ALchemist: Fusing Application and Audit Logs for Precise Attack Provenance without Instrumentation

Le Yu*, Shiqing Ma†, Zhuo Zhang*, Guanhong Tao*, Xiangyu Zhang*, Dongyan Xu*,
Vincent E. Urias‡, Han Wei Lin‡, Gabriela Ciocarlie§, Vinod Yegneswaran§ and Ashish Gehani§
*Purdue University; †Rutgers University; ‡Sandia National Laboratories; §SRI International
*{yu759, zhan3299, taog, xyzhang, dxu}@cs.purdue.edu,
†sm2283@cs.rutgers.edu, ‡{veuria, hwlin}@sandia.gov, §{gabriela, vinod, gehani}@csl.sri.com

***Abstract*—Cyber-attacks are becoming more persistent and complex. Most state-of-the-art attack forensics techniques either require annotating and instrumenting software applications or rely on high quality execution profiling to serve as the basis for anomaly detection. We propose a novel attack forensics technique ALchemist. It is based on the observations that built-in application logs provide critical high-level semantics and audit logs provide low-level fine-grained information; and the two share a lot of common elements. ALchemist is hence a log fusion technique that couples application logs and audit logs to derive critical attack information invisible in either log. It is based on a relational reasoning engine Datalog and features the capabilities of inferring new relations such as the task structure of execution (e.g., tabs in *firefox*), especially in the presence of complex asynchronous execution models, and high-level dependencies between log events. Our evaluation on 15 popular applications including *firefox*, *Chromium*, and *OpenOffice*, and 14 APT attacks from the literature demonstrates that although ALchemist does not require instrumentation, it is highly effective in partitioning execution to autonomous tasks (in order to avoid bogus dependencies) and deriving precise attack provenance graphs, with very small overhead. It also outperforms NoDoze and OmegaLog, two state-of-the-art techniques that do not require instrumentation.**

I. Introduction

Advanced Persistent Threat (APT) is a complex form of threat that contains multiple phases and targets specific organization or institute [4]. A popular method for attack investigation is to perform dependency analysis on system audit logs to reconstruct attack provenance. In [41], [42], [14], researchers analyzed dependencies among system objects (e.g., files and sockets) and subjects (i.e., processes) using system call logs. However, these approaches have limitations in analyzing attacks that involve long running processes (e.g., browsers). In particular, they all assume that an output operation depends on all the prior input operations in the same process, introducing substantial false dependencies. For example, the write to a downloaded file by *firefox* is considered dependent on all the websites *firefox* has visited before the download, which is very imprecise. This is known as the *dependency explosion problem*.

To solve this problem, researchers proposed using program analysis to enhance the collected log and partition long running processes into *execution units/tasks* [53], [45]. Each unit/task is an autonomous portion of the whole execution such as a tab in *firefox*. An output operation is considered dependent on *all* the preceding input operations *within the same unit*. Doing so, they can preclude a lot of false dependencies. Researchers have demonstrated the effectiveness of these unit partitioning based techniques, which yield very few dependence false positives and false negatives [53], [45]. However, these approaches require third-party instrumentation, which may not be acceptable in enterprise environments. In practice, software providers (e.g., Microsoft) provide maintenance services to their customers only when the integrity of their software is guaranteed. As instrumentation entails changing software (by some third party), it shifts the responsibility of maintaining the correctness of software from its original producer to the third party, which is undesirable. In fact, many vendors provide mechanisms to proactively prevent their software from being instrumented such as the Kernel Patch Protection by Microsoft [10]. In addition, these techniques record low-level events such as memory accesses such that the entailed overhead, especially the space overhead, is high [45]. Another line of work does not require third-party instrumentation. Instead, it tries to solve the dependency explosion problem by pruning graphs with heuristics such as prioritizing low frequency events [49], [31]. Depending on the quality of execution profile used to establish the baseline, these methods may flag rarely seen benign operations as malicious and attack steps leveraging benign software/IPs as normal (e.g., an APT attack using phishing pages on Github may evade such methods due to the frequent visits to Github). In addition, asynchronous and background behaviors pose significant challenges to learning based methods due to their non-deterministic nature.

Our goal is to develop a new attack investigation technique that can achieve the same accuracy as instrumentation based methods without requiring instrumentation. We observe that many widely used applications, especially those that are long-running and tend to cause dependence explosion, have well-designed built-in logs. These application logs record important events with application-specific semantics (e.g., switching-to/opening a tab in *firefox*). As such, they can be parsed and analyzed to reconstruct the unit structure of an execution, which is critical to precise dependence analysis as shown by the literature [53], [33]. On the other hand, the low level audit log provides fine-grained information that is invisible in application logs and typically corresponds to background activities (e.g., using JavaScript for background network communication). Therefore, we propose a novel log fusion technique,

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24445
www.ndss-symposium.org

ALchemist, that couples application logs and the audit log, to produce precise attack provenance. It does not require any instrumentation and the entailed overhead is low compared to existing techniques. During attack investigation, ALchemist first normalizes the raw application logs and the audit log to a canonical form such that their correlations can be inferred. The canonical form is general such that it can express all the execution models of common applications, including those having complex asynchronous/background behaviors. The canonical log entries are loaded into a Datalog engine [38] to derive new relations based on a set of pre-defined rules, which we call the *log fusion rules*. Precise dependency graphs can be easily constructed from the inferred relations. In summary, we make the following contributions:

- We propose a novel log fusion technique that features the capabilities of inferring new relations from existing logs.
- We develop a set of parsers that can normalize various logs to an expressive canonical form. We study the execution models of a set of popular applications from [54], [53], [45], [52] and their built-in application logs, and determine that their executions can be expressed by the canonical form that preserves the critical unit related information. In addition, we study the log format changes of these applications (Appendix A) and find that log formats rarely change, much less frequently compared to software releases. Note that for instrumentation based techniques, each software release entails re-instrumentation.
- We develop a comprehensive set of log fusion rules general for all applications. We devise a demand-driven inference algorithm to handle a large volume of log events in the Datalog engine.
- We develop a prototype on Linux and evaluate it on 8 machines for 7 days. The results show that ALchemist achieves 92.8% precision and 99.6% recall with only 1.1% run time overhead and 6.8% storage overhead, implying that ALchemist can achieve similar accuracy and lower overhead, when compared to instrumentation based approaches. In the study of 14 attacks collected from the literature, ALchemist outperforms NoDoze [31], a state-of-the-art technique that does not require instrumentation, and OmegaLog [33], another state-of-the-art technique that makes use of both application and audit logs.

Comparison with OmegaLog, NoDoze and Commercial Log Analysis Tools. OmegaLog [33] leverages application logs to recover execution paths, which can be used to partition execution to avoid dependence explosion. Particularly, a sequence of application log entries (e.g., those produced by `fprintf()`) can be used to recover an approximate program path, Repetition of such paths indicate an application is handling (independent) tasks. OmegaLog identifies such paths, projects each path to a corresponding audit log entry sequence, and then enables partitioning the audit log. It does not derive high level semantics from application logs except control flow path. Its dependence analysis is exclusively performed on the audit log. As such, although it works very well on server applications in which control flow paths of independent tasks do not interleave, it can hardly handle asynchronous/background behaviors that are very common in complex applications such as *firefox*. In contrast, ALchemist infers rich semantic information such as interleaving atomic sections from concurrent tasks and dependences invisible in either application log or the audit log alone, through log fusion. Please see our comparative results in Section V-D.

