OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis

Wajih Ul Hassan, Mohammad A. Noureddine, Pubali Datta, Adam Bates
University of Illinois at Urbana-Champaign
{whassan3, nouredd2, pdatta2, batesa}@illinois.edu

***Abstract*—Recent advances in causality analysis have enabled investigators to trace multi-stage attacks using provenance graphs. Based on system-layer audit logs (e.g., syscalls), these approaches omit vital sources of application context (e.g., email addresses, HTTP response codes) that can be found in higher layers of the system. Although such information is often essential to understanding attack behaviors, it is difficult to incorporate this evidence into causal analysis engines because of the semantic gap that exists between system layers. To address that shortcoming, we propose the notion of *universal provenance*, which encodes all forensically relevant causal dependencies regardless of their layer of origin. To transparently realize that vision on commodity systems, we present OmegaLog, a provenance tracker that bridges the semantic gap between system and application logging contexts. OmegaLog analyzes program binaries to identify and model application-layer logging behaviors, enabling accurate reconciliation of application events with system-layer accesses. OmegaLog then intercepts applications' runtime logging activities and grafts those events onto the system-layer provenance graph, allowing investigators to reason more precisely about the nature of attacks. We demonstrate that our system is widely applicable to existing software projects and can transparently facilitate *execution partitioning* of provenance graphs without any training or developer intervention. Evaluation on real-world attack scenarios shows that our technique generates concise provenance graphs with rich semantic information relative to the state-of-the-art, with an average runtime overhead of 4%.**

I. Introduction

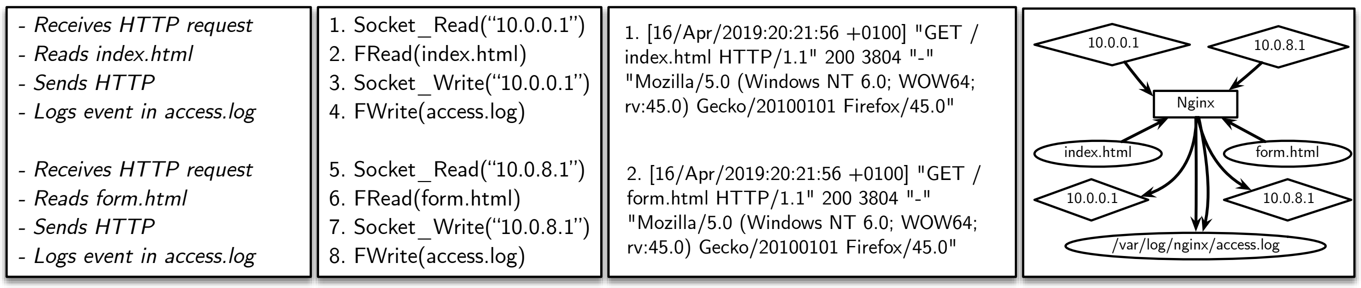
System intrusions are becoming progressively more subtle and complex. Using an approach exemplified by the "low and slow" attack strategy of *Advanced Persistent Threats*, attackers now lurk in target systems for extended periods to extend their reach before initiating devastating attacks. By avoiding actions that would immediately arouse suspicion, attackers can achieve dwell times that range from weeks to months, as was the case in numerous high-profile data breaches including Target [14], Equifax [12], and the Office of Personnel Management [13].

Against such odds, advancements in system auditing have proven invaluable in detecting, investigating, and ultimately responding to threats. The notion of data provenance has been applied to great effect on traditional system audit logs, parsing individual system events into provenance graphs that encode the history of a system's execution [17], [26], [51], [32], [36], [40], [31]. Such provenance graphs allow investigators to trace the root causes and ramifications of an attack by using causality analysis. Leveraging this principal capability, causality analysis has matured from a costly offline investigation tool to a highly-efficient method of tracing attackers in real-time [31], [16].

Given the importance of threat investigation to system defense, it is perhaps surprising that prior work on causality analysis has been oblivious to application-layer semantics. As an example, consider the execution of the web service shown in Fig. 1. Fig. 1(a) describes the event sequence of the example, in which the server responds to two HTTP requests for `index.html` and `form.html`, respectively, yielding the system log shown in Fig. 1(b). As a normal part of its execution, the server also maintains its own event logs that contain additional information (e.g., user-agent strings) shown in Fig. 1(c), that is opaque to the system layer. State-of-the-art causality analysis engines, using system audit logs, produce a provenance graph similar to Fig. 1(d); however, the forensic evidence disclosed by the application itself is not encoded in this graph. That is unfortunate, as recent studies [25], [21], [49] have shown that developers explicitly disclose the occurrence of *important* events through application logging. Further, we observe that the well-studied problem of *dependency explosion* [39], [42], [41], which considers the difficulty of tracing dependencies through high-fanout processes, is itself a result of unknown application semantics. For example, the dependency graph in Fig. 1 (d) is not aware that the NGINX vertex can be subdivided into two autonomous units of work, marked by the two HTTP requests found in the application event log.

Prior work on log analysis has not provided a generic and reliable (i.e., causality-based) solution to cross-layer attack investigation. Techniques for execution partitioning mitigate dependency explosion by identifying limited and coarse-grained application states, e.g., when a program starts its main event-handling loop [39], but require invasive instrumentation [39], [41] or error-prone training [39], [40], [42]. Past frameworks for layered provenance tracking [57], [28], [17], [47] technically support application semantics, but rather than harness the developer's original event logs, instead call for costly (and redundant!) instrumentation efforts. Elsewhere in the literature, application event logs have been leveraged for program debugging [24], [59], [60], profiling [65], [64], and runtime monitoring [48]; however, these approaches are application-centric, considering only one application's siloed event logs at a time, and thus cannot reconstruct complex workflows between multiple processes. Attempts to "stitch" application logs together to trace multi-application workflows [50], [65], [64] commonly ignore the system layer, but also

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24270
www.ndss-symposium.org



```
- Receives HTTP request
- Reads index.html
- Sends HTTP
- Logs event in access.log

- Receives HTTP request
- Reads form.html
- Sends HTTP
- Logs event in access.log
```

(a) Execution

```
1. Socket_Read("10.0.0.1")
2. FRead(index.html)
3. Socket_Write("10.0.0.1")
4. FWrite(access.log)

5. Socket_Read("10.0.8.1")
6. FRead(form.html)
7. Socket_Write("10.0.8.1")
8. FWrite(access.log)
```

(b) System Log

```
1. [16/Apr/2019:20:21:56 +0100] "GET /
index.html HTTP/1.1" 200 3804 "-"
"Mozilla/5.0 (Windows NT 6.0; WOW64;
rv:45.0) Gecko/20100101 Firefox/45.0"

2. [16/Apr/2019:20:21:56 +0100] "GET /
form.html HTTP/1.1" 200 3804 "-"
"Mozilla/5.0 (Windows NT 6.0; WOW64;
rv:45.0) Gecko/20100101 Firefox/45.0"
```

(c) Application Log

(d) System Provenance Graph

Fig. 1: NGINX application execution while two different HTTP requests are being served. **(a)** Actual execution behavior of NGINX. **(b)** System logs generated by whole-system provenance tracker. **(c)** Application event logs generated by NGINX. **(d)** Provenance graph generated using system logs by traditional solutions.

use ad hoc rules and *co-occurrence* of log events to assume a *causal* relationship; this assumption introduces error and could potentially undermine threat investigations.

In this work, we argue that attack investigation capabilities can be dramatically improved through the unification of *all* forensically relevant events on the system in a single holistic log. To achieve that vision transparently and effortlessly on today's commodity systems, we present OmegaLog, an end-to-end provenance tracker that merges application event logs with the system log to generate a *universal provenance graph* (UPG). This graph combines the causal reasoning strengths of whole-system logging with the rich semantic context of application event logs. To construct the UPG, OmegaLog automatically parses dispersed, intertwined, and heterogeneous application event log messages at runtime and associates each record with the appropriate abstractions in the whole-system provenance graph. Generating UPG allows OmegaLog to transparently solve both the dependency explosion problem (by identifying event-handling loops through the application event sequences) and the semantic gap problem (by grafting application event logs onto the whole-system provenance graph). Most excitingly, OmegaLog does not require any instrumentation on the applications or underlying system.

Several challenges exist in the design of a universal provenance collection system. First, the ecosystem of software logging frameworks is heterogeneous, and event logging is fundamentally similar to any other file I/O, making it difficult to automatically identify application logging activity. Second, event logs are regularly multiplexed across multiple threads in an application, making it difficult to differentiate concurrent units of work. Finally, each unit of work in an application will generate log events whose occurrence and ordering vary based on the dynamic control flow, requiring a deep understanding of the application's logging behavior to identify meaningful boundaries for execution unit partitioning.

To solve those challenges, OmegaLog performs static analysis on application binaries to automatically identify log message writing procedures, using symbolic execution and emulation to extract descriptive Log Message Strings (LMS) for each of the call sites. Then, OmegaLog performs control flow analysis on the binary to identify the temporal relationships between LMSes, generating a set of all valid LMS control flow paths that may occur during execution. At runtime, OmegaLog then uses a kernel module that intercepts write syscall and catches all log events emitted by the application, associating each event with the correct PID/TID and timestamp to detangle concurrent logging activity. Finally, those augmented application event logs are merged with system-level logs into a unified universal provenance log. Upon attack investigation, OmegaLog is able to use the LMS control flow paths to parse the flattened stream of application events in the universal log, partition them into execution units, and finally add them as vertices within the whole-system provenance graph in *causally correct* manner.

The main contributions of this paper are as follows:

⋆ We propose the concept of the universal provenance that combines the advantages of whole-system provenance with applications' innate event-logging activity, providing a transparent and generic solution for the semantic gap problem in threat investigations.
⋆ We develop robust binary analysis techniques to automatically extract logging behaviors from an application. Our proof-of-concept implementation, OmegaLog, non-intrusively collects and integrates applications' event logs with the Linux audit logs [5].
⋆ We evaluate OmegaLog for performance, accuracy, and efficacy. Our results indicate that OmegaLog exhibits low runtime overheads (4%), is broadly deployable to existing software projects, and enables semantically rich attack reconstructions in real-world attack scenarios.

