Algorithm 1: PalanTír's CFG Refinement Algorithm

Function RefineCFG
  Input: Call Graph CG
  Output: Syscall-reachable Functions $\mathcal{F}$, Audit-sensitive Call Graph CG_A
  /* Filter out insensitive (noisy) functions that could invoke syscalls */
  $CG \leftarrow$ FilterOutInsensitiveFuncs (CG)
  /* Compute transitive closure graphs for syscalls in CG */
  $CG_R \leftarrow$ TransitiveClosure (CG, {read, recv, ..})
  $CG_W \leftarrow$ TransitiveClosure (CG, {write, send, ..})
  $\mathcal{F} \leftarrow CG_R$.nodes $\cup$ CG_W.nodes
  $CG_A \leftarrow$ GenGraphFromEdges (CG_R.edges $\cap$ CG_W.edges)

Function RefineAnalysisScope
  Input: Call Graph CG
  Output: Tainting Scopes $\mathcal{P}$
  $\mathcal{P} \leftarrow \{\phi\}$
  $\mathcal{F}, CG_A \leftarrow$ RefineCFG (CG)
  $\mathcal{N}_{leaf} \leftarrow CG_A$.leaf_nodes
  $CG_T \leftarrow$ RemoveNodes (CG, $\mathcal{N}_{leaf}$)
  /* Sort CG_A nodes excluding $\mathcal{N}_{leaf}$ in reversed topology order */
  $\mathcal{N}_{sorted} \leftarrow$ ReverseTopoSorting ($\{CG_A$.nodes $- \mathcal{N}_{leaf}\}$)
  foreach *Node* $N \in \mathcal{N}_{sorted}$ **do**
    if HasPath (CG_T, N, {read, recv, write, send}) **then**
      $\mathcal{N}_{leaf} \leftarrow \mathcal{N}_{leaf} \cup \{N\}$
      $CG_T \leftarrow$ RemoveNode (CG_T, N)
    end if
  end foreach
  /* Get lowest common ancestors of $\mathcal{N}_{leaf}$ */
  $Queue_{LCA} \leftarrow$ GetLCAQueue (CG_A, $\mathcal{N}_{leaf}$)
  while $Queue_{LCA} \neq \phi$ **do**
    $N \leftarrow$ PopQueue($Queue_{LCA}$)
    if $N \in \mathcal{F}$ **then**
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{N\}$
      PushQueue($Queue_{LCA}$, {N.successors})
    end if
  end while

Algorithm 2: PalanTír's Analysis Procedure Algorithm

Function Summarize
  Input: Initial State S_0, Function f, Execution Context C
  Output: Return State S_n
  FuncGraph $G \leftarrow f$.local_graph
  EntryBlock $b_0 \leftarrow f$.entry_block
  InputStatesMap $\mathcal{M} \leftarrow [b_0 \mapsto \{S_0\}]$
  BlockQueue $\mathcal{B} \leftarrow \{b_0\}$
  ExitStates $\mathcal{E} \leftarrow \{\phi\}$
  while $\mathcal{B} \neq \phi$ **do**
    Block $b \leftarrow$ PopQueue($\mathcal{B}$)
    $S_i \leftarrow \bigcup\{S \mid S \in \mathcal{M}(b)\}$ // merge input states to S_i
    $S_i \leftarrow$ InitState($S_i, S_i[0], S_i[1], C$) // init S_i's taint summary
    /* evaluate statements of block b under input state S_i */
    $S_o \leftarrow$ ProcessStmts(S_i, b)
    /* dump block b's taint summary to the in-memory database */
    DumpTaintSummary(S_o)
    if *EndWithCall* (b) **then** // b ends up with *call* instruction
      if *Callee* (b) *in tainting scopes* $\mathcal{P}$ **then**
        $S_o \leftarrow$ Summarize(S_o, Callee(b), $C \circ b$.addr)
      end if
      foreach $b_s \in G$.Successors(b) **do**
        $\mathcal{M}(b_s) \leftarrow \mathcal{M}(b_s) \cup S_o$
        PushQueue ($\mathcal{B}, b_s$)
      end foreach
    end if
    else if *EndWithRet* (b) **then** // b ends up with *ret* instruction
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{S_o\}$
    end if
    else if *EndWithJumpOut* (b) **then** // b *jmp* to another function
      if *Callee* (b) *in tainting scopes* $\mathcal{P}$ **then**
        $S_o \leftarrow$ Summarize(S_o, Callee(b), C)
      end if
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{S_o\}$
    end if
    else
      foreach $b_s \in G$.successors(b) **do**
        $\mathcal{M}(b_s) \leftarrow \mathcal{M}(b_s) \cup S_o$
        PushQueue ($\mathcal{B}, b_s$)
      end foreach
    end if
  end while
  /* merge exit states */
  $S_n \leftarrow \bigcup\{S \mid S \in \mathcal{E}\}$