NoDoze [31] uses unsupervised learning to predict if a dependence edge is normal. It only includes the abnormal edges in the provenance graph. In our example, if *x.x.x.x* is rarely visited, it will be included. With NoDoze, most normal browsing behaviors (e.g., visiting `CNN.com`) are recognized as normal and precluded. While it can substantially reduce the graph size, depending on the quality of normal behavior profile, it may have both false positives (e.g., including benign websites that people rarely visit as part of the attack graph) and false negatives (e.g., missing malicious behaviors involving benign sites/IPs/applications). Similar to OmegaLog, it can hardly handle bogus dependencies caused by asynchronous/background behaviors.

Commercial log analysis tools such as Splunk [13] and Elasticsearch [8] use pre-built parsers to process unstructured application built-in logs to structured databases that can be queried. Multiple application logs can be correlated (e.g., through common file names). However, they do not construct a canonical representation. Neither do they derive new and implicit relations from existing ones. They are not designed for forensics and hence they cannnot directly generate attack provenance graphs or handle dependence explosion.

Threat Model. ALchemist aims to detect attacks which exploit application vulnerabilities or leverage social engineering techniques to get into victim systems for data exfiltration or manipulation. And we consider hardware or side channel related attacks to be out of scope of this paper. Similar to many existing works [19], [62], [61], [45], [44], [54], [31], [33], we assume the Linux kernel and the components associated with the audit logging system, which may be in the user space, are part of our trusted computing base (TCB). We also assume the application logs can be trusted. Note that existing works [53], [45], [33] also trust the (instrumented) applications or their built-in logs. As pointed out in [53], [45], [31], [61], although the attackers can subvert applications or even the kernel such that logs are compromised, the subversion procedure can be precisely captured (by the logs before they are compromised). Existing software and kernel hardening techniques (e.g., [30], [19]) can be used to secure log storage. Cryptographic hash values can be computed for log events (or event blocks) and stored as part of the application logs [51], [50], [16] such that tampering efforts can be detected. They are orthogonal to ALchemist and beyond the scope of our paper.

II. Motivation

One day, a user receives an email with a phishing link. She clicks the link and a compromised software repository website is opened in a new tab. During page loading, a malicious JS script is executed to download a compromised *fcopy* from *x.x.x.x*. Later, the user executes *fcopy* without realizing that it has been compromised. Upon execution, the malware copies sensitive data files to a shared folder */var/www/html*. In order to remove the attack trace, it also creates a php file *cleaner.php* which deletes attack-related files after sending them to the attacker (i.e., site *z.z.z.z*). The suspicious connection to *z.z.z.z* is detected, leading to investigation. The example is different

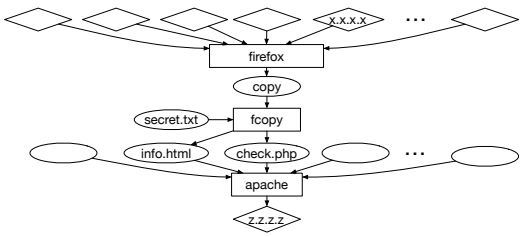

Fig. 1: Causal graph by syscall only methods (e.g., [41])

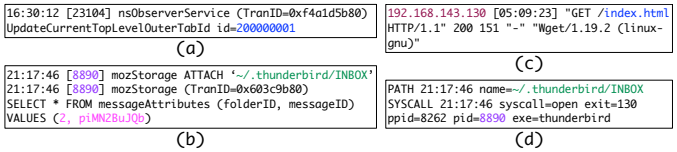

Fig. 2: (a) *firefox* tab switch log (b) *thunderbird* email open log (c) *apache* request log (d) *thunderbird* email open audit log

from attacks discussed in existing works [53], [31], [33] as it involves background JS execution as part of its attack chain, which is difficult for many existing works.

A. Syscall Only Approaches

Many existing approaches analyze only system logs generated by OS level logging tools (e.g., Linux Audit and Event Tracing for Windows) [41], [28], [39]. They consider a whole process as a subject and hence an output event is dependent on all the preceding input events. In a long running process such as *firefox*, such design leads to substantial bogus dependencies. This is the *dependence explosion problem* [45]. Fig. 1 shows the attack causal graph generated by these techniques. In this graph and also the rest of the paper, we use diamonds to represent sockets, oval nodes to represent files or application data structures, and boxes to represent processes or execution units. An execution unit is a part of process execution that handles an individual task (e.g., a tab in *firefox*). Existing works [31], [33], [45], [53], [46] have shown its importance in attack investigation. Edges correspond to causality oriented in the direction of data flow. Starting from the symptom, namely, the connection to *z.z.z.z* in Fig. 1, these approaches backtrace the depending subjects and objects. Specifically, as the connection is established by *apache*, a process node denoting *apache* is included in the graph. And all the related objects (e.g., *info.html*) are included too. Furthermore, process *fcopy* which updates these objects is included. It is determined that *fcopy* is downloaded via *firefox*. However, as *firefox* interacts with multiple IPs simultaneously (through foreground/background activities), all these IPs are included in the graph. Such dependence explosion causes substantial difficulty locating the root cause IP *x.x.x.x*.

B. Our approach

The inaccuracy of syscall-only approaches is because they are not aware of application semantics. The overarching idea of our technique is to couple the high level semantics in application log and the low level details in the audit log.

Built-in Application Log Providing Critical High Level Semantics. We observe that built-in application logs provide rich

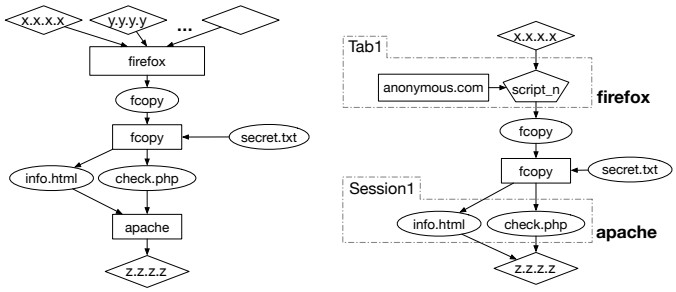

Fig. 3: Causal graphs by (a) NoDoze and (b) ALchemist

semantics regardless of the programming languages. In our example, the three applications involved, *firefox*, *thunderbird*, and *apache* all have built-in logs that provide critical information for execution partitioning and dependence identification, which are the key to the success of attack provenance tracking. For example, *firefox* by-default logs any tab creation and switch, allowing precise identification of execution unit boundaries. Fig. 2(a) shows a *firefox* log entry that records opening a new tab with a tab id 200000001. Note that such operation is oblivious at the syscall level. Similarly, *thunderbird* logs the opening of each individual email as shown in Fig. 2(b) with the `folderID` and `messageID` uniquely identifying an email. In contrast, since all emails are stored in the same INBOX file, accesses to different emails are indistinguishable at the syscall level. Fig. 2(c) shows an *apache* built-in log entry that records a new request, which is a natural execution unit for *apache*.