II. Motivation

In this section, we explain the motivation for our approach by considering a data exfiltration and defacement attack on an online shopping website. We use this example to illustrate the limitations of existing provenance tracking systems [17], [42], [41], [16], [36], [37], [38]. Consider a simple WordPress website hosted on a web server. Requests to the website are first received by an HAProxy, which balances load across different Apache instances running on the web server, while customer transactions are recorded in a PostgreSQL database. The administrator has turned on application event logging for Apache httpd, HAProxy, and PostgreSQL. In addition, the server is performing system-level logging, e.g., through Linux Audit (auditd) [5] or Linux Provenance Modules (LPM) [17], which continuously collect system logs. One day, the administrator discovers that the online store has been defaced and that some of the sensitive customer information has been posted to a public Pastebin website. On average, the shopping website

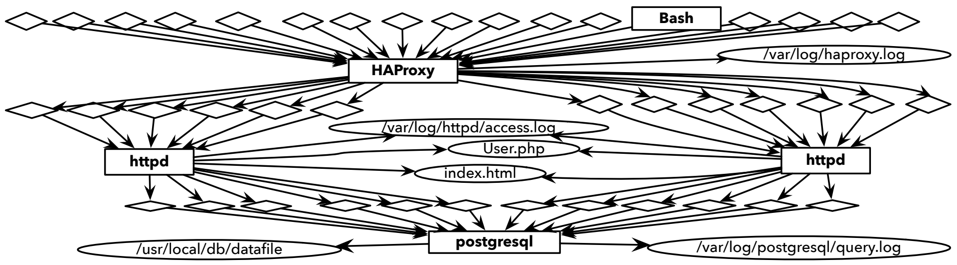


Fig. 2: A whole-system provenance graph showing the SQL injection attack scenario. Diamond, box, and oval vertices represent network connections, processes, and files, respectively. This graph suffers from both dependency explosion and semantic gap problems, frustrating attack investigation.

receives tens of thousands of requests per day; among those, one request was malicious.

A. Investigating with Application Event Logs

To attribute the attack and prepare an appropriate response, the administrator initiates a forensic inquiry by first inspecting the application event logs. The administrator finds that the `accounts` database table must have been accessed and uses this as a *symptom* to initiate attack investigations. The admin then runs a `grep` query on PostgreSQL event logs, which returns the following query log message:

```
SELECT * FROM users WHERE user_id=123 UNION SELECT
    password FROM accounts;
```

This log message strongly indicates that an attacker exploited a SQL injection vulnerability in the website, and also suggests that the attacker was able to retrieve the login credentials for `admin.php` which gave attacker privileged site access.

Limitations of Application Event Logs. At this point, the administrator is unable to proceed in the investigation using application event logs alone. It is clear that the HAProxy and Apache httpd logs contain important evidence such as the HTTP requests associated with the SQL injection attack, but re-running of the same `grep` query on Apache's logs did not return any result. The reason is that the attacker used a `POST` command to send the SQL query and that command was not contained in the URL captured in the Apache httpd event log messages. The investigation has stalled with important questions left unanswered: 1) *What was the IP address associated with the malicious HTTP request?* 2) *How were the login credentials used to deface the website, and what additional damage was caused?* 3) *Which PHP file on the site is not properly sanitizing user inputs, exposing the SQL injection vulnerability?* Those questions reflect an inherent limitation of application event logs: they cannot causally relate events across applications and thus cannot trace workflow dependencies.

B. Investigating with System Logs

To proceed, the administrator attempts to perform causality analysis using a whole-system provenance graph. At this layer, it is easy to trace dependencies across multiple coordinated processes in a workflow. Because the malicious query shown above resulted in a read to the PostgreSQL database, the administrator uses `/usr/local/db/datafile.db` as a *symptom* event and issues a backtrace query, yielding the provenance graph shown in Fig. 2. Unfortunately, the administrator discovers that this technique does not advance the investigation because of the inherent limitations of system logs.

TABLE I: Comparison of execution partitioning techniques to solve the dependency explosion problem.

	BEEP [39] ProTracer [42]	MPI [41]	MCI [38]	WinLog [40]	**OmegaLog**
Instrumentation	Yes	Yes	No	No	**No**
Training Run w/ Workloads	Yes	No	Yes	No	**No**
Space Overhead	Yes	Yes	Yes	Yes	**No**
Granularity	Coarse	Fine	Coarse	Coarse	**Fine**
App. Semantics	No	No	No	No	**Yes**

Limitation of System Logs #1: Dependency Explosion. The administrator's backtrace identifies thousands of "root causes" for the SQL injection attack because of the dependency explosion problem. The reason is that system-layer provenance trackers must conservatively assume that the output of a process is causally dependent on *all* preceding process inputs [39], [42], [41], [38]. Although the malicious query string is known, causal analysis does not allow the administrator to associate the query with a particular outbound edge of `/usr/local/db/datafile.db` in the provenance graph. Even if the administrator restricted most of the dependencies between Apache httpd and PostgreSQL (e.g., though timing bounds), admin would again face the same problem when identifying which input request from HAProxy to Apache httpd lies on the attack path.

Recent work [39], [42], [40] has introduced *execution partitioning* as a viable solution to the dependency explosion problem. These systems decompose long-running processes into autonomous "units", each representing an iteration of event-handling loop, such that input-output dependencies are traced only through their corresponding unit. Where event handling loops do not encode work units, Kwon et al. propose an inference-based technique for identifying units from system log traces [38] while Ma et al. propose a framework for manually annotating source code to disclose meaningful unit boundaries [41].

Unfortunately, prior approaches suffer from noteworthy limitations, which we summarize in Table I. Most execution partitioning systems rely on *instrumentation* to identify unit boundaries, requiring either domain knowledge or manual effort and assuming the right to modify program binaries, which is not always available [40]. The common requirement of *training runs* exposes systems like BEEP and Protracer to the classic code-coverage problem present in any dynamic analysis, and inference-based techniques (MCI) may also struggle with out-of-order events due to the presence of concurrent or cooperating applications during training runs. All past approaches introduce additional *space overhead* in order to track unit boundaries; fully automated identification of event loops (BEEP, Protracer) can generate excessive units that can waste space and CPU cycles [41]. Most notably, prior approaches do not consider the broader value of *application semantics* as forensic evidence outside of the bare minimum required for the identification of work units.

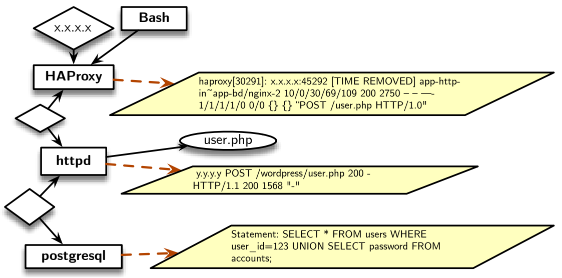
Limitation of System Logs #2: Semantic Gap. Existing system-level provenance logs are beneficial in that they offer a broad view of system activity, but unfortunately they lack knowledge of application-specific behaviors that are pivotal for attack reconstruction. In our motivating example, information such as failed login attempts, HTTP headers, WordPress plugin behavior, and SQL queries cannot be extracted from system logs. Such information is present in the siloed event logs of each application; PostgreSQL maintained a record of all SQL queries, and HAProxy recorded the headers for all HTTP requests. However, it is not possible to associate those event descriptions with the system records reliably in a post-hoc manner, because of multi-threaded activity and ambiguous or incomplete information within the application event logs.

Prior work has sought to address the semantic gap problem through instrumentation-based techniques [57], [28], [55]. Those approaches either statically or dynamically instrument function calls in the application to *disclose* function names, arguments, and return values. However, such instrumentation-based systems suffer from several limitations: (1) developers need to specify which functions to instrument, imposing a domain knowledge requirement; (2) the logging information is captured on a per-application basis and thus cannot be used to connect information flow between different applications; and (3) high-level semantic events may not always be effectively captured at the function call level.

C. Our Approach

Recent work in application logging [25], [65], [64], [21], [49] has shown the efficacy of application logs in program understanding, debugging, and profiling. OmegaLog takes inspiration from those efforts, with the goal of better leveraging event logs during attack investigation. *The key insight behind OmegaLog is that developers have already done the hard work of encoding high-level application semantics in the form of event logging statements*; these statements not only contain the relevant forensic information that we require, but also mark the boundaries of execution units in the program. The insertion of event logging statements is an organic byproduct of sound software engineering practices, permitting developers and users to better understand programs' runtime behavior. Thus, it is possible to enrich system logs with application semantics without further instrumentation or profiling. Moreover, these applications logs can be used to identify execution units.

Applying that intuition to our motivating example yields the provenance graph in Fig. 3a, which was generated using OmegaLog. The administrator can associate the malicious SQL

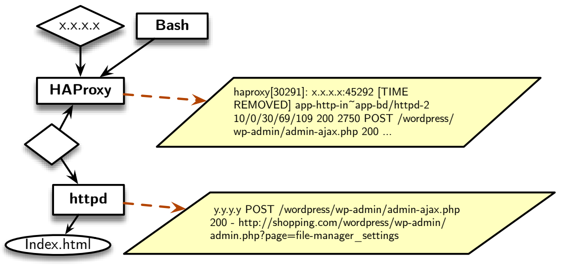


```
haproxy[30291]: x.x.x.x:45292 [TIME REMOVED] app-http-
in~app-bd/nginx-2 10/0/30/69/109 200 2750 – – —-
1/1/1/1/0 0/0 {} {} "POST /user.php HTTP/1.0"
```

```
y.y.y.y POST /wordpress/user.php 200 -
HTTP/1.1 200 1568 "-"
```

```
Statement: SELECT * FROM users WHERE
user_id=123 UNION SELECT password FROM
accounts;
```

(a) Investigating SQL injection attack using SQL query that reads the `accounts` table.



```
haproxy[30291]: x.x.x.x:45292 [TIME
REMOVED] app-http-in~app-bd/httpd-2
10/0/30/69/109 200 2750 POST /wordpress/
wp-admin/admin-ajax.php 200 ...
```

```
y.y.y.y POST /wordpress/wp-admin/admin-ajax.php
200 - http://shopping.com/wordpress/wp-admin/
admin.php?page=file-manager_settings
```

(b) Investigating website defacement using a file write event to `index.html` as a symptom.

Fig. 3: Graphs generated by OmegaLog for the SQL injection attack. The parallelograms represent the *app log vertices*. App log vertex is annotated with log messages which belong to the corresponding execution unit of attached process vertex.

query with a specific system call event (`read`). By performing execution partitioning on PostgreSQL using OmegaLog's logging behavior analysis, the administrator is then able to trace back to system calls issued and received by Apache httpd, which are also annotated with application events describing the vulnerable web form. Iteratively, OmegaLog uses execution partitioning again to trace back to the correct unit of work within HAProxy to identify the IP address of the attacker. After finding out how the user data and login credentials were stolen using SQL injection, the investigator tries to figure out how the website was defaced by issuing a backward-tracing query on the `index.html` file. Using the OmegaLog provenance graph shown in Fig. 3b, the investigator deduces that the attacker used a WordPress file manager plugin to change `index.html`.