A SOURCE CODE ANALYSIS

To gain further insights into the performance of tainting logic summarization, we present the source code analysis of HAProxy and Zip. The following code snippet shows how HAProxy invokes system calls in its proxying workflow.

```
/* HAProxy 1.8.30 */
void conn_fd_handler(int fd)
{
  struct connection *conn = fdtab[fd].owner;
  ...
  /* SEND: send request to the proxy backend */
  if (conn->xprt && fd_send_ready(fd) && ...) {
    conn->mux->send(conn);
  }
  /* RECV: receive response from proxy backend */
  if (conn->xprt && fd_recv_ready(fd) && ...) {
    conn->mux->recv(conn);
  }
  ...
  return;
}
```

Although HAProxy has a huge codebase (8,395KB), CFG refinement only focuses on the data transferring functionality (i.e., sending requests to a backend server and receiving responses), as shown in the *conn_fd_handler* above. The rest of HAProxy's functionalities (e.g., load balancing) will be ignored for taint summarization. To understand how execution contexts would affect the performance, we use the following code snippet of Zip:

```
/* Zip 3.0 */
uzoff_t flush_block(char buf, ulg stored_len, int eof) {
    build_tree((tree_desc near *)(&l_desc)); /* 1st callsite */
    build_tree((tree_desc near *)(&d_desc)); /* 2nd callsite */
    if (stored_len <= opt_lenb && ...) {
        copy_block(buf, (unsigned)stored_len, 0);
        ...
    } else
        if (stored_len+4 <= opt_lenb && buf != (char*)NULL) {
            send_bits((STORED_BLOCK<<1)+eof, 3);
            copy_block(buf, (unsigned)stored_len, 1);
            ...
        } else if (static_lenb == opt_lenb) {
            send_bits((STATIC_TREES<<1)+eof, 3);
            compress_block((ct_data near *)static_ltree, (ct_data near
                *)static_dtree);
            ...
        } else {
            send_bits((DYN_TREES<<1)+eof, 3);
            send_all_trees(l_desc.max_code+1, d_desc.max_code+1,
                max_blindex+1);
            compress_block((ct_data near *)dyn_ltree, (ct_data near
                *)dyn_dtree);
            ...
        }
    ...
}
```

In the analysis of function *flush_block*, we observe that it contains a plenty of callsites for functions such as *build_tree*, *copy_block*, *send_bits*, and *compress_block*. However, note that our taint summarization is context-sensitive. As such, more complex execution contexts lead to a higher overhead of taint analysis. In particular, when an inter-procedural call occurs, PALANTÍR will launch a new analysis procedure to handle its call target. That is, PALANTÍR will start multiple procedures for *flush_block* at different callsites.

B NOISE FUNCTION FILTERING

For clarity, we first define "noisy" functions as those that invoke system calls but do not cause the dependency explosion in provenance analysis. As such, these functions are not necessary for fine-grained provenance tracking. In particular, treating them as syscall-reachable functions (see Section 4.1) will increase the scope of taint analysis and further degrade the performance of both static binary analysis and post-forensic taint analysis.