Besides task structure, application logs also contain critical dependence information that is not available at the system call level. For example in *firefox*, a tab's execution is broken down to smaller sub-tasks (e.g., requesting a page, rendering an image, and executing a JS code blob) that are dispatched to various concurrent worker threads, which may further break down these subtasks. Subtask executions from different tabs interleave and are hence extremely difficult for existing techniques to separate at the system call level. To help developers debug and maintain the code base, *firefox* uniquely identifies each atomic sub-task internally and logs their creation. From such information, ALchemist can extract precise depenendences among sub-tasks through log fusion (see Section IV-B).

Syscall Log Providing Low Level Details. On the other hand, audit logs are irreplaceable as they record low-level and background information that is invisible or less interesting for developers (but critical for system-wide dependence). For example, while loading the `CNN.com` main page in *firefox* generates 2583 application log entries, the same operation leads to 107140 audit log entries, which contain information not captured by the application log, such as configuration file accesses, cache file accesses, and network traffic not through the standard *firefox* APIs. In our example, the access to *secret.txt* by the malicious *fcopy* is only visible at the syscall level. Hence, *our method is to fuse application log and audit log so that on one hand, the rich application semantics can be propagated to the syscall level, and on the other hand the low-level background information recorded in the audit log can be properly attributed to high-level application execution units, precluding bogus dependencies.*

Specifically, as shown by Fig. 6, application logs and the audit log (on the left) are first normalized to a canonical form using a set of parsers, one for each raw log format. Building such parsers is an almost one-time effort as raw log format rarely changes. A number of relations (like relations in databases) can be directly derived from the normalized log entries. For example, a relation $initUnit(Tab\_X)$ means that the current *firefox* tab is switched to a new Tab_X. These basic relations are provided to a Datalog inference engine [38] (in the center of Fig. 6), which can derive new relations from the basic ones following a set of pre-defined rules. In particular, these rules derive correlations and correspondence from both the application log relations and the audit log relations. The key is that these two levels of relations often share common fields. For instance, Fig. 2(d) shows the *thunderbird* read of the INBOX file in the audit log. Our technique projects it to the high-level email access (corresponding to the application log in Fig. 2(b)). Since application logs provide a clear execution unit structure, by projecting such structure to the low level audit log, we are able to achieve execution partitioning at the audit level *without any instrumentation.* Fig. 3(b) shows the graph by ALchemist. Observe that it avoids dependence explosion. That is, only the tab visiting the compromised website (*anonymous.com*) is included, compared to the graph in Fig. 1 that includes the execution of all tabs. In addition, it precisely identifies that *fcopy* is generated by *script_n*, which downloads from *x.x.x.x*, as the execution of the different JS files (*script_0* to *script_n*) is correctly separated by ALchemist. Fig. 3(a) presents the graph by NoDoze. Although it is also smaller than that in Fig. 1, it cannot fully prune the false dependencies of *firefox* as they are not frequent dependencies. It does not contain tab information either. Furthermore, the graph cannot indicate that *fcopy* is downloaded by the execution of a JS file downloaded from *x.x.x.x*.

III. Study of Popular Linux Application Execution Models and Feasibility of Log Fusion

To study the feasibility and generality of our design, we conduct a manual study of 32 Linux applications, which are the union of 30 most popular applications listed in [1] and 15 complex applications widely used in the APT attack literature, such as *firefox*, *Thunderbird*, *Chromium*, *OpenOffice*, *LibreOffice*, and *Apache*. We aim to study their execution models and available logs (including both built-in logs and the audit log) to validate the following: (1) *if log fusion can disclose (implicit) information to identify execution units, including interleaving/background units, and recover dependences that are invisible in neither application logs or the audit log*; (2) *how often log format changes*. The study focuses on the applications' background (asynchronous) activities, which are the most prominent challenge in dependence analysis due to their non-deterministic interleavings. The applications and their execution models are listed in Table X in Appendix B. We find that these models can be divided into five different categories, with each application using one or multiple models.

Class I: Handling Tasks Sequentially in A Single Process. A number of applications are single process such as *vim* and *wget*. They do not have asynchronous behavior, but rather handle tasks one by one in a main loop. *Vim* uses the main loop to execute user commands one by one. As shown in

```
1  static void auto_next_pat(...) {
2    ...
3    s = _("%s Auto commands for \"%s\"");
4    sprintf((char *)sourcing_name, s, ...);
5    smsg(("Executing %s"), sourcing_name);
6    ...
7  }
```

```
8  15:36:49 Executing BufEnter Auto commands
9  for function LocalBrowse('/home/user/
10 Desktop/file')
```

(a)

```
11 static void server_accept_loop(...) {
12   ...
13   if ((pid = fork()) == 0) {
14     debug("Forked child %ld.", pid);
15     ...
16   }
17 }
```

```
18 07:25:47 sshd[1054] Forked child 1580
```

(b)

Fig. 4: (a) Source code and log for *vim 7.3* (b) Source code and log for *sshd 7.4*

Fig. 4(a), it uses function `auto_next_pat()` to retrieve the next command and then executes it. Inside the function, *vim* leverages its logging function `smsg()` (line 5) to record each executed command. These recorded commands can be leveraged to identify units. For example, we partition *vim*'s execution based on files, which are denoted by the file buffer data structures internally, one buffer for each loaded file. Every time the user opens/switches-to a window of some file, a command "`BufEnter`" is executed. Every time the user exits a window, a command "`BufLeave`" is executed. Lines 8-10 show a log entry for the command "`BufEnter`" that opens a file "`/home/user/Desktop/ file`". Since the execution is sequential, all the low level audit events (e.g., file updates) that happen between this command and the corresponding "`BufLeave`" command can be correctly and safely attributed to the unit of the file. In fact, we observe that the application log contains so wealthy information that other partitioning schemes (e.g., based on folders) can be supported.

Class II: Handling Tasks by Forking Additional Processes. Some applications, especially those server applications that need privilege separation, fork processes for new tasks. Fig. 4(b) shows a code snippet from *sshd* (lines 11-17) for starting a new connection, and the corresponding log event (line 18). The *sshd* daemon process invokes function `server_accept_loop()` in a while loop to handle a remote connection request. In the function, the daemon process forks a child process to handle the request (line 13). The child process may further spawn other processes for various functionalities (e.g., authentication). The dependences of individual subtasks can be precisely reflected by process creation, which is captured by the audit log and sometimes by the application log as well. For example, *sshd* logs task process creation (line 14). Line 18 shows the corresponding application log. Table X (in Appendix B) shows that there are quite a number of applications in this class. For these applications, a unit consists of a chain of inter-dependent processes.