III. Threat Model & Assumptions

This work considers an attacker whose primary goal is to exploit a security vulnerability in an application running on a system and exfiltrate or manipulate sensitive information present in the system. We make the typical assumptions of work in this space about the integrity of the operating system, kernel-layer auditing framework, audit logs and application event logs, all of which is in our trusted computing base (TCB) (cf., [39], [42], [41], [36], [30], [38], [58], [16]). That assumption is made more reasonable through system-hardening techniques, e.g., [17], [20], designed to mitigate threats to system logs. Like all prior work on execution partitioning [39], [42], [41], [38], [29], [32], we also assume the integrity of applications' control flows (further discussed in §X). We consider hardware-layer trojans, side channel attacks, and backdoors to be out of scope of this paper.

IV. Application Logging Behaviour

Our approach to partition long-running program into execution units and overcome the dependence explosion problem depends on the pervasiveness of event-logging behavior in those applications. Fortunately, the importance of logging in applications has been widely established [33]. Practically, all open-source applications print event log messages, offering four levels of verbosity: *`FATAL`* is for an error that is forcing a shutdown, *`ERROR`* is for any error that is fatal to the operation, *`INFO`* is for generally useful information, and *`DEBUG`* is for information that is diagnostically helpful. Note that logging levels are inclusive; higher levels also print messages that belong to lower levels (i.e. `FATAL` $\subseteq$ `ERROR` $\subseteq$ `INFO` $\subseteq$ `DEBUG`).

However, to partition successful executions of an application into its units, we require log messages with verbosity level of `INFO` or `DEBUG` to be present *inside* event-handling loops. Unfortunately, such behavior in applications has not been investigated. In that regard, we studied a large number of popular open-source applications.

We collected a list of 79 long-running Linux applications which belong to different categories. Those applications are written in the C/C++, Java, Python, and Erlang programming languages. We investigated the source code and man pages of those applications to identify the event-handling loops and understand if they print log messages for each meaningful event. Lee et al. [39] conducted a similar study in 2013 but they only analyzed the design patterns of open-source applications and the pervasiveness of event-handling loops as drivers for execution. They did not however study the logging behavior of those applications and the presence of log messages inside event-handling loops.

We summarize our results in Table II. In the column "Apps with Log Verbosity of", we show how many of 79 profiled applications include log statements in their event-handling loop at verbosity of `INFO` and `DEBUG`, and how many of 79 applications do not print meaningful log messages for new events. We observe that 39 applications print log with both `INFO` and `DEBUG` verbosity levels (`IN+DE`) inside the event-handling loops. While 8 applications only log at `INFO` level and 17 applications only log at `DEBUG` level.[1] We show the intra-event-handling loop logging behavior of some of the well-know applications in Figure 4.

During our study, we found 15 applications that do not have any information about event logs in their source code or in man pages. We categorized those applications as follows:

- **Light-weight Applications:** Certain client-server applications are designed to be light-weight to keep a minimal resource footprint. Those applications – including `thttpd` (Web server) and `skod` (FTP client) – do not print log messages for new events.
- **GUI Applications:** We observe that 12 out of 17 GUI applications either (1) do not print log messages, or (2) they print log messages that do not match the expectations of the forensic investigator. In other words, those log messages were not meaningful to partition the execution. Ma et al. [41] also observed similar behavior for GUI applications where event-handling loops do not correspond to the high-level logic tasks. For example, we found that none of the PDF readers in our study printed log messages whenever a new PDF file was opened. Such PDF file open event is forensically important event for threat investigations [41].

[1] For web servers such as lighttpd and NGINX, we treat the Access Log as INFO level log. Moreover, for certain applications that do not have `DEBUG` log level, we categorize the Trace Log as `DEBUG` level log.

TABLE II: Logging behavior of long-running applications.

	Category	Total Apps	Apps with Log Verbosity of			
			`IN+DE`	**`INFO`**	**`DEBUG`**	**None**
Client-Server	Web server	9	7	1	0	1
	Database server	9	7	1	1	0
	SSH server	5	5	0	0	0
	FTP server	5	4	0	1	0
	Mail server	4	3	1	0	0
	Proxy server	4	3	1	0	0
	DNS server	3	2	0	1	0
	Version control server	2	0	1	1	0
	Message broker	3	2	0	1	0
	Print server	2	1	0	1	0
	FTP client	6	0	1	4	1
	Email client	3	1	0	1	1
	Bittorrent client	4	3	1	0	0
	NTP client	3	0	1	2	0
GUI	Audio/Video player	8	1	0	3	4
	PDF reader	4	0	0	0	4
	Image tool	5	0	0	1	4
	Total	**79**	**39**	**8**	**17**	**15**

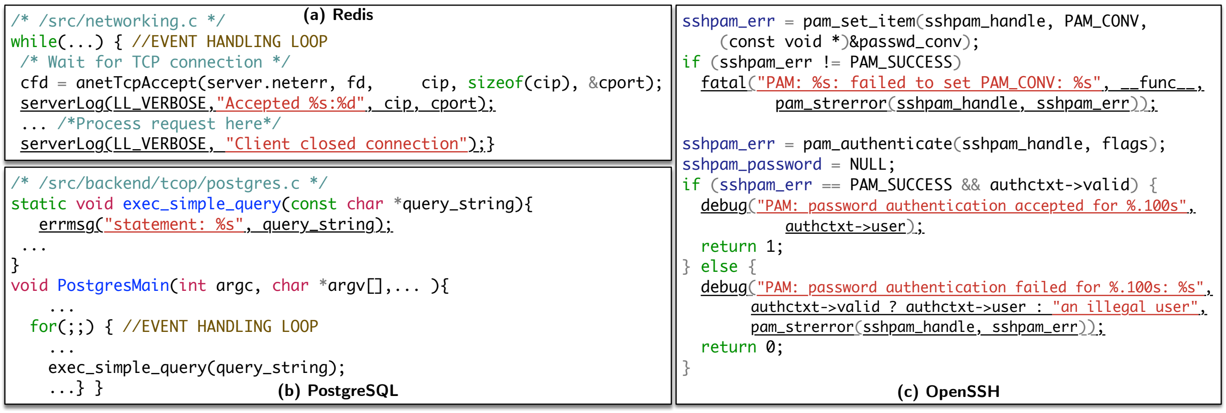
Our study suggests that sufficient logging information is present inside the event-handling loops of long-running applications. This behavior allows us to automatically identify the unit boundaries of those programs. For further evaluation, we only consider the applications shown in Table III. We picked those applications based on their popularity and category. Note that we did not pick any subjects from the category of applications that do not print meaningful log messages for new events. Moreover, GUI applications usually use asynchronous I/O with call backs and such programming model is not currently handled by OmegaLog (described more in §X).

V. Design Overview

A. Definitions

Whole-System Provenance Graph. A graph generated from system-level audit logs, in which the vertices, represent the system subject (*processes*) and system objects (*files* and *socket connection*), while the edges represent a causal dependency event. The edges are usually annotated with a timestamp of the event and the type of event, such as *read* or *execute*.

Causality Analysis. Forensic investigators use the whole-system provenance graph to find the root causes and ramifications of an attack by performing backward and forward causality analysis on the graph, respectively. Given a *symptom* of an attack, an investigator can issue a *backward-tracing* query on the graph; it will find root cause of the attack by traversing the ancestry of the symptom event. The investigator can also issue a *forward-tracing* query that starts from the root cause identified in the previous query and returns all the causally connected events in the progeny of the root cause, explaining the ramifications of the attack.



```
/* /src/networking.c */
while(...) { //EVENT HANDLING LOOP
 /* Wait for TCP connection */
 cfd = anetTcpAccept(server.neterr, fd,      cip, sizeof(cip), &cport);
 serverLog(LL_VERBOSE,"Accepted %s:%d", cip, cport);
 ... /*Process request here*/
 serverLog(LL_VERBOSE, "Client closed connection");}
```

```
/* /src/backend/tcop/postgres.c */
static void exec_simple_query(const char *query_string){
   errmsg("statement: %s", query_string);
  ...
}
void PostgresMain(int argc, char *argv[],... ){
    ...
  for(;;) { //EVENT HANDLING LOOP
    ...
    exec_simple_query(query_string);
    ...} }
```

```
sshpam_err = pam_set_item(sshpam_handle, PAM_CONV,
    (const void *)&passwd_conv);
if (sshpam_err != PAM_SUCCESS)
  fatal("PAM: %s: failed to set PAM_CONV: %s", __func__,
        pam_strerror(sshpam_handle, sshpam_err));

sshpam_err = pam_authenticate(sshpam_handle, flags);
sshpam_password = NULL;
if (sshpam_err == PAM_SUCCESS && authctxt->valid) {
  debug("PAM: password authentication accepted for %.100s",
         authctxt->user);
  return 1;
} else {
  debug("PAM: password authentication failed for %.100s: %s",
       authctxt->valid ? authctxt->user : "an illegal user",
       pam_strerror(sshpam_handle, sshpam_err));
  return 0;
}
```

Fig. 4: Logging behavior of different applications inside the event-handling loop. Underlined code represent log printing statements.

Properties of Causality Analysis. The provenance graph should preserve the following three properties of causality analysis. *Validity* means that the provenance graph describes the correct execution of the system ,i.e., the provenance graph does not add an edge between entities that are not causally related. *Soundness* means that the provenance graph respects the happens-before relationship during backward and forward tracing queries. *Completeness* means that the provenance graph is self-contained and fully explains the relevant event.

B. *Design Goals*

The limitations mentioned in §II on prior work motivated our identification of the following high-level goals:

- **Semantics-Aware.** Our threat investigation solution must be cognizant of the high-level semantic events that occurred within the contexts of each attack-related application.
- **Widely Applicable.** Our solution must be immediately deployable on a broad set of applications commonly found in enterprise environments. Therefore, the solution must not depend on instrumentation or developer annotations. Moreover, our techniques should be agnostic to applications' system architecture and should apply to proprietary software, for which source code is usually not available.
- **Forensically Correct.** Any modifications made to the whole-system provenance graph by our solution must support existing causal analysis queries and preserve the properties of validity, soundness, and completeness.

C. *OmegaLog*

Fig. 5 presents a high-level overview of the OmegaLog system, which requires that both system-level logging and application event logging be enabled. OmegaLog's functionality is divided into three phases: static binary analysis (§VI), runtime (§VII), and investigation (§VIII). In the static analysis phase, (①) OmegaLog first analyzes all application binaries to extract all log message strings (LMSes) that describe event-logging statements in the code, and then uses control flow analysis to identify all possible temporal paths of LMS in different executions of the program. (②) All those LMS control flow paths are stored in a database that is input to a log parser to bootstrap interpretation of application events. At runtime, (③) OmegaLog captures all the application events and augments them with the application's PID/TID and a timestamp of log event through kernel module that intercepts write syscalls. Simultaneously, (④) OmegaLog collects system logs from the underlying whole-system provenance tracker and associates them with the appropriate application events by using the PID/TID as a disambiguator; and store them into a unified log. Upon attack investigation, (⑤) OmegaLog passes that universal log and the LMS control flow paths database to a log parser that partitions associated processes in the whole-system graph by inserting a new *app log vertex*. This vertex is connected to the corresponding partitioned process and annotated with log messages in that particular execution unit of the process. The semantic-aware and execution-partitioned graph is called universal provenance graph (UPG), which is presented to the investigator.

