (0 != chunkqueue_open_file_chunk(srv, cq)) return -1;
    ...
    /* READ: fill server's tmp buffer with requested file */
    if (-1 == (toSend = read(c->file.fd, srv->tmp_buf->ptr, toSend))) {
       log_error_write(srv, __FILE__, __LINE__,
              "ss","read:",strerror(errno));
       return -1;
    }
    /* WRITE: use server's tmp buffer to send response */
    wr = network_write_data_len(fd, srv->tmp_buf->ptr, toSend);
    ...
}
```

8.3 Design of Static Analysis

As static binary analysis plays an important role in PalanTír, we investigate how its different designs may affect precision and scalability. Specifically, as PalanTír defines N_{sym} and N_{dep} to limit the upper bound of symbolic pointer dereference depth and out-of-scope function call depth, we evaluate how PalanTír performs in attack investigation with different values of these two parameters. We report the results in Table 4. The rows under **Min. Effective Parameter** indicate that our static analysis achieves accurate forensic investigation if both N_{sym} and N_{dep} are set above two.

To further study how these parameters affect PalanTír's scalability, we change each parameter's value from zero to five while fixing the other as three. The result shows that N_{sym} only noticeably affects seven of the 15 applications in Table 2. This is because

Table 3: Storage cost of the PT traces and audit logs for the programs used in the attack scenarios and PalanTír's performance overhead of attack provenance optimization. EPR, TSR, ST, and PGT denote execution path reconstruction, taint summary retrieval, selective tainting, and provenance graph tagging, respectively.

Attack Scenario	Program	Statistics				Storage (MB)		Time Cost (s)			
		Audit Logs	PT Packages	Blocks	Instructions	Audit	PT	EPR	TSR	ST	PGT
Watering Hole	Wget	10,265	62,175,669	192,176,614	1,329,321,333	6.4	138	1.095	10.866	0.087	0.160
	Nginx	1,830	401,708	1,103,725	5,160,695	1.3	23	0.023	2.815	0.026	0.034
Data Leakage	Curl	10,309	1,882,471	3,758,138	17,516,456	6.3	22	0.043	9.203	0.142	0.225
	Pure-ftpd	25,562	21,402,396	107,684,839	2,833,740,916	17	45	0.460	2.086	0.300	0.437
Insider Threat	Cp	1,814	134,161	203,802	1,048,907	1.3	8.5	0.009	0.188	0.006	0.035
	Lighttpd	4,800	499,995	1,248,778	5,448,715	3.2	18	0.026	0.548	0.008	0.086
Phishing	Sendmail	29,433	7,588,895	29,160,413	120,264,352	20	125	0.213	17.518	0.360	0.411

the remaining eight applications (e.g., Cp) are less likely to use deep nested structures or pointers within the scope of taint analysis. On the contrary, N_{dep} affects almost all the applications. This is expected as a higher upper bound of the call depth in out-of-scope functions leads to more complex execution contexts and thus more taint summaries. In conclusion, our evaluation justifies that PalanTír has balanced the trade-off between the precision of attack investigations and the scalability of static analysis.

Table 4: Results of PalanTír's static analysis in different designs. Rows under Min. Effective Parameter indicate the minimal parameter value needed for accurate attack investigation. Rows under Parameter Selection present how PalanTír's performance changes with different parameter values.

Min. Effective Parameter	Watering-hole		Data Leakage		Insider Threat		Phishing Email
	Wget	Nginx	Curl	Pure-ftpd	Cp	Lighttpd	Sendmail
N_{sym}	0	2	2	1	1	2	2
N_{dep}	0	1	0	1	0	0	2

Parameter Sel. (0-5)	Affected Programs	Incr. Time(%) Min/Med/Max	Incr. Memory(%) Min/Med/Max	#Incr. Taint Flows Min/Med/Max
N_{sym}	7/15	6.5%/9.1%/47.3%	3.5%/8.3%/22.4%	4.2%/11.8%/395.3%
N_{dep}	14/15	65.9%/668.7%/8109.2%	2%/122.2%/756.1%	25.0%/250.8%/6885.8%

8.4 System Performance

Analysis performance. To study PalanTír's performance, we measure its time duration at individual analysis phases, including execution path reconstruction (i.e., parsing PT traces), taint summary retrieval, selective tainting, and provenance graph tagging (i.e., aligning PT traces with audit logs). All experiments are performed five times, and we report the mean results in Table 3.