Class III: Asynchronous Task Queue. A few applications such as *firefox*, *thunderbird*, and *foxit* make use of a more complex asynchronous execution model, in which the application has a main thread and a number of worker threads dedicated to some special functionalities. The main thread receives independent tasks (from the user), such as loading a page and accessing an email. It then dispatches the tasks to worker threads. The worker threads work in a pipeline, for example, a socket thread downloads a JS file and then hands it over to the JS helper thread to compile and execute. Each worker thread serves multiple tasks. The communication between main thread and worker threads, among worker threads themselves, is through task queues. Such an asynchronous execution model creates lots of difficulties for the low-level audit logging system [53] as syscalls from different pages,

```
19 static void * APR_THREAD_FUNC listener_thread(...) {
20   apr_pool_t *tpool = apr_thread_pool_get(thd);
21   while (1) {
22     ...
23     apr_log(plog, s);
24   }
25   ...
26 }
```
(c)

```
27 20:45:34 [9071] DOM Worker (TranID=0xae3ad5080) has
28 new timeout: delay=5000ms
29 ...
30 20:45:39 [9071] DOM Worker (TranID=0xae3ad5080)
31 executing timeout with original delay 5000 ms
```
(d)

```
32 WorkerPrivate::SetTimeout(JSContext* aCx, ...) {
33   ...
34   LOG(("(TranID=%p) has new timeout:delay=%f", ...));
35   rv = data->mTimer->InitWithCallback(
36                 data->mTimerRunnable, delay,
37                 nsITimer::TYPE_ONE_SHOT);
38 }
39
40 WorkerPrivate::RunExpiredTimeouts(JSContext* aCx) {
41   LOG(("(TranID=%p) executing timeout with original
42         delay %f ms.\n", ...));
43   ...
44 }
```
(d)

Fig. 5: (c) Request processing in `apache 2.4.20`; (d) Source code `firefox 60.0` DOM thread and its app log

tabs, JS blobs, and other background tasks are all interleaving, without any hints about their origins.

Firefox uses the NSPR logging module [11] which has been the uniform logging component for all Mozilla applications for 10 years. NSPR defines and records a large set of events that are important for Mozilla products. In the context of *firefox*, it intercepts and records events such as page loading, tab switches, and opening a page through some hyper link. More importantly, it is designed particularly for the asynchronous execution model. It treats each sub-task dispatched to some worker thread (e.g., saving a file) as a *transaction*, uniquely identified by a transaction id. Each sub-task dispatch is recorded as a transaction initialization event. The end of a sub-task is recorded by a destruction event of the transaction. Other events that happen within a transaction are often recorded with the enclosing transaction id.

By observing the chain of transaction initializations (e.g., a tab creates a transaction, which creates another transaction, and so on), we can identify an execution unit. By fusing application log events with the corresponding audit log events (e.g., the events that record the same file access), we could project the unit structure to the audit log. In addition, dependences with high level semantics (such as clicking a hyperlink) and hence invisible in the audit log can be inferred. An detailed example can be found in Section IV-B.

Class IV: Thread Pool. Many applications adopt a scheme slightly simpler than asynchronous task queue while providing a similar level support of asynchrony. Specifically, they dispatch tasks to available threads in a thread pool. Take *apache* as an example, in the code snippet in Fig. 5(c), the listener thread first acquires a pointer to the thread pool (line 20). Then it listens to any requests through a while loop in lines 21-24. Each time a request is received, it finds an idle thread from the pool or waits when such threads are not available. The request is then served by the thread. After handling a request, it logs the request at line 23. Note that although it is clear that in this execution model the handling of a request is a unit, we cannot simply consider all behaviors in a worker thread belong to the same unit as worker threads are being recycled. Instead, the application log entry contains the IP of the remote request, which can be leveraged to couple the application log entry and the corresponding audit log entries (of the worker thread), such as socket creation, read and write. Since a worker thread's execution is sequential, all the audit events in the time-span of coupled audit events belong to the same unit. An example is presented in Section IV-B.

Class V: Background Activities in Virtual Environment. Many applications support internal virtual environment in which (script) code blobs get executed. Such script languages are often very powerful, capable of conducting activities as

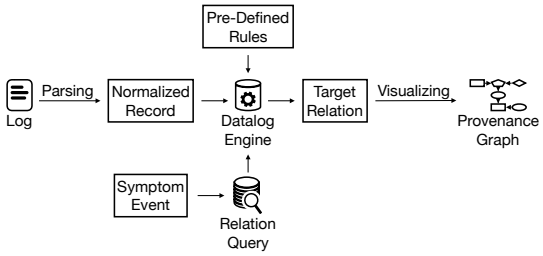

Fig. 6: ALchemist's workflow

complex as a full-fledged application. While the execution of a code blob can be correctly attributed to the proper execution unit as such execution is usually performed through some standard interface (e.g., the *firefox-spidermonkey* interface), which is recorded in the application log, the code blob could be designed in a way that itself induces the execution of other code blobs. Log fusion can nonetheless handle these cases, attributing the follow-up executions of other code blobs. In *firefox*, a JS code blob can invoke other code blobs asynchronously by registering them as event handlers. These events could be as simple as timeouts. Specifically, a JS code blob can call a built-in API `SetTimeOut()` (lines 32-38 in Fig. 5), to instruct *firefox* to execute a specified code blob when the timeout event happens. Function `RunExpiredTimeouts()` (lines 40-44) handles timeout events. Both functions log the current transaction id (line 34 and lines 41-42). Observe that the resulted log entries (lines 27-31) clearly indicate the event handling code blob and the original code blob share the same transaction id, allowing correct unit partitioning. Other event handling has a similar mechanism.

To summarize, our study shows that 30 of the 32 applications are long running, and 31 have built-in logs (or some history files). All the 31 applications' built-in logs record critical events that denote unit boundaries. Besides unit boundaries, our study also shows that the fusion of built-in logs and the audit log allows precisely tracking dependences in complex asynchronous execution models such as worker threads and thread pools, which can hardly be handled by existing techniques. For the 2 applications that are not long running, one of them does not have built-in log. Note that even *bash* has a history file that records all the interactive commands. Individual commands can hence be considered as different tasks such that dependence explosion through *bash* can be avoided. It does not mean we will miss the inter-dependences across commands as such dependences are visible at the audit log level (e.g., through files accessed). More details about the study can be found in Appendix B. We believe the reason that popular and/or long running applications have informative and well-designed built-in logs is that their developers tend to pay a lot of attention to ease of maintenance. Our study further shows that the design of logging component tends to be stable, much stabler than the application itself. For example, *firefox* has been using the same logging facility for 10 years while it has 64 different releases in that period. More can be found in Appendix A.

IV. System Design

In this section, we discuss the design details, including how to normalize various logs to basic relations and how to fuse them by performing inference and deriving new relations.