VI. OmegaLog: Static Binary Analysis Phase

The static analysis routine profiles application binaries before their execution. During static analysis, OmegaLog performs several passes over the binary's *control flow graph* (CFG) to identify logging behaviors and generate all possible LMS paths that are possible during execution of that binary. Specifically, we leverage the Angr [53] toolchain to build the CFG, and then introduce new methods to automatically identify logging procedures in the binary (§VI-A). Next, we concretize LMS (§VI-B) using the identified logging procedure, and finally we generate all possible LMS control flow paths that can occur during execution of the binary (§VI-D). Those steps are also shown in Fig. 5.

As highlighted in earlier work [19], binary analysis imposes high costs, especially when symbolic execution and emulation are necessary. In what follows, we describe how OmegaLog avoids prohibitive analysis costs while profiling application-logging behaviors. Although, OmegaLog works on application binaries, for convenience, we explain static analysis procedures by using source code snippets. Algorithm 1 offers a high-level overview of our static analysis routines.

A. *Identifying Logging Procedures*

The ecosystem of event-logging frameworks is diverse and heterogeneous; to overcome the resulting issues, OmegaLog identifies logging procedures in a binary by using two heuristics. 1) Applications use either well-known libraries (e.g., `syslog` [27], `log4c` [6]) or functionally-similar custom routines to produce, store, and flush log messages to a log

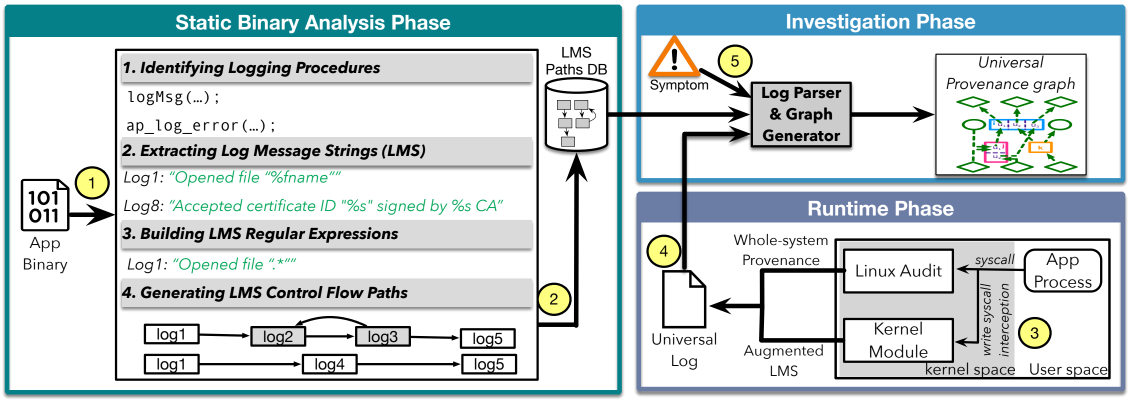

Static Binary Analysis Phase
1. Identifying Logging Procedures
logMsg(…);
ap_log_error(…);
2. Extracting Log Message Strings (LMS)
Log1: "Opened file "%fname""
Log8: "Accepted certificate ID "%s" signed by %s CA"
3. Building LMS Regular Expressions
Log1: "Opened file ".*""
4. Generating LMS Control Flow Paths
log1 log2 log3 log5
log1 log4 log5
LMS Paths DB
App Binary
Investigation Phase
Symptom
Log Parser & Graph Generator
Universal Provenance graph
Runtime Phase
Whole-system Provenance
Linux Audit
syscall
App Process
Kernel Module
write syscall interception
Universal Log
Augmented LMS
kernel space
User space

Fig. 5: OmegaLog architecture overview. During the offline phase, OmegaLog first generates control flow graph and extracts log message strings (LMSes) from application's binary and then contructs LMS control flow paths. During the runtime phase, OmegaLog combines application event logs and audit logs together into universal provenance logs. Finally, during the investigation phase, OmegaLog uses LMS control flow paths to parse universal provenance log into universal provenance graphs.

file. The libraries leverage the I/O procedures of `Libc`, such as `fprintf` or `snprintf`, to write the log messages to disk. OmegaLog can thus identify candidate logging procedures through a backward traversal of the CFG from these procedures call sites. 2) Most applications that create event logs store messages in the `/var/log/` directory by default. Thus, OmegaLog can differentiate log I/O from other I/O based on the file path and consider all the procedures that write to `/var/log/` directory as logging procedures. Combining these two heuristics was sufficient to identify logging behaviors for applications in our evaluation dataset. Nevertheless, OmegaLog also provides an interface that sysadmins can use to add the names of their logging procedures, if the binary does not follow the aforementioned conventions.

B. Extracting Log Message Strings (LMS)

Once we have identified all the logging procedure names in the previous step, we assign a unique identifier for each logging procedure callsite. We need to generate an LMS that describes the format specifier arguments (template) of the log message. This step requires OmegaLog to extract the binary's full control flow graph and perform symbolic execution [35] to extract the values of such arguments. We henceforth refer to this process as *concretization*. However, performing a complete symbolic execution over the binary is a computationally expensive operation that leads to the path explosion problem, especially for applications with complex compile-time optimizations. In fact, while experimenting with the applications listed in Table III, we realized that most applications are compiled with at least the `-O2` compiler optimization level, which greatly complicated the task of CFG extraction and symbolic execution. For example, when we used the Angr toolset, extracting the CFG and performing symbolic execution on the `openssh` server binary quickly exhausted 64 GB of memory on our experimental machine and did not return a conclusive result, even after running for several hours.

To overcome that problem, we first note that our exclusive purpose is to obtain the format specifier arguments for logging function calls; any symbolic execution operation that does not serve this purpose is unnecessary. Therefore, OmegaLog first references the CFG built without symbolic execution (referred to as a `FastCFG` in Angr toolset), which is generated by traversing the binary and using several heuristics to resolve indirect jumps; that approach greatly reduces the CFG computational and memory requirements [53]. Using the `FastCFG`, we identify the basic blocks that contain function calls or jumps to logging procedures, and thus we can focus our attention solely on such blocks. Nevertheless, unlike the full CFG, the `FastCFG` does not retain any state about the binary that would allow OmegaLog to concretize the values of the logging procedures' arguments.

To complete our analysis, we introduce an optimized concretization we refer to as *peephole concretization*. While studying the code of the open-source programs shown in Table III, we observed that for the most part, format specifier arguments to logging procedures are passed either (1) as direct constant strings or (2) through constant variables defined near the procedure call. For example, consider the call to the `debug` logging procedure in the `OpenSSH` application shown in Fig. 4. The LMS we are interested in extracting is the message ``PAM: password authentication accepted for %.100s'' passed directly as a constant to the function call. At the machine instructions level, that observation reflects the fact that LMSes are typically defined within the same basic block that ends with the `call` or `jump` instruction to the address of a logging function, or in a nearby preceding block.

Using peephole concretization, we only need to perform local symbolic execution starting from the basic blocks identified in the previous step, stopping directly after executing the call instruction to the target logging procedure. We show the pseudocode for our peephole concretization step in Algorithm 1. If the symbolic execution task of a given basic block b fails to concretize LMS values, OmegaLog then launches new symbolic execution tasks from each of b's predecessors (referred to as $b.predecessors()$ in Algorithm 1). We refer to the operation of restarting symbolic execution from a basic block's predecessors as *backtracing*. OmegaLog bounds the computational resources employed for the concretization step by halting symbolic execution after performing `maxBackTrace` backtrace operations from a given block b. If symbolic execution fails to produce concretized LMS values after `maxBackTrace` operations, OmegaLog marks the function as *unresolved* and thus produces incomplete LMS paths.

Algorithm 1: Static Binary Analysis

```
Func GetLMS(Binary $\mathcal{B}$, Log functions $\mathcal{F}$)
    /* Overall process to build the LMS paths */
    $g \leftarrow$ AngrGetFastCFG($\mathcal{B}$)
    $C \leftarrow$ ExtractCallSites($g, \mathcal{F}$)
    /* Concretization step */
    $\mathcal{V} \leftarrow$ PeepholeConcretization($g$, $C$)
    /* Building the LMS paths step */
    $\mathcal{G} \leftarrow$ BuildLMSPaths($g, \mathcal{V}, \mathcal{F}$)

Func ExtractCallSites(cfg, $\mathcal{F}$)
    $C \leftarrow \Phi$
    foreach basic block $b \in$ cfg do
        /* Check if the basic block jumps into a logging function */
        if $b$.jump_target_address $\in \mathcal{F}.addresses$ then
            $C \leftarrow C \cup \{b\}$
        end
    end
    return $C$

Func PeepholeConcretization(cfg, call_sites, maxBackTrace)
    $\mathcal{V} \leftarrow \Phi$
    $V \leftarrow \{(b, 0)$ for $b \in$ call_sites$\}$
    while $V \neq \Phi$ do
        $(b, backtrace) \leftarrow V.pop()$
        /* L is of the form $\{$(LMS $\ell$, call_stack $cs$)$\}$ */
        $L \leftarrow$ SymbolicExecution($g$, $v$)
        if $L \neq \Phi$ then
            foreach $(\ell, cs) \in L$ do
                /* Taking care of context sensitivity */
                topBlock $\leftarrow$ cs.top()
                if $(\ell, topBlock) \notin \mathcal{V}$ then
                    $\mathcal{V} \leftarrow \mathcal{V} \cup \{(\ell, topBlock)\}$
                end
            end
        end
        else if backtrace $\leq$ maxBackTrace then
            $V \leftarrow V \cup \{(v, backtrace + 1)$ for $v \in$
              $b.predecessors()\}$
        end
    end
    return $\mathcal{V}$

Func BuildLMSPaths($cfg, \mathcal{V}, \mathcal{F}$)
    /* $\mathcal{E}$ is the set of paths between LMS */
    $\mathcal{E} \leftarrow \Phi$
    foreach $f \in cfg.functions() \backslash \{\mathcal{F}\}$ do
        /* Extract the entry points and external returns */
        entries $\leftarrow f$.entry_points()
        returns $\leftarrow f$.jumps()
        $\mathcal{E} \leftarrow \mathcal{E} \cup$ GetLocalPaths($\mathcal{V}, f$)
    end
```

Our algorithm may yield ambiguous LMS paths in the rare cases in which the function call can have different format specifiers based on the sequence of basic blocks that lead to it (i.e., *context sensitivity*). We address that challenge during the peephole concretization step by recording the call stack that produced each LMS. If two different call stacks produce different LMS for the logging function call, we create a new LMS for each call and then associate it with the topmost basic block on each corresponding function call. That process will guarantee that we do not miss any LMS and that we do not over-approximate the reachability between LMSes when constructing the LMS control flow paths. We note, however, that making format specifiers to logging procedures context-dependent is not a frequently observed programming practice; in fact, we encountered this issue only when processing the `transmission` and `CUPSD` applications.