PalanTír's analysis overhead generally increases along with the size of audit logs and PT traces. Upon closer investigation, we find that most overhead comes from the taint summary retrieval. This is expected since a single program can produce over ten thousand taint summaries, and PalanTír has to search for the corresponding summary for every basic block executed by the program (e.g., over 100 million blocks by Pure-ftpd). From Table 3, we also observe that PalanTír consistently spends less than 0.5 seconds on selective tainting. At first glance, this result seems unreasonable as the taint analysis is known to be time-consuming. However, recall that PalanTír only propagates taints for the blocks causally relevant to system calls of audit logs, which reduces the scope and thus the overhead of taint analysis. More importantly, the complexity of taint propagation has been simplified by static taint summarization, which costs PalanTír fewer computations in taint analysis on PT traces. Note that PalanTír is currently designed using a single thread, but its performance can be further improved using parallel PT trace/audit log processing techniques [33, 45, 92].

Storage cost. We evaluate PalanTír's storage cost with the attack scenarios used in Section 8.1. Following the common practice in recent security solutions [24, 33, 44, 46], we design PalanTír to collect and store audit logs and PT traces on local hosts. As such, PalanTír's storage is measured with the at-rest sizes of collected data on disk, which are summarized in Table 3. Here, for ease of illustration, we only report the storage of audit logs for the programs that require instruction-level provenance tracking, but note that PalanTír, in effect, performs system-wide audit logging. To better justify PalanTír's practicality, we also conduct whole-system PT tracing (excluding the OS) on a server for seven days. The server is used by a lab student daily for development, research, and administration (e.g., hosting personal websites). In our experiment, we observe that PalanTír generates 98.4GB–111.6GB PT data per day. We would like to point out that the storage of PT traces is temporary. Once the taint analysis on the traces is completed, they can be discarded to free space. Eventually, PalanTír only stores audit logs with hardware-enhanced observability for attack investigation. We further evaluate PalanTír' storage cost on long-running services by stressing web servers with a high frequency of client requests. The results are summarized in [9].

Runtime overhead. We measure PalanTír's runtime overhead using SPEC CPU 2006 benchmarks and real-world programs from Table 2. The measurement is performed by comparing PalanTír to the baseline of Linux Audit with Intel PT disabled. Specifically, we run the SPEC benchmarks[1] with the standard "reference" workloads. As for the real-world programs, we adopt either their official benchmarks, if available, or daily usages (explained in [9]). In particular, the SPEC benchmarks indicate CPU-bound cases, while the programs from Table 2 present IO-bound cases.

As shown in Figure 12, PalanTír introduces an average overhead of 4.5% on the SPEC benchmarks and 3.7% on the real-world programs. Our experiments demonstrate that PalanTír only incurs negligible performance slowdown to enable fine-grained provenance tracking. An interesting observation is that the overhead for different programs can be highly varied, ranging from 1% to 28%,

[1] PalanTír is not evaluated on all the SPEC programs because some of them (e.g., `481.wrf`) cannot be compiled on our system [33].

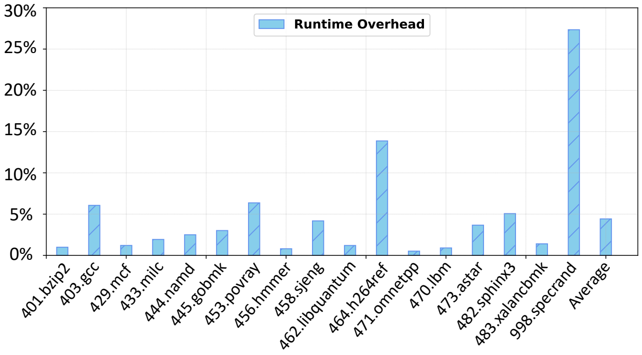


Figure 12: Runtime overhead on the SPEC CPU 2006.