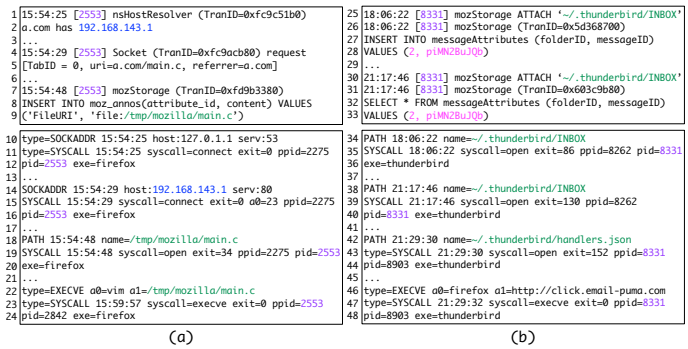

Fig. 7: Examples of raw application logs (top) and the corresponding audit logs (bottom). (a) *Firefox* saves `main.c` from `a.com` to `/tmp` and opens it with *vim*; (b) User opens a downloaded email from the local email box in Thunderbird and clicks a hyper link in the email.

TABLE I: Normalized audit records, A_1 denotes *firefox* and A_2 *thunderbird*, IP_0 denotes 127.0.0.1, IP_1 192.168.143.1

Index	Time	PID	PPID	PNAME	IP	Port	File	Action	Return
S1	15:54:25	2553	2275	A_1	IP_0	53	-	connect	0
S2	15:54:29	2553	2275	A_1	IP_1	80	-	connect	0
S3	15:54:48	2553	2275	A_1	-	-	main.c	open	34
S4	15:59:57	2842	2553	A_1	-	-	vim	execve	0
S5	18:06:22	8331	8262	A_2	-	-	INBOX	open	86
S6	21:17:46	8331	8262	A_2	-	-	INBOX	open	130
S7	21:29:30	8903	8331	A_2	-	-	handlers.json	open	152
S8	21:29:32	8903	8331	A_2	-	-	firefox	execve	0

Existing commercial log analysis tools such as Splunk [13] and Elasticsearch [8] have different structures for various applications' built-in logs. Different correlation rules may be needed for various pairs/combinations of logs. As such, they require intensive human efforts. A very important design goal of ALchemist is *generality*, that is, minimizing the efforts of constructing parsers and fusion rules for individual applications. Therefore, a key design choice is to parse all logs, including both application logs and the audit log, to an expressive canonical representation. General log fusion rules can hence be built on this central representation to reduce application-specific efforts. As such, in order to support an application in ALchemist, we just need to write a parser to parse its built-in log to the canonical form. According to our aforementioned study, such efforts are largely one-time.

A. Normalizing Logs to the Canonical Form

A Universal Execution Model. In Section III, we identify that the execution models of popular applications can be mainly classified in five categories. In order to enable the canonical log representation, we devise a universal execution model that can describe all these classes and captures all the information needed in attack forensics. The model features expressing the asynchronous/background behaviors. It is formally defined in Fig. 8. An execution consists of a set of *units*, whose definition is similar to that in the literature [53], [33]. A unit is composed of a sequence of *transactions*, each of which denotes an atomic sequential sub-execution of the unit. Each unit may require multiple transactions to complete its task. Transactions from different units may interleave, e.g., when they are executed by a thread. A transaction is composed of a sequence of events that access certain objects that are of interest for forensic analysts

TABLE II: Normalized *firefox* log (top) and *thunderbird* log (bottom), A_1 denotes *firefox* and A_2 denotes *thunderbird*, IP_1 denotes IP 192.168.143.1

Index	Time	PID	PNAME	IP	File	Action	UnitID	TranID	URI
F1	15:54:25	2553	A_1	IP_1	-	resolve	-	fc9c51b0	a.com
F2	15:54:29	2553	A_1	-	main.c	request	0	fc9acb80	a.com/main.c
F3	15:54:48	2553	A_1	-	main.c	createFile	-	fd9b3380	-

Index	Time	PID	PNAME	IP	File	Action	UnitID	TranID	URI
T1	18:06:22	8331	A_2	-	INBOX	createFile	piMN2BuJQb	5d368700	piMN2BuJQb
T2	21:17:46	8890	A_2	-	INBOX	openFile	piMN2BuJQb	603c9b80	piMN2BuJQb

$$
\begin{array}{lcl}
\textit{Execution } E & := & \{u_1, u_2, ...\} \\
\textit{Unit } u & := & t^* \mid e^+ \\
\textit{Transaction } t & := & e^+ \\
\textit{Event } e & := & \textit{read/write/create/...}\ (\textit{file} \mid \langle \textit{IP,port} \rangle \mid \textit{URI} \mid ...)
\end{array}
$$

Fig. 8: Universal Execution Model

such as files, IPs, and URIs. In some cases, a unit does not have transactions but rather a sequence of events.

The universal model can describe the aforementioned five classes. For example, in the task queue execution model (*class III*) of *firefox*, a unit is a tab. A transaction is a *firefox* transaction created and dispatched to some worker thread, which serves many tabs. In the thread pool execution model (*class IV*) of *apache*, a unit is a request. A transaction is the execution of a thread (from the pool) that handles part of the request. In some execution models such as *class I* that sequentially processes tasks, transactions are not necessary.

Canonical Log Representation. Based on the universal execution model, we devise the canonical log form. Each log event in the application and audit logs are parsed to an entry in the canonical form. It consists of 20 fields, which denote the timestamp, process/thread information, unit id, transaction id, operation (action), return value, and the resource that is being accessed, such as file and IP. Note that many of these fields(e.g., unit and transaction ids) may be vacant if the raw log does not have such information. However, such information can be inferred through log fusion.

In the following, we show a few examples of raw log entries and their canonical forms. To save space, we omit the vacant fields in the canonical forms. These examples are also intended to illustrate the benefits of log fusion.

Application Log and the Corresponding Audit Log For Sample Operations. Fig. 7 shows some (simplified) sample application logs and the corresponding audit logs. Specifically, Fig. 7(a) shows the log for *firefox* downloading a C file and then invoking *vim* to edit it; Fig. 7(b) shows the log for opening an email in *thunderbird* and then accessing an embedded hyper-link. The application logs are on the top and the corresponding audit logs are on the bottom.

In Fig. 7(a), the application log shows that *firefox* first resolves website `a.com` to IP address `192.168.143.1` (lines 1-2), and then starts a transaction `0xfc9acb80` to request resource `main.c` from `a.com` (lines 4-5). Next, *firefox* saves the file to `/tmp/mozilla/main.c` (lines 7-9). In contrast, at the low level, we see the socket connections to a local port `127.0.1.1:53` for name resolution (lines 10-12) and then to `192.168.143.1:80` for file download (lines 14-16). As audit log does not have semantic information,

it is difficult to know that the network connection at lines 14-16 is for sending the HTTP request and receiving `main.c`. On the other hand, the audit log discloses that *firefox* opens *vim* (lines 22-24), which is invisible in the application log.

In Fig. 7(b), the application log shows that an email is received and put into INBOX through a database insertion operation (lines 25-28) with a folder id and a message id. Then the email is read through a database selection operation (lines 30-33). Observe that the highlighted id values (in pink) denote an email, which is a natural execution unit for the email client. In the audit log, the email write and read are recorded as accesses to file `.thunderbird/INBOX`, without any information about the specific email. On the other hand, the audit log captures the behavior that the user clicks a link in the email and then opens a web page (lines 42-48), which are invisible from the application log.