Here, we classify noisy functions into three categories as follows: (1) Configuration and user input parsing procedures. These functions invoke input system calls (e.g., `read`) to configure and initialize a process from either local configuration files or the standard user input. (2) Logging procedures. As a common usage to monitor program execution status, logging procedures write application logs into a local file (e.g., `/var/log/apache2/access.log`) through output system calls (e.g., `write`). (3) Special file procedures. In order to load libraries or request hardware-level resources, those procedures invoke input system calls to access read-only or device files such as `/dev/urandom`.

We currently provide system administrators with an interface to define noisy functions. A fully automatic solution for identifying such procedures can be done based on selective symbolic execution or static analysis [42, 95], which we leave for future work.

C DETAILS OF STATIC ANALYSIS

In this section, we describe technical details of PALANTÍR's static analysis in tainting logic summarization. In the following, we use $a.\texttt{type}$ to denote the type of an abstract location a. For example, given a global region $a = \text{Global}(o)$, $a.\texttt{type}$ returns Global. In addition, we use $m.\texttt{keys}$ to represent all keys contained in a map m. For example, given a register map R of an abstract state S, $R.\texttt{keys}$ returns all registers held by R.

C.1 Join Operation

Before defining the join operation, we first introduce the *merge* of abstract values. Given two input abstract values v_1 and v_2, $MergeVal$ returns their merged set. If v_1 and v_2 share the same variable (i.e., $v_1[0] = v_2[0]$), then the returned abstract value set contains a single abstract value whose abstract region set and taint tag set are merged from those of v_1 and v_2. Otherwise, $MergeVal$ will group two abstract values together as the returned value set. Intuitively, $MergeVal$ can also be extended to operate on multiple abstract values. We present the procedure of $MergeVal$ as follows:

$$MergeVal(v_1, v_2) = \begin{cases} \{\langle v_1[0], v_1[1] \cup v_2[1], v_1[2] \cup v_2[2] \rangle\} & \text{if } v_1[0] = v_2[0] \\ \{v_1, v_2\} & \text{otherwise} \end{cases}$$

Then, we define the join operation on our abstract domains, including abstract value set (V), register map (R), memory map (M), and abstract state (S). Note that the join operation on abstract states can only apply to two abstract states with the same execution context. The join operations are formally defined below.

$$V_1 \cup V_2 = \bot_{MergeVal}\{MergeVal(v_i, v_j) \mid v_i \in V_1, v_j \in V_2\}$$

$$R_1 \cup R_2 = \bigcup_{r_i}[r_i \mapsto R_1(r_i) \cup R_2(r_i)], \text{ where } r_i \in R_1.\texttt{keys} \cup R_2.\texttt{keys}$$

$$M_1 \cup M_2 = \bigcup_{a_i}[a_i \mapsto M_1(a_i) \cup M_2(a_i)], \text{ where } a_i \in M_1.\texttt{keys} \cup M_2.\texttt{keys}$$

$$S_1 \cup S_2 = \langle S_1[0] \cup S_2[0], S_1[1] \cup S_2[1], S_1[2] \rangle \quad (\text{where } S_1[2] = S_2[2])$$

C.2 Binary Operation

First of all, we introduce binary operations (generalized as $\hat{\diamond}$) on abstract variables. Specifically, We design binary operations based on NTFuzz [22], in which we only support linear operation for a single symbol in abstract variables. If two abstract variables contain different symbols, we conservatively return $\top$. Here we only provide the semantics of $\hat{+}$ for simplicity, but it can also be generalized to other operations:

$$i \mathbin{\hat{\diamond}} \top = \top,\ i \mathbin{\hat{\diamond}} \bot = \bot \quad (\diamond \in \{+, -, \times, \div, \wedge, \vee, \oplus, \ldots\})$$

$$(a_1 y_1 + b_1) \mathbin{\hat{+}} (a_2 y_2 + b_2) = \begin{cases} (b_1 + b_2) & a_1 = a_2 = 0 \\ a_1 y_1 + (b_1 + b_2) & (a_2 = 0) \\ a_2 y_2 + (b_1 + b_2) & (a_1 = 0) \\ (a_1 + a_2) y_1 + (b_1 + b_2) & (y_1 = y_2) \\ \top & \text{otherwise} \end{cases}$$