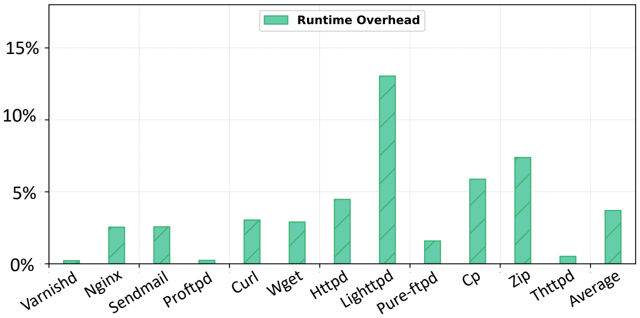


Figure 13: Runtime overhead on the programs from Table 2.

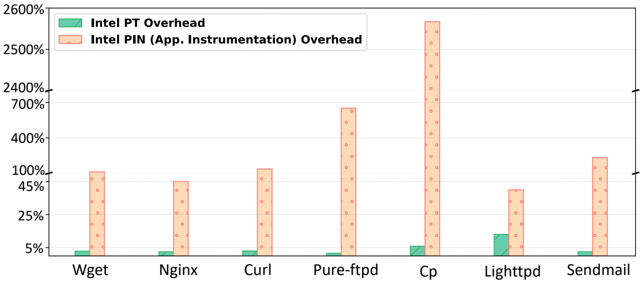


Figure 14: Comparison of runtime overhead between Intel PT and app instrumentation using Intel PIN.

which is consistent with previous work [33, 86, 87]. The noticeably higher overhead results from the programs that have more frequent control transfers and thus force Intel PT to generate more packets. We also observe that PalanTír usually incurs higher runtime overhead on the SPEC benchmarks than our real-world programs. The reason is that most of the SPEC benchmarks are CPU-intensive, where PalanTír never stops PT tracing. However, for I/O-intensive programs (e.g., web and FTP servers), when they wait for I/O, PalanTír does not impose any overhead by Intel PT.

8.5 Empirical Comparison

Processor tracing (PT) vs. App Instrumentation. To understand the advantage of hardware tracing, we compare it with app instrumentation, particularly Intel PIN [6]. At PIN's core is a just-in-time compiler that enables inserting arbitrary code (e.g., for taint analysis) into a running program. For comparison, we evaluate the runtime overhead (i.e., slowdown to native app execution) of PalanTír and nullpin [50] (an implementation widely adopted to test PIN's performance). Figure 14 illustrates the results when running the programs in Table 3 under PT and app instrumentation. We observe that instrumentation imposes a slowdown ranging from 40% to 2567%, whereas PT significantly outperforms instrumentation with 3x–436x faster with a slowdown from 2% to 13%. It is also worth noticing that the overhead of nullpin exclusively comes from app instrumentation. To support control flow tracing like PT, it introduces an additional slowdown.

PalanTír vs. Rtag. Rtag [47] is the state-of-the-art system that supports instruction-level provenance tracking on audit logs. By using a record-and-replay technology [26], it enables dynamic information flow tracking (DIFT) on the replay of program executions to identify fine-grained causalities. For a fair comparison, we assign the same provenance tracking tasks to PalanTír and the DIFT engine[2] used by Rtag [50]. In Table 5, we report their results of forensic investigations on four attack scenarios. Here, we measure PalanTír's time cost with the duration of taint analysis on PT traces (from parsing PT packets to taint propagation). Thanks to our static taint summarization, PalanTír spends 77% to 96% less time than Rtag on the provenance tracking. Notice that by building atop taint analysis (or DIFT), both PalanTír and Rtag are inevitably affected by potential under-tainting and over-tainting problems. However, we surprisingly find that both approaches achieve zero false positive/negative in our attack forensics — i.e., the attack causalities successfully match the ground truth (e.g., where a sensitive file is sent to). In Section 9, we further discuss how PalanTír could mitigate the under- and over-tainting problems.

Table 5: Comparison between PalanTír and Rtag [47] on fine-grained provenance tracking. #FP and #FN denote the number of false-positive and false-negative roots/ramifications in attack investigation, respectively.