In addition to being complementary, audit log and application logs share a lot of common information, which can be leveraged in log fusion. For instance, in the *firefox* example, the two levels of logs share the same IP address and the same file name; in the *thunderbird* example, the two levels of logs share the same INBOX directory and similar timestamps.

Table I shows the reduced canonical representation of audit log entries. Most fields are self-explaining. `Index` field is a global ID; `PNAME` is the process name; `Action` represents the type of the syscall and `Return` the return value. Similarly, Table II shows the normalized application logs. Observe that the normalized audit logs and application log entries can be correlated through their common fields. Observe that some canonical *firefox* log entries have the `UnitID`, `TranID` and `URI` fields filled as such information can be directly extracted from the tab id, transaction id, and resource URL, respectively, in *firefox*'s built-in log. The `UnitID`'s are missing in some entries. They will be filled by log fusion. As shown in the lower part of Table II, a normalized *thunderbird* log entry contains transaction id `TranID` similar to the *firefox* transaction id. An email is essentially a block inside the INBOX file and uniquely identified by a so-called `StorageInfo`, for instance the string "`piMN2BuJQb`" in the table. Since we consider continuous operations on an email as an unit, both the `URI` and `UnitID` fields are filled with the `StorageInfo` string.

In ALchemist, we have developed 15 parsers. Most logs can be expressed using regular expressions, without requiring the more complex context-free or even context-sensitive languages. As mentioned earlier, most popular applications have their own logs. For those that do not (e.g., *wget* in our benchmark set), ALchemist resorts only to the audit log to derive dependence. Specifically, for an application without its own log, ALchemist conservatively assumes any output event in the process of the application is dependent on all the preceding input events in the same process, similar to previous works [41], [42]. Since these applications are rarely long running, the conservativeness unlikely leads to undesirable consequences in practice.

B. Log Fusion

After normalization, ALchemist performs log fusion on the canonical logs. It first infers critical information from built-in log entries of individual applications, e.g., identifying tab switches in *firefox* log that serve as execution unit boundaries. It then correlates logs of different kinds through their shared fields to allow information to be propagated across applications, enabling discovery of new dependencies and avoiding the bogus ones. While the correlation analysis can be directly performed among different applications, doing so incurs quadratic complexity. We hence design a star-shape fusion scheme, in which each application log is fused with the common audit log. Information can be propagated from one application to another through the central audit log. The inference and fusion procedures are denoted as a set of inference rules in Datalog [7], which is a Prolog-like representation for relation computation. Note that these rules are general (i.e., not application-specific). Intuitively, each rule derives a new relation from existing ones. The inference terminates when a fixed point is reached.

The rules and the related definitions are presented in Fig. 9. At the beginning, we first define a number of types. Specifically, an application log event after normalization HR is a relation of 20 fields, which are a direct mapping from the canonical form. An audit event LR is similarly defined. We distinguish application log event from audit event although they are normalized to the same canonical form, because the fusion rules entail different operations on the two kinds. We also call them *high level event* and *low level event*, respectively. We define $ActionH$ and $ActionL$ as the type of event for the two kinds, respectively. For example, `switch` means switching to a tab, `init` and `end` denote transaction initialization and termination, respectively.

In the middle of Fig. 9, we define a number of basic relations called *atoms*. These relations are directly acquired from the normalized log entries without inference. We use form $p(x_1, x_2, \cdots, x_n)$ to represent a relation, with p the predicate (or name of the relation), and x_1, $\cdots$, x_n the variables. For instance, *isMember(whale, mammal)* means that the pair (*whale*, *mammal*) is a tuple in the relation with the name of *isMember*, or, the predicate *isMember* holds on the pair.

Atom (A1) $inSeqL(LR_1, LR_2)$ denotes that two low-level events LR_1, LR_2 are next to each other (in the audit log). Note that the explanation of each atom is to its right. These atoms also denote a list of relation short-hands for log entries. For example, (A7) $initUnit(HR,\ UnitID)$ denotes an event that starts a unit, for instance, a *firefox* event switching to a new tab denoted by $UnitID$. Atoms (A9)-(A13) denote the I/O related application events. For instance, (A9) denotes reading a *URI*, which could be a file, a remote URL, a file block, and so on. Note that URI stands for *uniform resource id* that can represent a wide range of resources. We also have a similar set of I/O atoms for low level events. In fact, we have a total of 258 atoms and only those necessary for the illustration of our technique are presented. These atoms are not application specific.

After the atoms, we define a set of inference rules that derive additional relations from atoms and fuse application and audit logs. These rules are in the following format.

$$H :- B_1 \ \& \ B_2 \ \& \ \cdots \ \& \ B_n$$

Specifically, H is the target relation, and B_t a predicate or a relation. It means that the presence of relations B_1, B_2,

Types:

$AppEvent$	HR	:=	$\langle Time, IDX, PID, PPID, PNAME, IP, Port, File, UnitID, TranID, ActionH, Return, URI$: uniform resource identifier, ...$\rangle$
$AuditEvent$	LR	:=	$\langle Time, IDX, PID, PPID, PNAME, IP, Port, File, UnitID, TranID, ActionL, Return, ...\rangle$
$ActionH$		:=	`switch` | `request` | `init` | `end` | `readURI` | `writeURI` | $\cdots$
$ActionL$		:=	`open` | `close` | `socket` | `connect` | `read` | `write` | $\cdots$
$Action$	A	:=	$ActionH \mid ActionL$

Atoms:

(A1) $inSeqL(LR_1, LR_2)$	:	$LR_1.IDX + 1 = LR_2.IDX$
(A2) $atomicL(LR_1, LR_2)$	:	LR_1 and LR_2 belong to the same atomic operation (i.e., socket create and connect)
(A3) $sameTime(Time_1, Time_2)$	:	the two timestamps have negligible difference
(A4) $inputAction(A)$	:	A is an input-related action
(A5) $outputAction(A)$	:	A is an output related action
(A6) $sameType(A_1, A_2)$	:	A_1 and A_2 belong to the same I/O type
(A7) $initUnit(HR, UnitID)$	:	HR starts a unit $UnitID$, e.g., firefox switches to a tab denoted by $UnitID$
(A8) $initTran(HR, TranID)$	:	HR starts a transaction with $TranID$
(A9) $readURI(HR, URI)$	:	HR requests URI
(A10) $writeURI(HR, URI)$	:	HR writes to URI, e.g., an email denoted by URI
(A11) $resolve(HR, URI, IP)$	:	HR resolves URI to IP
(A12) $readNtwk(HR, IP, Port)$	:	HR reads from $Port$ of IP
(A13) $requestFrom(HR, IP, URI)$	:	HR denotes a remote request from IP for URI

Inference Rules:

/* high level record is correlated to low level record if they operate on the same ip and port */
(R1) $correlated(HR, LR)$:- $HR.PID = LR.PID \ \& \ HR.IP = LR.IP \ \& \ HR.Port = LR.Port \ \& \ sameType(HR.ActionH, LR.ActionL)$

/* high level record is correlated to low level record if they operate on the same file */
(R2) $correlated(HR, LR)$:- $HR.PID = LR.PID \ \& \ HR.File = LR.File \ \& \ sameType(HR.ActionH, LR.ActionL)$