C. Building LMS Regular Expressions

Finally, once an LMS has been concretized, we can extract a regex that can be used to match event messages at runtime. The resulting regex describes the format specifiers in the LMS that depend on runtime context (e.g., `%s`, `%d`, `%%s`). Each format specifier is replaced with a suitable regex, e.g., "`%d`" with "`[0-9]+`" and "`%s`" with "`.`". For example, one LMS we encounter in `OpenSSH` is

```
PAM: password from user %.12s accepted.
```

After extraction, that yields the regex

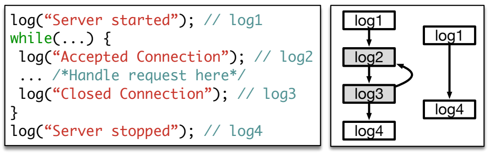
```
PAM: password from user .* accepted.
```

D. Generating LMS Control Flow Paths

After concretizing LMS with selective symbolic execution, OmegaLog can continue to use the `FastCFG` to enumerate the valid sequences of LMS that can appear in a typical lifecycle of the application. Extraction of all the possible paths is not a direct application of depth-first traversal (DFS); DFS renders an under-approximation of the possible paths for the following reasons. (1) The same basic blocks can be called from different callees and thus must be traversed multiple times. (2) Function calls (i.e., `call` instructions) must be matched with their appropriate `return` or `jump` instructions. Finally, (3) the applications we study use an abundance of loops and recursive functions that must be traversed multiple times in order to avoid skipping over loop paths. Instead, our approach addresses (1) and (2) by using caching and temporary nodes, and (3) by using *fixed-point* iterations. Pseudocode for OmegaLog's control flow path building algorithm (BuildLMSPaths) is given in Algorithm 1.

Instead of traversing the full binary's CFG, OmegaLog subdivides the path identification task into several function-local traversals that generate subgraphs for each function in the binary. It then links these subgraphs by following `call` and `return`/`jump` instructions to build the full LMS paths. For each function f in the binary's functions (referred to as $cfg.functions()$ in Algorithm 1), OmegaLog identifies f's entry points, in which control flow passes into the function, and its exit points, in which control flow crosses the f's local body, creating dummy LMS nodes for these points. Then, OmegaLog performs a local traversal of f's subgraph; starting from f's entry points, we traverse the control flow edges between the basic blocks that do not leave f's address space.

Every time OmegaLog encounters a basic block containing an LMS, that block is added to the path, and its outgoing edges are traversed. To accurately capture looping behavior, we perform a *fixed-point* iteration over the loop edges until no further changes occur to the LMS path being built. In other words, we keep traversing the same loop edge until no further LMS paths are detected; we then consider the loop edge to be exhausted and move to the next control flow edge. Finally, to speed up the traversal, OmegaLog caches processed basic blocks so that it needs to only traverse them once if multiple paths coincide. Note that we do not consider any loops that do not contain any syscalls because such loops do not produce audit logs and thus cannot be used for execution partitioning.



```
log("Server started"); // log1
while(...) {
 log("Accepted Connection"); // log2
 ... /*Handle request here*/
 log("Closed Connection"); // log3
}
log("Server stopped"); // log4
```

Fig. 6: On the right, LMS control flow paths representation is shown for the code snippet on the left.

After building the function-local subgraphs, OmegaLog resolves the `call` and `jump` instructions in each of them to complete the full LMS paths. For each function call that is on an LMS path, OmegaLog injects the callee's subgraph into the path by creating links between the caller's basic block and the callee's entry points and between the callee's exit points (`return` blocks and `jump` instructions targeting the caller) and the callee's return basic block. Using that approach, OmegaLog completes the full LMS paths while also handling recursive functions by creating self-cycles. Subsequently, OmegaLog compresses the graph by removing the dummy nodes created by the BUILDLMSPATHS function and merging their fan-in and fan-out edges. The resulting compressed graph will then contain all the detected LMS paths. Fig. 6 shows an example of LMS control flow paths from a code snippet. The code is shown on the left, and the corresponding LMS paths are shown on the right. The backedge from `log3` to `log2` just shows that these logs are inside a loop and can appear more than one time.

LMS control flow paths guide OmegaLog to partition universal provenance log into execution units; however, in some applications printed LMSes in the event-handling loop are not precise enough to partition the loop. For example, Redis event-handling loop shown in Figure 4 prints two LMSes in each iteration of the event-handling loop. The first LMS is printed after the `accept` syscall and if we partition the event-handling loop based on the *both* first and second LMSes, then we will miss that `accept` syscall in the execution unit and only capture syscalls that happened in between two LMSes. However, if we partition the event-handling loop only on the second LMS then we will generate correct execution units because there is no syscall after second LMS in the event-handling loop.

Thus, during LMS control flow paths construction OmegaLog marks all the LMSes present inside the loops that do not have any syscalls before or after in that loop. Such marking helps OmegaLog to make correct execution partitioning of universal provenance log during investigation phase. If there is no such LMS inside the loop then OmegaLog keeps track of either all the syscalls present after the last LMS (loop-ending LMS) in the loop or all the syscalls present before the first LMS (loop-starting LMS) in the loop whichever has least number of syscalls. OmegaLog uses such syscall mappings during investigation phase to make correct execution units.

E. Discussion of Static Analysis Limitations

Our approach is agnostic to the underlying binary analysis tool, but in this work, we used Angr tool, which came with its own set of limitations. Below we discuss these limitations and, in some cases, how we handled them to recover LMS paths.

False Positives & False Negatives. For more information on accuracy and completeness of Angr's recovered CFG, we refer the reader to [53]. In brief, if Angr mistakenly adds an edge that should not be in the CFG of an application, OmegaLog will generate an erroneous LMS path in the LMS path database. However, since that execution path will never happen during runtime, OmegaLog will just ignore this false positive LMS path during UPG construction. In case Angr misses an edge in a CFG, we have implemented Lookahead and Lookback matching (described in §VIII), which handle this case.

Runtime Performance. OmegaLog's static analysis runtime performance was significantly impacted by Angr's performance of symbolic execution. We introduced `PeepholeConcretization` to improve runtime while preserving the accuracy of LMS path recovery. Note that static analysis is a one-time, offline cost: once a binary has been profiled, there is no need to re-analyze it unless it has been changed. On modestly provisioned workstations, that task could even be outsourced to more powerful machines.

Binary Restrictions. First, Angr tool can only work on binaries compiled from C/C++ code. Second, the format modifier argument to a logging procedure should not be built dynamically at runtime as an element of a `struct`, i.e., it should be a constant string. Third, our binary analysis can only recover logging functions that are not inlined. However, we did not encounter inlined logging functions during our evaluation.

VII. OMEGALOG: RUNTIME PHASE

At runtime, OmegaLog performs minimal maintenance of application and whole-system logs; the LMS control flow path models are stored in a database (② in Fig. 5) and are not consulted until an investigation is initiated. The primary runtime challenge for OmegaLog is that of reconciling logs from different layers, which is difficult when considering a flattened event log of concurrent activities in multi-threaded applications. To address that, OmegaLog intercepts all write syscalls on the host using a kernel module and identifies which write syscalls belong to application event logging using heuristics discussed in §VI. After that it only appends the `PID/TID` of the process/thread that emitted the event and along with the timestamp of the event's occurrence to the identified log messages, generating enhanced event log messages.[2] Finally, OmegaLog uses Linux Audit API to add the enhanced event log message to the whole-system provenance log file, which provides an ordering for both application- and system-level events.

VIII. OMEGALOG: INVESTIGATION PHASE

Following an attack, an administrator can query OmegaLog's log parser and graph generator modules (⑤ in Fig. 5) to construct a UPG chronicling the system- and application-layer events related to the intrusion.

[2]Applications that make use of rsyslog facility [8] to write LMS is the one exception to the rule where LMS writing process's PID is not equal to the original application process that produced the LMS. However, in such case we can easily extract the `PID/TID` of original application process because rsyslog use well-defined message format [27] with PID added by default.

Algorithm 2: UPG Construction

```
Inputs   : Universal log file $L_{uni}$;
           Symptom event $e_s$;
           LMS control flow paths $Paths_{lms}$;
Output   : Backward universal provenance graph $G$
Variables: $LMS_{state}$ ← Current state of LMS;
           $eventUnit[Pid]$ ← events in current unit related to $Pid$;
           $endUnit$ ← flag to partition execution into unit;

$endUnit \leftarrow false$
foreach event $e \in L_{uni}$ happened before $e_s$ do
    if IsAppEntry($e$) then
        $LMS_{cand}$ = GetLMSRegex($e$)
        $endUnit$ = MatchLMS($LMS_{cand}$, $Paths_{lms}$,
          $LMS_{state}$, eventUnit[$Pid_e$], $L_{uni}$)
    end
    if $endUnit$ then
        eventUnit[$Pid_e$].add($e$)
        Add all events from eventUnit[$Pid_e$] to $G$
        $endUnit \leftarrow false$
        eventUnit[$Pid_e$] ← null
    end
    else
        eventUnit[$Pid_e$].add($e$)
    end
end
return $G$
```

A. Universal Provenance

Given application binaries, whole-system provenance logs, and application event logs, during the investigation phase, we aim to generate a UPG while preserving the three properties of causality analysis. Algorithm 2 describes how to construct the backward-tracing UPG from the universal log file, specifically a backtrace query from an observable attack *symptom event*; the approach to building forward-trace graph follows naturally from this algorithm and is therefore omitted. When an application event log (an augmented LMS) is encountered while parsing the universal log (Function IsAppEntry in Algorithm 2), it is necessary to match the event to a known LMS for the application in our LMS paths. That matching is performed by the MatchLMS function as described below.

B. LMS State Matching

This procedure entails matching of a given runtime application log entry to its associated LMS in the LMS control flow paths DB. For each log entry in the universal log, the matcher identifies all LMS regexes that are candidate matches. For example, if the event message is

```
02/15/19 sshd [PID]: PAM: password from user root accepted
```

the matcher will look for substring matches, and this will solve the issue of identifying the actual application log entry from the preamble metadata, e.g., "02/15/19 sshd[PID]:".

Ranking LMS. An application log entry may match to multiple LMS regexes in the LMS path DB; this happens because of the prevalence of the %s format specifier in LMS, which can match anything. Therefore, OmegaLog performs a ranking of all the possible candidate matches. We use regex matching to identify the number of non-regex expressions (i.e. constants) in each match. Going back to the example, "`PAM: password from user root accepted`" will match "`PAM: password from user .* accepted`" with a ranking of 5, which is equal to the number of non-regex word matches. Finally, the matcher will return the LMS that has the highest rank or the highest number of non-regex word matches that reflects the true state among the candidate LMSes.