Attack Scenario	Program	Time Cost (s)		#FP / #FN	
		Rtag	PalanTír	Rtag	PalanTír
Watering Hole	Wget	67.93	12.05	0 / 0	0 / 0
	Nginx	37.50	2.86	0 / 0	0 / 0
Data Leakage	Curl	50.03	9.39	0 / 0	0 / 0
	Pure-ftpd	78.16	2.85	0 / 0	0 / 0
Insider Threat	Cp	0.89	0.20	0 / 0	0 / 0
	Lighttpd	12.13	0.58	0 / 0	0 / 0
Phishing	Sendmail	238.2	18.09	0 / 0	0 / 0

9 LIMITATIONS & DISCUSSIONS

Comparison with value-set analysis (VSA). VSA [14, 15] is a powerful static binary analysis technique that leverages *interval domains* to calculate abstract locations. Specifically, VSA separates abstract locations into three disjoint regions: Stack, Heap, and Global. To guarantee soundness, it performs an over-approximation in tracking memory accesses of abstract locations. However, this design usually results in imprecise memory alias analysis in real-world applications [20], especially for uninitialized program states and/or pointer-rich data structures. For example, in the analysis of [[rbx + 0x8] + 0x10], VSA returns the memory access of $[-\infty, +\infty]$,

[2] We reproduce an extension of libdft [18] to support evaluation on x86_64 architecture.

assuming that it can refer to a memory region of an arbitrary offset. Different from VSA, PalanTír uses *symbolic domains* to determine memory accesses. In this way, PalanTír always tracks concrete offsets of abstract locations using symbols, even for unknown inputs, and thus achieves better precision for resolving memory alias.

Threat to Validity. There are four main threats to the validity. First, our static analysis is designed to trade off soundness for scalability and precision, e.g., by limiting the call depth for out-of-scope functions. Although PalanTír proves to be forensically accurate in our evaluation, its unsoundness potentially causes false-negative causalities in attack investigation. To improve soundness, one possible strategy is to adopt the bottom-up style analysis [19, 21] to further summarize out-of-scope functions that exceed the limit of the call depth. We leave this extension to future work. Second, PalanTír relies on taint analysis and thus inherits the over-tainting and under-tainting problems that cause false-positive and false-negative attack causalities. Technically, these problems are consequences of inaccurate taint policies — e.g., whether to propagate taints for variables in a conditional branch — which are open research problems [22, 45, 69] and orthogonal to PalanTír. That said, modularity is one of our guiding principles in the design of PalanTír, where its static analysis can be easily adjusted according to different taint policies. Finally, the current PalanTír does not generate taint summaries for dynamically generated (i.e., jitted) code. While PalanTír is capable of capturing jitted code by monitoring all the executable pages loaded into memory, it requires non-trivial efforts to further enable taint analysis [33]. The key challenge is that jitted code usually changes over time while PalanTír summarizes taint propagation logic statically. At last, PalanTír cannot handle GUI-based applications (e.g., Vim) as they have overly non-deterministic program states that depend on user inputs.

10 RELATED WORK

System Auditing. System auditing is a fundamental monitoring capability with a broad spectrum of security practices, such as attack scenario reconstruction [13, 42, 73], intrusion detection [37, 81, 92], alert triage [39, 40], and network troubleshooting [80, 94]. Backtracker [52] is the first to represent audit logs into a dependency graph (aka provenance graph) for attack forensics. To accelerate the investigation process, PrioTracker [61] prioritizes the provenance tracking of abnormal (or rare) dependencies. Nodoze [40] makes an improvement by prioritizing dependency paths rather than individual dependencies based on rareness. However, the resulting provenance graph is usually very large due to the overwhelming volume of audit logs. Even worse, given limited analysis time, malicious activities may be crowded out in the noise of benign logs. In order to scale up provenance-based investigation, recent studies strive to eliminate irrelevant activities from audit logs [38, 56, 79] and increase the efficiency of log queries [30, 32, 67]. Another line of research aims for a higher-level summary of malicious activities. Holmes [68] abstracts audit logs into a high-level scenario graph (HSG) by matching system activities against a knowledge base of Tactics, Techniques, and Procedures (TTPs) and connecting them with information flow. Similarly, RapSheet [39] identifies TTPs from audit logs to construct a tactical provenance graph (TPG) that conforms to kill chains of advanced persistent attacks (APT) defined by MITRE. Dependency explosion is another important drawback that hinders provenance analysis. There exists a rich literature to address this problem, which can be roughly categorized into three directions: execution-unit partitioning [41, 55, 64], model-based inference [54], and taint tracking [46, 47]. Nonetheless, these techniques require manual assistance to perform application instrumentation or logging, which are either not acceptable or practical in deployment. PalanTír provides the first attack investigation system that combines system-call-level audit logs and instruction-level PT traces without any modifications to applications.