/* high level record is mapped to the nearest correlated low level record */
(R3) $project(HR, LR)$:- $correlated(HR, LR) \ \& \ sameTime(HR.Time, LR.Time)$

/* if two low level records belong to the same atomic action (e.g. socket create and connect), they are all mapped to the same high level record */
(R4) $project(HR, LR)$:- $project(HR, LR_1) \ \& \ atomicL(LR, LR_1)$

/* two high level events belong to the same transaction if they have the same transaction id */
(R5) $sameTran(HR_1, HR_2)$:- $HR_1.TranID = HR_2.TranID$

/* two high level events with the same unit id belong to the same unit */
(R6) $sameUnitH(HR_1, HR_2)$:- $HR_1.UnitID = HR_2.UnitID$

/* two high level events with the same transaction id belong to the same unit */
(R7) $sameUnitH(HR_1, HR_2)$:- $sameTran(HR_1, HR_2)$

/* two high level events with different transaction id belong to the same unit if the first transaction initializes the second one */
(R8) $sameUnitH(HR_1, HR_2)$:- $initTran(HR_1, HR_2.TranID)$

/* two low level events belong to the same unit if the corresponding high level records belong to same high level unit */
(R10) $sameUnitL(LR_1, LR_2)$:- $sameUnitH(HR_1, HR_2) \ \& \ project(HR_1, LR_1) \ \& \ project(HR_2, LR_2)$

/* a low level record is in the same unit as its preceding low level record if itself is not projected to a high level record */
(R11) $sameUnitL(LR_1, LR_2)$:- $project(HR_1, LR_1) \ \& \ \neg project(HR_2, LR_2) \ \& \ sameUnitL(LR_1, LR_3) \ \& \ inSeqL(LR_3, LR_2)$

/* HR_2 reads some data (denoted by URI) updated by HR_1 */
(R12) $depH(HR_1, HR_2)$:- $writeURI(HR_1, URI) \ \& \ readURI(HR_2, URI) \ \& \ HR_1.IDX < HR_2.IDX$

/* two low level records have dependence as long as the corresponding high level records have dependence */
(R13) $depL(LR_1, LR_2)$:- $depH(HR_1, HR_2) \ \& \ project(HR_1, LR_1) \ \& \ project(HR_2, LR_2)$

/* in the same unit, a record of the output type depends on preceding records of the input type, regardless of the resources they access */
(R14) $depL(LR_1, LR_2)$:- $sameUnitL(LR_1, LR_2) \ \& \ LR_1.IDX < LR_2.IDX \ \& \ inputAction(LR_1.ActionL) \ \& \ outputAction(LR_2.ActionL)$

/* for read/write of regular file, a file read depends on preceding writes to the same file regardless of their units */
(R15) $depL(LR_1, LR_2)$:- $LR_1.File = LR_2.File \ \& \ LR_1.IDX < LR_2.IDX \ \& \ outputAction(LR_1.ActionL) \ \& \ inputAction(LR_2.ActionL)$

Fig. 9: Log Fusion Rules

$\cdots$, B_n leads to the introduction of H. The ultimate goal of these inference rules is to derive four types of critical relations $sameUnitH(HR_1, HR_2)$, $sameUnitL(LR_1, LR_2)$, $depH(HR_1, HR_2)$, and $depL(LR_1, LR_2)$ that assert two high-level application log entries belong to the same execution unit, two low-level audit log entries belong to the same unit, two application log entries have (direct) dependence, and two audit log entries have (direct) dependence, respectively. They denote the two kinds of information derived by fusion: *execution units* and *dependences between events*, from which the attack provenance graph can be precisely constructed. An execution unit denotes an autonomous task (e.g., a tab in *firefox*). As shown by existing works [53], [33], unit partitioning is the key to high precision in dependence analysis. In particular, dependences may be induced between an input event and an output event through computation *in memory*. For example, a file write may be dependent on a socket read if part of the socket buffer is appended to the file. However, such dependences are invisible for syscall level analysis (as the I/O events operate on different system resources). Although instruction level tracing can detect them, it is too expensive in practice. With unit partitioning, an output event is considered to have dependences on all the preceding input events *within the same unit*, even when they operate on different system resources. In

addition, dependences across units can be directly derived from operations on common resources. Many of them are invisible without fusion. For example, reads/writes of different emails are recorded as reads/writes to the same INBOX file in the audit log and hence not distinguishable. However, through log fusion, dependences induced by reads and writes of the same email across units can be precisely derived (through the email URI in the corresponding application log entries).

The first two rules (R1) and (R2) of $correlated(HR, LR)$ correlate a high level event and a low level event through their shared fields, indicating that they may denote the same resource access. Note that the two rules have different pre-conditions (i.e., different right-hand-sides), denoting the different scenarios (i.e., through network connection or file) that correlate an HR and an LR. However, two events being correlated may not mean they correspond to each other. For example, assume a file is read twice at two distant timestamps. Each read gives rise to an HR and an LR. The HR of the first access is correlated to the LR of both accesses while it only corresponds to the first LR. Hence, we introduce a relation $project(HR, LR)$ to derive precise correspondence. The first $project(HR, LR)$ rule (R3) projects an HR to a low level LR if they are correlated and their timestamps have negligible differences; and the second rule (R4) projects HR to LR if there has been another low level event LR_1 such that LR and LR_1 belong to the same atomic operation, and there has been projection from HR to LR_1. Note that it is common that an atomic operation at the application level (e.g., establishing a network connection) corresponds to multiple low-level audit events (e.g., socket creation and connection). Such atomic relations are captured by the aforementioned atom (A2) $atomicL$.

The next rule (R5) $sameTran(HR_1, HR_2)$ identifies two HR's with the same transaction id belong to the same transaction. The next three rules (R6)-(R8) $sameUnitH(HR_1, HR_2)$ infer the high level events that belong to the same unit. Recall that a unit may have many transactions. For example, a tab's execution may consist of requesting a page, downloading a file, and executing a piece of JS code. In the first rule (R6), there are some log entries that contain explicit unit id. For example, a switch to tab t event and all the URL request events from tab t share the same tab id t and hence belong to the same tab. The second rule (R7) dictates that all the events in the same transaction must belong to the same tab. The last rule (R8) includes all the transitive transactions into the same tab. Note that it is very common a sub-task in *firefox* spawns its own sub-tasks, and so on, leading to a chain of transactions.

The next two $sameUnitL(LR_1, LR_2)$ rules (R10) and (R11) group audit log entries to the same unit. The first rule (R10) dictates that the audit events that have the same HR projection belong to the same unit. The second rule (R11) says that if LR_1 is projected to HR_1, LR_2 does not have any projection, but it is right after another audit event LR_3 that has been determined to be in the same unit as LR_1, then LR_2 is considered to be in the same unit as LR_1. This rule essentially renders *forward attribution*, which means that *if there are low level (audit) events that are not projected to any high level (application) event in between two low level events that have projection to high level, these un-projected low event events are considered to be in the same unit as the preceding projected low level event*. As we will demonstrate with examples, this rule is particularly important for proper attribution of background activities.