For binary operations on abstract regions, we only accept arithmetic operations (specifically, addition and subtraction) that track constant offsets on the regions. For example, given an abstract region a with the offset o_1, it can add a constant offset o_2 and returns an abstract region with offset $o_1 + o_2$.

$$\text{Stack}(o_1) \mathbin{\hat{+}} o_2 = \text{Stack}(o_1 + o_2) \quad \text{where } o_2 \in \mathbb{Z}$$

$$\ldots$$

$$\text{SymLoc}(y, o_1) \mathbin{\hat{+}} o_2 = \text{SymLoc}(y, o_1 + o_2) \quad \text{where } o_2 \in \mathbb{Z}$$

Before explaining how we handle the evaluation on binary operation expressions, we first define the difference set operation (denoted as $\dot{-}$) on two abstract region sets A_1 and A_2, as shown in the following equation:

$$A_1 \dot{-} A_2 = \{a_1 \mid a_1 \in A_1 \text{ and } a_1.\texttt{type} \notin \{a_2.\texttt{type} \mid a_2 \in A_2\}\}$$

Next, we give the formal definition of evaluating binary operation expression $OP(S, e_1, e_2)$. Recall that $BinOP$ operates on two abstract value sets V_1 and V_2, where $V_1 = \mathcal{G}(S, e_1)$ and $V_2 = \mathcal{G}(S, e_2)$, and returns the resulting abstract value set according to the semantics of binary operations (e.g., addition). It iterates on V_1 and V_2 and computes all pairs of abstract values. The taint tag set of each returned value is a join of those sets in the input pair of abstract values. Note that for the addition operation, we only track the offsets from constant expressions while computing abstract regions for returned values. As such, we ignore the array index and achieve

$$ADD(S, e_1, e_2) = \begin{cases} \bigcup_{v_1 \in V_1} \{\langle v_1[0] \hat{+} c, v_1[1] \hat{+} c, v_1[2] \rangle\} \\ \text{where } V_1 = \mathcal{G}(S, e_1),\ e_2 = \text{CONST}(c) \text{ and } c \notin data\_section \\ \bigcup_{v_2 \in V_2} \{\langle v_2[0] \hat{+} c, v_2[1] \hat{+} c, v_2[2] \rangle\} \\ \text{where } V_2 = \mathcal{G}(S, e_2),\ e_1 = \text{CONST}(c) \text{ and } c \notin data\_section \\ \bigcup_{v_1 \in V_1} \bigcup_{v_2 \in V_2} \{\langle v_1[0] \hat{+} v_2[0], v_1[1] \cup v_2[1], v_1[2] \cup v_2[2] \rangle\} \\ \text{where } V_1 = \mathcal{G}(S, e_1),\ V_2 = \mathcal{G}(S, e_2) \end{cases}$$

$$SUB(S, e_1, e_2) = \begin{cases} \bigcup_{v_1 \in V_1} \{\langle v_1[0] \hat{-} c, v_1[1] \hat{-} c, v_1[2] \rangle\} \\ \text{where } V_1 = \mathcal{G}(S, e_1),\ e_2 = \text{CONST}(c) \\ \bigcup_{v_1 \in V_1} \bigcup_{v_2 \in V_2} \{\langle v_1[0] \hat{-} v_2[0], v_1[1] \dot{-} v_2[1], v_1[2] \cup v_2[2] \rangle\} \\ \text{where } V_1 = \mathcal{G}(S, e_1),\ V_2 = \mathcal{G}(S, e_2) \end{cases}$$