Program Diagnosis using PT. Processor tracing (PT) has recently attracted increasing attention due to its high efficiency in recording program executions. Extensive literature exists on the use of PT for program diagnoses, such as root cause analysis [24, 48, 49, 84, 88], bug reproduction [95], and bug hunting [86]. Gist [49] presents the first failure sketching system that leverages PT to identify race conditions. Lazy Diagnosis [48] relies on PT to track thread interleavings and detect concurrency bugs. POMP [84] and REPT [24] perform reverse debugging of program failures by tracking information flow on PT traces. ARCUS [88] further helps interpret the root causes of failures by answering "what if" questions via symbolic execution. BunkerBuster [86] also conducts symbolic execution on PT traces but for the purpose of bug hunting. PT-assisted control-flow integrity (CFI) enforcement is another popular research area. While FlowGuard [60] and PT-CFI [36] leverage PT to accelerate the detection of CFI violations, Griffin [33] and μCFI [44] adopts it to enforce finer-grained CFI policies. MARSARA [87] demonstrates the first CFI violation attack in audit log analysis and presents a new defense to validate the integrity of audit logs based on PT. PalanTír presents the first attack investigation system that incorporates audit logs with PT traces to reason about fine-grained provenance and solves the dependency explosion problem in provenance tracking.

11 CONCLUSION

In this paper, we present an observability-enhanced attack investigation system, PalanTír. It takes the first step to bringing the benefits of hardware tracing to attack provenance reconstruction. By incorporating hardware-assisted processor tracing (PT), PalanTír achieves efficient program execution monitoring. Then, it selectively propagates taints based on PT traces to resolve instruction-level provenance. PalanTír also leverages static binary analysis to generate taint summaries to accelerate the taint analysis. In our evaluation against real-world cyber-attacks, PalanTír shows its advantage in supporting efficient and effective provenance analysis.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful feedback in finalizing this paper. We also thank Zheng Leong Chua, Kaihang Ji, Jiahao Liu, Jianing Wang, and Yuancheng Jiang for their valuable discussions. This research is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] 2011. ARM Embedded Trace Macrocell. https://developer.arm.com/documentation/ihi0014/q/Introduction. Online; Accessed 29 March 2022.

[2] 2015. Hacker group that hit Twitter, Facebook, Apple and Microsoft intensifies attacks. https://www.computerworld.com/article/2945652/hacker-group-that-hit-twitter-facebook-apple-and-microsoft-intensifies-attacks.html. Online; Accessed 1 April 2022.

[3] 2019. Equifax Information Leakage. https://en.wikipedia.org/wiki/Equifax. Online; Accessed 9 March 2021.

[4] 2020. Twitter hack. https://www.theguardian.com/technology/2020/jul/15/twitter-elon-musk-joe-biden-hacked-bitcoin. Online; Accessed 25 March 2020.

[5] 2021. Intel Processor Trace. https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html. Online; Accessed 29 March 2022.

[6] 2021. Pin: A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html. Online; Accessed 1 August 2022.

[7] 2021. Powerful Disassembler Library For x86/AMD64. https://github.com/gdabah/distorm. Online; Accessed 6 April 2022.

[8] 2021. SolarWinds: How Russian spies hacked the Justice, State, Treasury, Energy and Commerce Departments. https://www.cbsnews.com/news/solarwinds-hack-russia-cyberattack-60-minutes-2021-02-14/. Online; Accessed 17 August 2021.