State Machine Matching. Once the candidate LMS (LMS_{cand}) has been identified for an application log entry, OmegaLog attempts to match the LMS_{cand} to a valid LMS path in the database. If this is the first event message, we use a set of heuristics to figure out where we should start from. However, since the matching process can start anywhere in the applications lifetime, usually we have to resort to an exhaustive search over all nodes in the LMS control flow paths. Once we identified the starting node, we keep state in the parser that points to the possible transitions in the LMS paths graph. Upon the next log entry, we search the neighbors of the previous LMS for possible candidate matches. We rank those and return the one with the highest rank, and then advance the parser's state pointer. If OmegaLog cannot find a match in the neighboring LMS states, it advances to the lookahead and lookback matching steps.

Lookahead Matching. When the previous state in the LMS path is known, we may not find a match in a neighboring LMS state because for example (1) the application is running at a different log level, (2) OmegaLog missed the LMS corresponding to the log message in the static analysis phase (for example, the function might be inlined, or we could not concretize its values), or (3) the log message is coming from a third-party library. We therefore start looking deeper into the reachable states from the current parser state. If we find multiple candidates, we again rank them and return the one with the highest rank. If we do not find one, we then keep increasing the lookahead up until we hit a certain threshold that can be set at runtime. If we find a match, we move the parser to that state and repeat until we match a candidate LMS at the end of LMS control flow path. At that point, we set the $endUnit$ flag to true.

As described in §VI, in certain cases LMS may not be able to correctly partition the execution because there are syscalls after the loop-ending LMS or syscalls before loop-starting LMS. During offline analysis, OmegaLog marks such LMS and keep track of any syscalls that we should expect during runtime. If we observe such case during state matching process, we match those syscalls besides matching LMS and add those syscalls into the execution unit. Function MatchLMS in Algorithm 2 also handles such cases and appropriately sets the $endUnit$ flag to true.

Lookback Matching. If the above lookahead step fails because we cannot find the end state in the LMS path, then we first try to search the heads of loops that are of the form (`while(1)`, `for(;;)`) in the LMS control flow path. The intuition behind loop head identification step is that we might have hit the start of a new execution unit and thus we would need to restart from a new stage. If this fails, then we perform an exhaustive search of LMS that can happen before the current state in the LMS paths using the same intuition mentioned before. If in either case, we get a match we set the $endUnit$ flag to true. Note that fallback matching allows us to generate execution units even if we have only one log message at start or end of the loop, because we use the next execution unit's log message to partition the current execution unit.

IX. Evaluation

In this section, we evaluate OmegaLog to answer the following research questions (RQs):

RQ1: What is the cost of OmegaLog's static analysis routines when extracting logging information from binaries?
RQ2: How complete is our binary analysis in terms of finding all the LMSes in an application?
RQ3: What time and space overheads does OmegaLog impose at runtime, relative to a typical logging baseline?
RQ4: Is the universal provenance graph causally correct?
RQ5: How effective is OmegaLog at reconstructing attacks, relative to a typical causal analysis baseline?

Experimental Setup. We evaluated our approach against 18 real-world applications. We selected these applications from our pool of applications discussed in §IV based on popularity and category. Moreover, most of these applications were used in the evaluation of prior work on provenance tracking [42], [41], [38], [39]. For each program, we profile two verbosity levels, `INFO` and `DEBUG`, when considering the above research questions. Workloads were generated for the applications in our dataset using the standard benchmarking tools such as Apache Benchmark `ab` [1] and FTPbench [2].

All tests were conducted on a server-class machine with an Intel Core(TM) i7-6700 CPU @ 3.40 GHz and 32 GB of memory, running Ubuntu 16.04. To collect whole-system provenance logs we used Linux Audit Module[3] with the following syscall ruleset: clone, close, creat, dup, dup2, dup3, execve, exit, exit_group, fork, open, openat, rename, renameat, unlink, unlinkat, vfork, connect, accept, accept4, bind. OmegaLog's offline algorithm accepts a single configuration parameter, `maxBackTrace`, that sets the maximum depth of symbolic execution operations. After experimenting with that parameter, we found that a value of 5 was enough to guarantee >95% coverage for 12 of the 18 applications we analyzed, as we discuss in the following section. In fact, our experiments have shown that we did not need to increase the symbolic execution depth beyond 3 basic blocks.

A. *Static Analysis Performance*

Table III shows how much time it takes to identify and concretize LMS from application binaries and subsequently generate LMS path models (Algorithm 1). We first note that the overhead of building the LMS paths (LMSPs) is reasonable for a one-time cost, taking 1–8 seconds for most applications, with a maximum of 3 minutes for `PostgreSQL`; the increase for `PostgreSQL` is due to the larger number of LMS paths captured by OmegaLog. On the other hand, average time to generate an LMS column shows the time to generate the `FastCFG` and concretize the LMS dominates OmegaLog's static analysis tasks, ranging from a minimum of a minute and a half (`Transmission`) to a maximum of 1.2 hours (`PostgreSQL`). Those two tasks are in fact highly dependent on Angr's raw performance. As acknowledged by the Angr tool developers [11], the static analyzer's performance is handicapped because it is written in the Python language with no official support for parallel execution.

[3]We make use of the Linux Audit framework in our implementation. However, our results are generalizable to other system logs, such as Windows ETW [4] and FreeBSD DTrace [3].

Our results show no direct relationship between the size of the binary of the application being analyzed and the overall analysis time. By inspecting the applications' source code, we found that OmegaLog's performance is more informed by the structure of the code and the logging procedures. We can see intuitively that as the number of found callsites increases, the number of peephole symbolic execution steps needed also increases, thus increasing the total concretization time. However, that does not generalize to all the applications; for example, the analysis of `NGINX` (2044 KB binary) completed in 13 minutes concretizing 925 LMS while `Lighttpd` (1212 KB, almost half of `NGINX`'s binary size) required 32 minutes concretizing only 358 LMSes.

Upon closer investigation of `Lighttpd`'s source code, we found that format specifiers (and thus LMS) were often passed as structure members rather than as constant strings (which form the majority of LMS in the case of `NGINX`). That will trigger the backtracing behavior of the PeepholeConcretization algorithm in an attempt to concretize the values of the struct members, thus increasing the cost of the symbolic execution operations performed by Angr. Below we show sample code snippets from `Lighttpd` that trigger such behavior:

```
/* log function signature: /src/log.c */
int log_error_write(server *srv, const char *filename, unsigned int line,
        const char *fmt /* our tool looks for fmt */, ...)
/* format specifier passed as struct member: /src/config-glue.c */
if (con->conf.log_condition_handling) {
  log_error_write(srv, __FILE__, __LINE__, "dss",
      dc->context_ndx, /* the fmt argument */
       "(cached) result:",
      cond_result_to_string(caches[dc->context_ndx].result)); }
```

The cases of `Lighttpd` and `NGINX` highlight the unpredictability of runtime of OmegaLog's static analysis when only the binary size or the number of identified callsites is considered. Rather, the runtime depends on the structure of the code and the anatomy of the calls to the log functions.

B. *Static Analysis Completeness*

We report on OmegaLog's coverage ratio, which represents the percent of concretized LMS relative to the count of identified callsites to logging procedures. As shown in the last column of Table III, OmegaLog's coverage is >95% for all the applications except `PostgreSQL`, `Transmission`, and `wget`. We disregard `thttpd` since it presents a small sample size in terms of LMS where OmegaLog only missed 1 LMS during concretization. That speaks to OmegaLog's ability to consistently obtain most of the required LMSes and build their corresponding LMS control flow paths. We show in our experiments, this coverage ratio is sufficient to enable OmegaLog to perform execution partitioning and aid the investigation process without loss of precision. In addition, when LMSes are missing, OmegaLog's runtime parser can handle missing log messages through lookahead and lookback techniques. If OmegaLog fails to concretize an LMS, it is a reflection of the symbolic execution task's ability to resolve a format specifier for a logging procedure.

TABLE III: Application logging behavior and performance results of OmegaLog's static analysis phase. EHL stands for event handling loop; IN+DE means that both INFO and DEBUG verbosity levels are present in the loop; LMSPs: Log message string paths; Callsites are identified log statements; and "Cov. %" denotes coverage percentage which is the percentage of concretized LMS to callsites.

Program	Binary	Log Level	Avg. Time (sec)		Number of		Completeness	
	Size (kB)	inside EHL	LMS	LMSPs	LMS	LMSPs	Callsites	Cov. %
Squid	64,250	IN+DE	831	46	64	157,829	70	91
PostgreSQL	22,299	IN+DE	3,880	258	3,530	4,713,072	5,529	64
Redis	8,296	INFO	495	7	375	34,690	394	95
HAProxy	4,095	IN+DE	144	4	53	13,113	56	95
ntpd	3,503	INFO	2,602	4	490	10,314	518	95
OpenSSH	2,959	IN+DE	734	4	845	11,422	869	97
NGINX	2,044	IN+DE	775	11	923	8,463	925	100
Httpd	1,473	IN+DE	99	2	211	3,910	211	100
Proftpd	1,392	IN+DE	201	4	717	9,899	718	100
Lighttpd	1,212	INFO	1,906	2	349	5,304	358	97
CUPSD	1,210	DEBUG	1,426	3	531	4,927	531	100
yafc	1,007	IN+DE	88	2	57	3,183	60	95
Transmission	930	IN+DE	102	2	178	5,560	227	78
Postfix	900	INFO	97	3	96	2,636	98	98
memcached	673	IN+DE	193	7	64	19,510	69	93
wget	559	INFO	200	3	84	3,923	275	31
thttpd	105	N/A	157	8	4	14,847	5	80
skod	47	N/A	12	0	25	115	25	100

To better understand the conditions of OmegaLog's performance, we analyzed the source code of `PostgreSQL`, `Transmission`, and `wget` (64%, 78%, and 31% coverage, respectively). Our analysis revealed that in all three cases, symbolic execution was failing for logging procedures that use GNU's `gettext` for internalization (called using the "_" operator), as shown below:

```
/* Below code from Transmission: /libtransmission/rpc-server.c */
tr_logAddNamedError(MY_NAME, _("Couldn't find settings key \"%s\""), str);
/* Below code from wget: /src/convert.c */
 logprintf  (LOG_VERBOSE, _("Converting links in %s... "), file);
/* Below code from PostGreSQL: /src/backend/commands/tablecmds.c */
default:
        /* shouldn't get here, add all necessary cases above */
        msg = _("\"%s\" is of the wrong type");
        break; }
```

Since `gettext` is loaded dynamically as a shared library, Angr is not able to handle it appropriately during symbolic execution and cannot extract its return value, thus causing the failure of LMS extraction during the peephole concretization step. To confirm our findings, we reran the static analysis for `wget` and `Transmission` with the calls to `gettext` removed and were able to achieve coverage of 98.18% and 96.03%, respectively. One approach to addressing that issue using Angr would be to add hooks for all of `gettext`'s methods and return the arguments without changes. That would in turn provide Angr's symbolic execution engine with the arguments for concretization. We plan to address the issue in future work.