$$MUL(S, e_1, e_2) = \bigcup_{v_1 \in V_1} \bigcup_{v_2 \in V_2} \{\langle v_1[0] \hat{\times} v_2[0], \phi, v_1[2] \cup v_2[2] \rangle\}$$
$$\text{where } V_1 = \mathcal{G}(S, e_1),\ V_2 = \mathcal{G}(S, e_2)$$

array-insensitive analysis while tracking memory regions. In subtraction operation, we leverage the difference set operation ($\dot{-}$) to determine abstract regions of returned values when the subtrahend's expression is non-constant. For the multiplication operation, we assign abstract regions of return values as an empty set. For simplicity, we only present typical operations like *ADD*, *SUB*, and *MUL* below. Other operations, such as logical *OR* and *AND*, are similar to *MUL*.

D PHISHING EMAIL ATTACK

Sending phishing emails to gain access to victim systems is a common attack vector [4]. In this attack, an employee of a national government one day receives a shedload of emails through `sendmail`. An email (`msg.77BC`) titled "EMERGENCY LISTS" quickly raises her attention. After opening the email, she finds an EXCEL spreadsheet called "LIST.xlsx" in the attainment. Out of curiosity, she then downloads the attached file, which is, in effect, malware that allows attackers to gain initial access.

Later, the employee discovers a suspicious program that frequently communicates with an external IP address. After identifying that the program is downloaded from an email (`msg.77BC`), she performs provenance tracking on `msg.77BC` for attack forensics. Figure 15 shows the provenance graph generated by PALANTÍR, where the green box presents that the email is sent from `x.x.x.x`. As can be observed, `sendmail` first receives the email from `x.x.x.x` and saves it to a temporary file (`df23K4aMRW003661`). Then, it spawns another process (`procmail`) to decode the email so that it can be parsed by email clients (e.g., Mutt). Note that through the provenance tracking, we find that `sendmail` does not lead to the dependency explosion problem. Even so, PALANTÍR helps confirm the information flow from `x.x.x.x` to `msg.77BC`, and traditional provenance-based system [53] only conservatively assume that there exists causality.

E SETUP FOR RUNTIME EVALUATION

Besides the CPU-bound SPEC benchmarks, we also evaluate PALANTÍR's runtime performance on IO-intensive programs from Table 2, including both server applications and client tools.

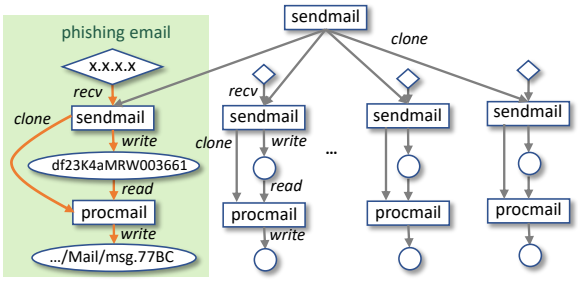

Figure 15: Fine-grained provenance of Phishing Email.

Settings. We first create random files in different sizes. To do so, we use a Linux utility `dd` to generate files with the input source from `/dev/urandom`. Afterward, to evaluate on HTTP web servers (e.g., Nginx and Httpd), we leverage ApacheBench [3] to request a 10MB file 10,000 times using 32 concurrent connections. For FTP web servers (e.g., Proftpd and Pure-ftpd), we use Siege [4] to request a 10MB file 1,000 times using five concurrent connections. For client-side tools (i.e., Wget and Curl), we download a 2GB file five times. For a fair evaluation, we set up local area network connections to avoid potential network perturbations. As for native GNU software (i.e., Cp and Zip), we run them using 2GB files (copying/compressing 2GB files) five times.

F STORAGE FOR LONG-RUNNING SERVICES

We evaluate PALANTÍR's storage on long-running services using four web servers (i.e., Nginx, Httpd, Lighttpd, and Thttpd). To simulate a high frequency of client requests, we adopt ApacheBench to request files (ranging from 100KB to 10MB) 10,000 times through 32 concurrent connections. The results are summarized in Table 6. In general, PALANTÍR's storage overhead per request increases with the size of the requested file. Httpd produces noticeably more PT data as it needs to create processes to handle client requests. However, even for Httpd, in the use of PT tracing, we observe only around 30KB cost per 10MB file request. That is, given a 512GB disk, PALANTÍR allows storing the past 17 million requests, which we believe is practical in production systems.