[9] 2022. Artifact Release and Appendix. https://github.com/Icegrave0391/Palantir.

[10] 2022. Linux Kernel Audit Subsystem. https://github.com/linux-audit/audit-kernel. Online; Accessed 10 March 2021.

[11] 2022. Neo4j Graph Database. https://neo4j.com. Online; Accessed 6 April 2022.

[12] 2022. Redis. https://redis.io. Online; Accessed 6 April 2022.

[13] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z Berkay Celik, Xiangyu Zhang, and Dongyan Xu. 2021. ATLAS: A Sequence-based Learning Approach for Attack Investigation. In *USENIX Security Symposium (USENIX)*.

[14] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/X86—A Platform for Analyzing X86 Executables. In *International Conference on Compiler Construction (CC)*.

[15] Gogul Balakrishnan and T. Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *International Conference on Compiler Construction (CC)*.

[16] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. 2015. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security Symposium (USENIX)*.

[17] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO)*.

[18] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)*.

[19] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. 2015. Static Detection of Packet Injection Vulnerabilities: A Case for Identifying Attacker-Controlled Implicit Information Leaks. In *ACM Conference on Computer and Communications Security (CCS)*.

[20] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting. In *USENIX Security Symposium (USENIX)*.

[21] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*.

[22] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics.. In *the Symposium on Network and Distributed System Security (NDSS)*.

[23] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*.

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

[25] Audit Daemon. 2021. Linux Audit Daemon. https://github.com/linux-audit/audit-userspace. Online; Accessed 12 March 2021.

[26] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. 2014. Eidetic systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[27] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Fluid Updates: Beyond Strong vs. Weak Updates. In *ACM European Conference on Programming Languages and Systems (ESOP)*.

[28] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary Rewriting without Control Flow Recovery. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.

[29] Dawson Engler and Daniel Dunbar. 2007. Under-constrained execution: Making automatic code destruction easy and scalable. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[30] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. 2021. SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression. In *USENIX Security Symposium (USENIX)*.

[31] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In *International Conference on Software Engineering (ICSE)*.

[32] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. 2018. SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *USENIX Security Symposium (USENIX)*.

[33] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. In *International Conference on Architectural support for programming languages and operating systems (ASPLOS)*.

[34] Ashish Gehani and Dawood Tariq. 2012. SPADE: support for provenance auditing in distributed environments. In *International Middleware Conference*.

[35] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *the Symposium on Network and Distributed System Security (NDSS)*.

[36] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *ACM Conference on Data and Applications Security (CODASPY)*.

[37] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. 2020. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *the Symposium on Network and Distributed System Security (NDSS)*.

[38] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *the Symposium on Network and Distributed System Security (NDSS)*.

[39] Wajih Ul Hassan, Adam Bates, and Daniel Marino. 2020. Tactical Provenance Analysis for Endpoint Detection and Response Systems. In *IEEE Symposium on Security and Privacy (S&P)*.

[40] 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 *the Symposium on Network and Distributed System Security (NDSS)*.

[41] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. 2020. OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis. In *the Symposium on Network and Distributed System Security (NDSS)*.

[42] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. 2017. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security Symposium (USENIX)*.

[43] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. 2020. Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics. In *IEEE Symposium on Security and Privacy (S&P)*.

[44] 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 *ACM Conference on Computer and Communications Security (CCS)*.

[45] Kaihang Ji, Jun Zeng, Yuancheng Jiang, Zhenkai Liang, Zheng Leong Chua, Prateek Saxena, and Abhik Roychoudhury. 2022. FLOWMATRIX: GPU-Assisted Information-Flow Analysis through Matrix-Based Representation. In *USENIX Security Symposium (USENIX)*.

[46] 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 *ACM Conference on Computer and Communications Security (CCS)*.

[47] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2018. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security Symposium (USENIX)*.

[48] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[49] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of in-Production Failures. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[50] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*.

[51] Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An abstract interpretation-based framework for control flow reconstruction from binaries. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

[52] Samuel T King and Peter M Chen. 2003. Backtracking intrusions. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[53] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality.. In *the Symposium on Network and Distributed System Security (NDSS)*.