C. Runtime & Space Overhead

We measured the runtime overhead of OmegaLog compared to a baseline of application event log collection at the `INFO` and `DEBUG` verbosity with Linux Audit running. We turn on `INFO` and `DEBUG` level based on the application's logging behaviour required for execution partitioning. As shown in Fig. 7, OmegaLog's average runtime overhead was 4% for all the applications that had logging inside the event-handling loop. Some applications, such as Memcached and Proftpd,

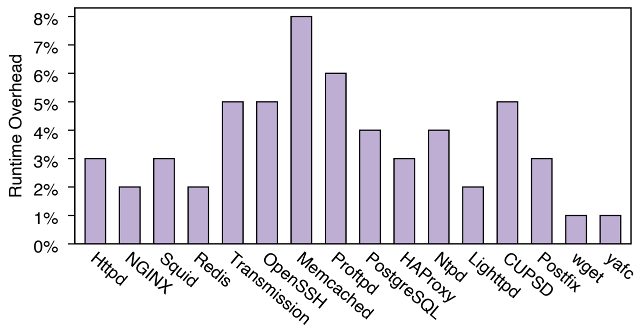


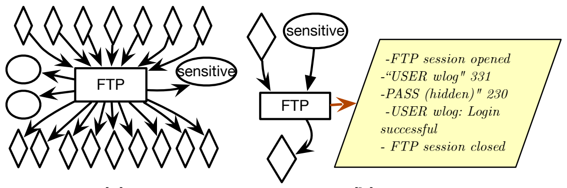
Fig. 7: Runtime overhead for each applications in our dataset that has logging statement in the event-handling loop.

exhibit high overhead because they are write-intensive applications; since OmegaLog intercepts every write syscall to disambiguate `PID/TID`, we expect to see higher runtime costs here. However, we argue that the benefits of OmegaLog for forensic analysis already justify the cost, and will consider alternative methods for process disambiguation in future work.

OmegaLog incurs space overhead because it records the `PID/TID` and timestamp for each application event message so that it can match the event to the appropriate system-layer task. At most, that addition requires 12 bytes per LMS entry. Our experiments confirm that the cost is negligible during typical use. For example, each unenhanced event message in NGINX is approximately 8.6 kB. If an NGINX server received 1 million requests per day and each request generated one event, the original event log would be 860 MB and OmegaLog would add just 12 MB to that total, around 1% space overhead.

D. Correctness of Universal Provenance Graph

OmegaLog modifies the whole-system provenance graph by adding *app log vertices* to generate semantic-aware and execution-partitioned universal provenance graphs. We describe three causal graph properties in §V that the universal provenance graph needs to preserve for correct forensic



```
-FTP session opened
-"USER wlog" 331
-PASS (hidden)" 230
 -USER wlog: Login
successful
- FTP session closed
```

Fig. 8: Information theft attack scenario. (a) Provenance graph generated using a traditional solution, which led to a dependency explosion problem with no semantic information. (b) Concise provenance graph generated using OmegaLog with semantic information.

analysis. To ensure the *Validity* property, we augment LMS with PID/TID information along with timestamps during the runtime phase so that we can causally associate application log vertices with process vertices in the whole-system provenance graph. To ensure the *Soundness* property, OmegaLog augments LMS with timestamps from the same system clock as the whole-system provenance graph and uses this timestamp as an annotation from process vertices to application log vertices. That edge annotation allows OmegaLog to respect the happens-before relationships while doing backward and forward tracing on the graph. Finally, since universal provenance graphs do not remove any causally connected vertices (besides false provenance introduced by dependency explosion in a manner consistent with previous work [39], [42]) we achieve the property of *Completeness*.

E. Attack Investigation

We now evaluate OmegaLog's ability to aid in a typical attack investigation. To do so, we consider two additional scenarios as case studies. For each attack scenario, we manually verified its UPG to check that it preserved the three causality analysis properties that we discussed in §V. We note that the result that we presented in the motivating scenario (§II) was also procedurally generated using OmegaLog.

1) Information Theft Attack: An administrator made a mistake when configuring an FTP server, allowing users to read and transfer sensitive files from the server's directories. The issue was identified after several days, but the administrator now needs to identify which files were leaked, if any, to ensure that company secrets are safe. Using the sensitive files as a *symptom*, the administrator runs a backtrace query.

Fig. 8(a) shows the attack investigation results using a traditional causal analysis solution, which confirms that the sensitive file was accessed. However, because of dependency explosion, it is impossible to determine who accessed the file and where it was transferred to. In contrast, Fig. 8(b) shows the universal provenance graph produced by OmegaLog. OmegaLog was able to partition the server into individual units of work based on event log analysis, removing the dependency explosion and identifying an IP address to which the sensitive file was downloaded. However, that information may not prove precise enough to attribute the attack to a particular employee or remote agent; fortunately, because OmegaLog was able to associate the causal graph with event messages from the FTP server, the administrator is able to attribute the theft to a specific set of user credentials. Note that while existing execution-partitioning systems such as ProTracer [42] and BEEP [39] could eliminate dependency explosion in this scenario, they would not enable user-level attribution of the attack.

2) Phishing Email: An employee uses the Mutt email client to send and receive personal emails on a BYOD workstation. One day, the employee receives a phishing email that offers a torrent for downloading a blockbuster movie. Employee opens the email, downloads the attached `.torrent` file. After that employee, used Transmission application to download the purported movie torrent file. Finally, employee opens the downloaded movie file but the file is actually malware that establishes a backdoor on the machine.

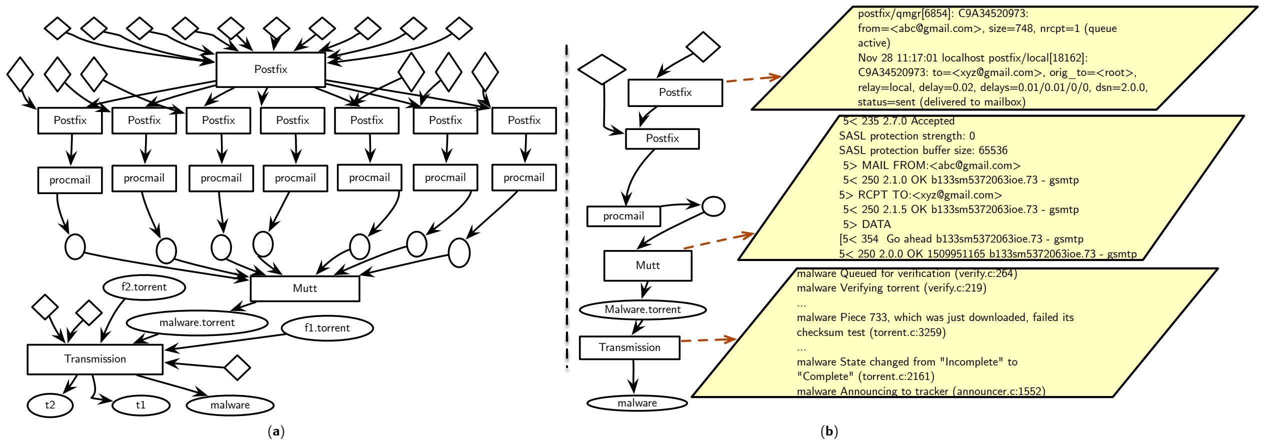
An administrator later notices that a suspicious program is running on the workstation and initiates forensic analysis to identify its origin. Fig. 9(a) shows the causal graph that the investigation would yield based on simple `auditd`. As can be seen in the graph, the employee has actually opened three `.torrent` files with transmission-daemon. It is impossible to determine which `.torrent` input file led to the malware download. Even if out-of-band knowledge is used to identify the malicious torrent, the administrator will still be unable to trace back to the phishing email.

Fig. 9(b) shows the UPG produced by OmegaLog. Because OmegaLog successfully partitioned the Postfix and Transmission processes, the graph does not exhibit dependency explosion, making it easy to trace from the suspicious process back to the phishing email. Further, the OmegaLog graph provides additional documentation of application semantics, such as the email address of the sender, which may help the administrator correlate this attack with other intrusions. Such evidence cannot be provided by existing provenance trackers.

X. Discussion & Limitations

Control flow integrity (CFI) assumption is a limitation of OmegaLog; in fact, this is a big problem for almost the entirety of recent work in provenance-based forensic analysis space [16], [17], [26], [30], [31], [32], [37], [40], [39], [41], [38], [42], [28], [45]. OmegaLog assumes CFI of program execution because violation of CFI makes it impossible to give assertions about the trace logs of program execution. For example, execution units emitted from BEEP system [39] can not be trusted because an attacker can hijack control flow of the running application to emit misleading boundaries, confusing the investigator. Moreover, violations of CFI assumption enables post-mortem tampering of audit logs or even runtime control flow bending that causes misleading application event records to be emitted. Even though the main focus of our study is improving forensic analysis and solving CFI problem is ultimately an orthogonal problem to our study but we envision that future work on provenance will cater CFI violation problem for accurate forensic analysis.

Provided that an underlying binary analysis tool has generated a reasonably accurate CFG, there are two considerations when one is evaluating the generality of OmegaLog. The first is whether or not the application being profiled includes logging events at key positions in the CFG such as the event handling loop. Our survey in §IV demonstrates that this is the case for mature open source client-server applications. The second consideration is whether the event logging statements are correctly identified and extracted by OmegaLog. Our



```
postfix/qmgr[6854]: C9A34520973:
from=<abc@gmail.com>, size=748, nrcpt=1 (queue
active)
Nov 28 11:17:01 localhost postfix/local[18162]:
C9A34520973: to=<xyz@gmail.com>, orig_to=<root>,
relay=local, delay=0.02, delays=0.01/0.01/0/0, dsn=2.0.0,
status=sent (delivered to mailbox)
```

```
 5< 235 2.7.0 Accepted
SASL protection strength: 0
SASL protection buffer size: 65536
 5> MAIL FROM:<abc@gmail.com>
 5< 250 2.1.0 OK b133sm5372063ioe.73 - gsmtp
5> RCPT TO:<xyz@gmail.com>
 5< 250 2.1.5 OK b133sm5372063ioe.73 - gsmtp
 5> DATA
[5< 354  Go ahead b133sm5372063ioe.73 - gsmtp
5< 250 2.0.0 OK 1509951165 b133sm5372063ioe.73 - gsmtp
```

```
malware Queued for verification (verify.c:264)
malware Verifying torrent (verify.c:219)
...
malware Piece 733, which was just downloaded, failed its
checksum test (torrent.c:3259)
...
malware State changed from "Incomplete" to
"Complete" (torrent.c:2161)
malware Announcing to tracker (announcer.c:1552)
```

Fig. 9: Phishing email attack scenario. **(a)** Attack provenance graph generated by traditional solutions. **(b)** Semantic-aware and execution-partitioned provenance graph generated by OmegaLog.

evaluation (§IX) demonstrated that we are able to identify log statements in all the profiled applications based on our heuristics for event-logging extraction.