Table 6: Storage cost (MB) to trace web servers using PT for 10,000 file requests.

	100KB	512KB	1MB	10MB
Nginx	74	74	75	83
Httpd	274	275	274	304
Lighttpd	42	51	55	107
Thttpd	46	50	50	57

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] https://httpd.apache.org/docs/2.4/programs/ab.html
[4] https://github.com/JoeDog/siege

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

[4] 2020. MITRE T1566: Phishing. https://attack.mitre.org/techniques/T1566/. Online; Accessed 15 March 2020.

[5] 2020. Twitter hack. https://www.theguardian.com/technology/2020/jul/15/twitter-elon-musk-joe-biden-hacked-bitcoin. Online; Accessed 25 March 2020.

[6] 2021. Intel Processor Trace. https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html. Online; Accessed 29 March 2022.

[7] 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.

[8] 2021. Powerful Disassembler Library For x86/AMD64. https://github.com/gdabah/distorm. Online; Accessed 6 April 2022.

[9] 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.

[10] 2022. Artifact Release and Appendix. https://github.com/Icegrave0391/Palantir.

[11] 2022. Linux Kernel Audit Subsystem. https://github.com/linux-audit/audit-kernel. Online; Accessed 10 March 2021.

[12] 2022. Neo4j Graph Database. https://neo4j.com. Online; Accessed 6 April 2022.

[13] 2022. Redis. https://redis.io. Online; Accessed 6 April 2022.

[14] 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)*.

[15] 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)*.

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

[17] 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)*.

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

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

[20] 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)*.

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

[22] 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)*.

[23] 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)*.

[24] 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)*.

[25] 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)*.

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

[27] 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)*.

[28] 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)*.

[29] 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)*.

[30] 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)*.

[31] 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)*.

[32] 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)*.

[33] 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)*.

[34] 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)*.

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

[36] 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)*.

[37] 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)*.

[38] 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)*.

[39] 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)*.

[40] 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)*.

[41] 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)*.

[42] 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)*.

[43] 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)*.

[44] 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)*.

[45] 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)*.

[46] 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)*.

[47] 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)*.

[48] 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)*.

[49] 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)*.

[50] 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)*.

[51] 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)*.

[52] 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)*.

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

[54] 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)*.

[55] 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)*.
[56] 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)*.
[57] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *ACM Conference on Computer and Communications Security (CCS)*.
[58] 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*.
[59] 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)*.
[60] 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)*.
[61] 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)*.
[62] 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)*.
[63] 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*.
[64] 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)*.
[65] 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)*.
[66] 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)*.
[67] 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)*.
[68] 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)*.
[69] 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)*.
[70] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *USENIX Security Symposium (USENIX)*.
[71] 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.
[72] 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)*.
[73] 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)*.
[74] 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)*.
[75] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Security Symposium (USENIX)*.
[76] David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *International Conference on Computer Aided Verification (CAV)*.
[77] 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)*.
[78] 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)*.
[79] 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)*.
[80] 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)*.
[81] 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)*.
[82] 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)*.
[83] Wikipedia. 2022. Observability. https://en.wikipedia.org/wiki/Observability. Online; Accessed 18 January 2022.
[84] 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)*.
[85] 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)*.
[86] 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)*.
[87] 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)*.
[88] 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)*.
[89] 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)*.
[90] 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)*.
[91] 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)*.
[92] 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)*.
[93] 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)*.
[94] 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)*.
[95] Shulin Zhou, Xiaodong Liu, Shanshan Li, Wei Dong, Xiangke Liao, and Yun Xiong. 2016. ConfMapper: Automated Variable Finding for Configuration Items in Source Code. In *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*.
[96] 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)*.
[97] 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)*.