[54] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *the Symposium on Network and Distributed System Security (NDSS)*.

[55] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *the Symposium on Network and Distributed System Security (NDSS)*.

[56] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *ACM Conference on Computer and Communications Security (CCS)*.

[57] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. 2020. A qualitative study of the benefits and costs of logging from developers' perspectives. In *IEEE Transactions on Software Engineering*.

[58] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. 2020. Where shall we log? studying and suggesting logging locations in code blocks. In *International Conference on Software Engineering (ICSE)*.

[59] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. 2022. TELL: Log Level Suggestions via Modeling Multi-level Code Block Information. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[60] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[61] 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 *the Symposium on Network and Distributed System Security (NDSS)*.

[62] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*.

[63] 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 *Annual Computer Security Applications Conference (ACSAC)*.

[64] 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 *USENIX Security Symposium (USENIX)*.

[65] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. Protracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *the Symposium on Network and Distributed System Security (NDSS)*.

[66] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR.CHECKER: A soundy analysis for linux kernel drivers. In *USENIX Security Symposium (USENIX)*.

[67] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2019. POIROT: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. In *ACM Conference on Computer and Communications Security (CCS)*.

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

[69] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *USENIX Security Symposium (USENIX)*.

[70] Nathaniel Mott. 2013. Google Reveals 'Watering Hole' Attack Targeting Apple Device Owners. https://sea.pcmag.com/security/47209/google-reveals-watering-hole-attack-targeting-apple-device-owners. Online; Accessed 17 January 2022.

[71] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *the International Symposium on Recent Advances in Intrusion Detection (RAID)*.

[72] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.. In *the Symposium on Network and Distributed System Security (NDSS)*.

[73] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2016. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Annual Computer Security Applications Conference (ACSAC)*.

[74] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Security Symposium (USENIX)*.

[75] David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *International Conference on Computer Aided Verification (CAV)*.

[76] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. Bintrimmer: Towards static binary debloating through abstract interpretation. In *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[77] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *the Symposium on Network and Distributed System Security (NDSS)*.

[78] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*.

[79] 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 *ACM Conference on Computer and Communications Security (CCS)*.

[80] Benjamin E Ujcich, Samuel Jero, Richard Skowyra, Adam Bates, William H Sanders, and Hamed Okhravi. 2021. Causal Analysis for {Software-Defined} Networking Attacks. In *USENIX Security Symposium (USENIX)*.

[81] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, et al. 2020. You are what you do: Hunting stealthy malware via data provenance analysis. In *the Symposium on Network and Distributed System Security (NDSS)*.

[82] Wikipedia. 2022. Observability. https://en.wikipedia.org/wiki/Observability. Online; Accessed 18 January 2022.

[83] Yichen Xie, Andy Chou, and Dawson Engler. 2003. ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

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

[85] 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 *ACM Conference on Computer and Communications Security (CCS)*.

[86] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. In *ACM Conference on Computer and Communications Security (CCS)*.

[87] 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 *ACM Conference on Computer and Communications Security (CCS)*.

[88] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *USENIX Security Symposium (USENIX)*.

[89] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. 2020. Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. In *the Symposium on Network and Distributed System Security (NDSS)*.

[90] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela Ciocarlie, Vinod Yegneswaran, et al. 2021. ALchemist: Fusing Application and Audit Logs for Precise Attack Provenance without Instrumentation. In *the Symposium on Network and Distributed System Security (NDSS)*.

[91] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. 2021. WATSON: Abstracting Behaviors from Audit Logs via Aggregation of Contextual Semantics. In *the Symposium on Network and Distributed System Security (NDSS)*.

[92] Jun Zeng, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. 2022. ShadeWatcher: Recommendation-guided Cyber Threat Analysis using System Audit Records. In *IEEE Symposium on Security and Privacy (S&P)*.

[93] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. 2021. Statically Discovering High-Order Taint Style Vulnerabilities in OS Kernels. In *ACM Conference on Computer and Communications Security (CCS)*.

[94] 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)*.

[95] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.