OmegaLog assumes at least one log message printed in the event-handling loop to partition execution. OmegaLog uses ordered log messages in the universal provenance logs as a way to partition syscalls and make unit boundaries. Such an assumption only works for the applications that use synchronous I/O programming model. For instance, if an application is using asynchronous I/O and only prints one log message at the end of the event-handling loop then concurrent requests will generate multiple syscalls without immediately printing log message at the end of each request. In such case, OmegaLog will not be able to correctly partition each request. One approach to solve this problem is to generate complete syscall mapping along with LMS paths model inside the event-handling loop during offline analysis and use this mapping to divide execution. We leave that as our future work.

Malware binaries may not produce any of the application logs that are required for execution partitioning. In that case, OmegaLog treats the whole malware execution as one unit and does not provide execution partitioning. That is acceptable since every output and input event from malware execution is important in forensic analysis.

XI. Related Work

In §II, we described several shortcomings of existing provenance-tracking systems that OmegaLog addresses. Here we provide additional discussion of related work.

Application Log Analysis. Application logs contain a wealth of information that can be useful in aiding software system maintenance, and thus become an important data source for postmortem analysis [48], anomaly detection [24], [59], [60], program verification [52], and security monitoring [46]. Existing guidelines and practices [66], [34], [25], [21], [49] indicate the importance of well-designed log messages in failure diagnosis. Xu et al. [59] analyzed console logs to learn common patterns by using machine learning and to detect abnormal log patterns at runtime. SherLog [61] used application source code and runtime error log to infer what must or may have happened during a failed run and provide detailed *post mortem* analysis of the error. Similarly, LogEnhancer [62] and LogAdvisor [66] automatically improves existing log messages and provides suggestions on where to log in the code in order to aid in future post-failure debugging. HERCULE [50] uses expert-written log parsers and rules to first extract log fields such as IP addresses and then correlate log entries across application logs based on these fields. Unlike OmegaLog, HERCULE's rule-based approach does not accurately capture causality across applications that use th whole-system layer and that can ultimately undermine forensic investigations.

Several log analysis systems [56], [63], [44] have been proposed to reconstruct behaviour of applications running on Android OS. Unlike OmegaLog, these existing systems are not transparent as they either require code instrumentation or an emulator to collect logs for analysis. DroidHolmes [44] and CopperDroid [56] are single-layer log analysis systems while OmegaLog is a multi-layer log analysis system. DroidForensic [63] collects logs from different layers for forensic analysis; however, in its case, the onus is on the user to correlate and combine logs from different layers. On the other hand, OmegaLog integrates logs from different layers without user-involvement using program-analysis techniques.

Application Log Parsing. Automated log parsing allows developers and support engineers to extract structured data from unstructured log messages for subsequent analysis. Many open source tools, such as Logstash [7] and Rsyslog [8] and commercial tools such as, VMWare LogInsight [10] and Splunk [9] provide built-in log-parsing modules/recipes for popular applications such as MySQL and Apache httpd; that allows users to automatically extract useful information, such as PID, hostname, and filenames from log messages. For custom parsing of log messages, those tools provide easy-to-use, regex-based languages to define parsers.

Distributed System Tracing. End-to-end tracing is required in distributed systems to enable comprehensive profiling. Existing tools, such as Dtrace [18], Dapper [54], X-trace [23], MagPie [15], Fay [22], and PivotTracing [43] instrument the underlying application to log key metrics at run time. On the other hand, lprof [65] and Stitch [64] allow users to profile a single request without instrumenting any distributed application. lprof uses static analysis to find identifiers that can distinguish output logs of different requests. However, lprof only correlate logs from the same distributed application. On the other hand, Stitch requires certain identifiers in the log messages in order to correlate log messages across different distributed applications. Finally, both systems capture mere correlations instead of true causality between application logs and that can reduce the accuracy of attack reconstruction.

XII. Conclusion

In this work, we introduce OmegaLog an end-to-end provenance-tracking system that uses the notion of universal provenance to solve the semantic gap and dependency explosion problem that currently exist in causality analysis frameworks. Universal provenance combines whole-system audit logs and application event logs while preserving the correctness of causality analysis. OmegaLog leverages static binary analysis to parse and interpret the application event logs and generates semantic-aware and execution-partitioned provenance graphs. We implemented our prototype using the Angr binary analysis framework and the Linux Audit Module. Evaluation on real-world attack scenarios shows that OmegaLog's generated graphs are concise and rich with semantic information, compared to the state-of-the-art.

Acknowledgment

We thank our shepherd, Yonghwi Kwon, and the anonymous reviewers for their comments and suggestions. We also thank Akul Goyal and Riccardo Paccagnella for feedback on early drafts of this paper. Wajih Ul Hassan was partially supported by the Sohaib & Sara Abbasi Fellowship and the Symantec Graduate Fellowship. This work was supported in part by the National Science Foundation under contracts CNS-16-57534 and CNS-17-50024. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsors.

References

[1] "Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] "Benchmark for ftp servers," https://pypi.python.org/pypi/ftpbench.

[3] "DTrace," https://www.freebsd.org/doc/handbook/dtrace.html.

[4] "Event tracing," https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal.

[5] "The Linux audit daemon," https://linux.die.net/man/8/auditd.

[6] "Log4c : Logging for C library," http://log4c.sourceforge.net/.

[7] "Logstash: Collect, Parse, Transform Logs," https://www.elastic.co/products/logstash.

[8] "Rsyslogd," http://man7.org/linux/man-pages/man8/rsyslogd.8.html.

[9] "Splunk Log Management," https://www.splunk.com/en_us/central-log-management.html.

[10] "VMware vCenter Log Insight," http://www.vmware.com/ca/en/products/vcenter-log-insight.

[11] "Speed considerations," https://github.com/angr/angr-doc/blob/master/docs/speed.md.

[12] "Equifax says cyberattack may have affected 143 million in the U.S." https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html.

[13] "Inside the cyberattack that shocked the US government," ttps://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/.

[14] "Target Missed Warnings in Epic Hack of Credit Card Data," https://bloom.bg/2KjElxM.

[15] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling." in *OSDI*, 2004.

[16] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, "Transparent web service auditing via network provenance functions," in *WWW*, 2017.

[17] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the Linux kernel," in *USENIX Security*, 2015.

[18] B. Cantrill, M. W. Shapiro, A. H. Leventhal *et al.*, "Dynamic instrumentation of production systems." in *USENIX ATC*, 2004.

[19] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *USENIX Security*, 2014.

[20] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging." in *USENIX Security Symposium*, 2009.

[21] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *USENIX ATC*, 2015.

[22] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz, "Fay: Extensible distributed tracing from kernels to clusters," *ACM Trans. Comput. Syst.*, vol. 30, 2012.

[23] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *NSDI*. USENIX, 2007.

[24] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM*. IEEE, 2009.

[25] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *ICSE Companion*. ACM, 2014.

[26] A. Gehani and D. Tariq, "SPADE: Support for provenance auditing in distributed environments," in *Middleware*, 2012.

[27] R. Gerhards, "The syslog protocol," Internet Requests for Comments, RFC 5424, 2009.

[28] E. Gessiou, V. Pappas, E. Athanasopoulos, A. D. Keromytis, and S. Ioannidis, "Towards a universal data provenance framework using dynamic instrumentation," in *Information Security and Privacy Research*, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds. Springer Berlin Heidelberg, 2012.

[29] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting threat alert fatigue with automated provenance triage," in *NDSS*, 2019.

[30] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *NDSS*, 2018.

[31] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *USENIX Security*, 2017.

[32] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *CCS*. ACM, 2017.

[33] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1978.

[34] B. W. Kernighan and R. Pike, *The practice of programming*. Addison-Wesley Professional, 1999.

[35] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, 1976.

[36] S. T. King and P. M. Chen, "Backtracking intrusions," in *SOSP*. ACM, 2003.

[37] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality." in *NDSS*, 2005.

[38] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie *et al.*, "MCI: Modeling-based causality inference in audit logging for attack investigation," in *NDSS*, 2018.

[39] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *NDSS*, 2013.

[40] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for Windows," in *ACSAC*. ACM, 2015.

[41] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *USENIX Security*, 2017.

[42] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting," in *NDSS*, 2016.

[43] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *SOSP*. ACM, 2015.

[44] Z. Meng, Y. Xiong, W. Huang, F. Miao, T. Jung, and J. Huang, "Divide and conquer: recovering contextual information of behaviors in android apps around limited-quantity audit logs," in *ICSE: Companion Proceedings*. IEEE Press, 2019.

[45] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-time APT detection through correlation of suspicious information flows," in *Symposium on Security and Privacy*. IEEE, 2019.

[46] M. Montanari, J. H. Huh, D. Dagit, R. B. Bobba, and R. H. Campbell, "Evidence of log integrity in policy-based security monitoring," in *DSN*. IEEE, 2012.

[47] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in provenance systems," in *ATC*, 2009.

[48] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Communications of the ACM*, vol. 55, no. 2, 2012.

[49] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *ICSE*. IEEE, 2015.

[50] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, "HERCULE: Attack story reconstruction via community discovery on correlated log graph," in *ACSAC*. ACM, 2016.

[51] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting high-fidelity whole-system provenance," in *ACSAC*, 2012.

[52] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *ICSE*. IEEE Press, 2013.

[53] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[54] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc, Tech. Rep., 2010.

[55] M. Stamatogiannakis, P. Groth, and H. Bos, "Looking inside the black-box: Capturing data provenance using dynamic instrumentation," in *IPAW*. Springer-Verlag New York, Inc., 2015.

[56] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of android malware behaviors." in *NDSS*, 2015.

[57] D. Tariq, M. Ali, and A. Gehani, "Towards automated collection of application-level data provenance," in *TaPP*. USENIX, 2012.

[58] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *NDSS*, 2018.

[59] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *SOSP*. ACM, 2009.

[60] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, 2016.

[61] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: Error diagnosis by connecting clues from run-time logs," in *ASPLOS*. ACM, 2010.

[62] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM TOCS*, vol. 30, no. 1, 2012.

[63] X. Yuan, O. Setayeshfar, H. Yan, P. Panage, X. Wei, and K. H. Lee, "Droidforensics: Accurate reconstruction of android attacks via multi-layer forensic logging," in *AsiaCCS*. ACM, 2017.

[64] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle." in *OSDI*, 2016.

[65] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems," in *OSDI*, 2014.

[66] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to Log: Helping developers make informed logging decisions," in *ICSE*, 2015.