Fig. 10: *Firefox* Asynchronous Download

Foreground	Background	Thread	Line#	Firefox Logs	Audit Logs
Tab A	Tab A	Socket	1		socket(fd0)
			2	readURI(A, #0, cnn.com/index.html)	connect(151.101.129.67, fd0)
			3		...
Tab B	Tab B	Main	4	initUnit(B)	
	Tab A	Resolver	5	resolve(-, #1, a.com, 192.168.143.1)	connect(127.0.0.1, fd1)
			6		open(TRRBlacklist.txt, fd2)
			7		write(TRRBlacklist.txt, fd2)
			8	initTran(-, #1, #2)	
		Socket	9	readURI(A, #2, a.com/tp.js)	connect(192.168.143.1, fd3)
	Tab B		10	readURI(B, #3, cnn.com/news.img)	connect(151.101.129.67, fd4)
			11	...	...
			12	readURI(-, #3, 151,101.129.67)	recvfrom(151.101.129.67, fd4)
	Tab A		13	readURI(-, #2, 192,168,143,1)	recvfrom(192.168.143.1, fd3)
			14	initTran(-, #2, #4)	
		Cache	15	readURI(-, #4, cache/tp.js)	open(cache/tp.js, fd5)
			16	writeURI(-, #4, cache/tp.js)	write(cache/tp.js, fd5)
			17	initTran(-, #4, #5)	
		JS Helper	18	readURI(-, #5, cache/tp.js)	open(cache/tp.js, fd6)
			19	...	...
			20	initTran(#5, #6)	
		FS Broker	21	readURI(-, #6, secret.txt)	open(secret.txt, fd7)
			22		read(secret.txt, fd7)

Fig. 11: Log for *Firefox* Asynchronous Download (*readURI*(A,#0,...) means a HR in tab A with transcation id 0 reads some URI)

The last four rules (R12)-(R15) derive dependencies between events. The first $depH$ rule (R12) specifies that read and write on the same URI induces dependence. This rule allows us to infer high level semantic dependences invisible in the audit log. Besides the email dependence example mentioned earlier, another example is that *firefox* stores cookies (of all websites) to the same file `cookies.sqlite`. Without the URIs identifying individual cookies and only considering syscall information, reading the cookie for a website would be dependent on all the preceding writes to cookie from any website. The first $depL$ rule (R13) inherits dependence from high level log entries. The second $depL$ rule (R14) specifies that any output event is dependent on all the preceding input events in the same unit. Note that a preceding input event (e.g., a socket read) may be on an object different from the output event (e.g., a file write). This approximation is critical for capturing invisible data-flow (e.g., through memory). The third $depL$ rule (R15) derives cross-unit and even cross-process dependence by the common resource that is operated on. In ALchemist, we have 135 fusion rules in total and we only present the ones that are necessary to illustrate the idea.

Example 1: Fusing Firefox Log and Audit Log. Fig. 10 shows a sample execution of *firefox* accessing `CNN.com`. In this execution, the user first loads the `CNN.com` main page (step ①). As part of the page loading, a JS file `tp.js` is requested. However, before the file is downloaded and executed, the user clicks a page link on the main page, which loads a news page about measles (step ②). The downloading and the execution of the JS file are hence happening in the

background (step ③), interleaved with the loading process of the news page (e.g., loading `news.img` in step ④). The resulting syscall interleaving makes causality inference very difficult for existing techniques. We will use this example to demonstrate how log fusion allows dis-entangling the complex interleaving.

Fig. 11 shows the runtime information of the example execution in Fig. 10 that accesses `CNN.com`. The first column shows that in the foreground, there are two tabs, with tab B displayed after tab A. The second column shows that in the background, the execution of the two tabs interleave (with B's execution shaded). The third column shows the list of threads that execute in the temporal order. There are multiple worker threads with the `Socket` thread managing network communication, `Resolver` resolving host names, `Cache` maintaining the file cache, `JS Helper` compiling and executing JS code blob, and `FS Broker` performing file system operations. Observe that a thread may serve multiple tabs (e.g., lines 9-14 in column four). Columns five and six show the application log atoms and audit log atoms.

From the application log, we can see that the `Socket` thread first requests the main page of `CNN.com` (line 2) in transaction #0 in tab A, which is normalized to a *readURI* atom. Then the `Main` thread switches to tab B (normalized to *initUnit* at line 4). In the background, the `Resolver` thread resolves the host of the JS file, `a.com`, in transaction #1 (line 5). Observe that the unit id is unknown in the atom as the raw log does not have such information. We will see log fusion can recover such information later. It then initializes a child transaction #2 (line 8) that will download the JS file. In lines 9-14, the `Socket` thread first requests the JS file in transaction #2 and then requests and reads `news.img` for tab B (lines 10-12). Observe that the *readURI* at line 13 is directly from the IP (without unit information). At the end, it switches back to serve tab A by receiving the JS file (line 13) and starting a new transaction #4 (line 14) to cache the JS file (lines 15-16). Transaction #4 initiates #5 to compile and execute the JS file (lines 18-20), which opens and reads a file "`secret.txt`" through the `FS Broker` thread (lines 21-22). From the audit log (in the last column), we observe the corresponding syscalls for many of the application level operations. For example, the first request of the main page at the application level corresponds to a socket creation syscall (line 1) and a connect syscall (line 2). There are also syscalls that are invisible at the application level, such as the open and write of file "`TRRBlacklist.txt`" (lines 6-7) that contains a list of websites that are blocked.

Observe that many high level atoms miss the unit information and none of the low level atoms have any unit information. In addition, dependences are invisible. In the following, we show how the two logs are fused to derive such missing information. We use F_t and A_t to denote the *firefox* event and the audit event at line t in Fig. 11.

According to rule (R5) in Fig. 9, we can derive $sameTran(F_5, F_8)$ and $sameTran(F_{13}, F_{14})$. By rule (R7), these pairs are in the same unit. By rule (R8), we have $sameUnitH(F_8, F_9)$ and $sameUnitH(F_{14}, F_{15})$. By rule (R6), we have $sameUnitH(F_2, F_9)$ due to the same tab id A. At the end of inference, we can determine that all the plain *firefox* log entries in Fig. 11 belong to the same unit. The shaded entries belong to another unit.

TABLE III: *Apache* execution

Thread	Line	Apache Logs	Audit Logs
Worker	1		`socket(fd0)`
	2		`accept4(172.16.163.1, fd0)`
	3		`read(172.16.163.1, fd0)`
	4		`...`
	5	`REQ(...)`*	`open(/var/www/html/payload.php, fd1)`
	6		`...`
	7		`open(/var/www/html/secret.txt, fd2)`
	8		`...`
	9		`writev(172.16.163.1, fd0)`
	10		`shutdown(172.16.163.1, fd0)`
	11		`...`
	12		`accept4(168.128.16.1, fd3)`

* `requestFrom(-,-,172.16.163.1, payload.php)`

Following rules (R1) and (R3), we have $project(F_2, A_2)$. Note that although F_2 has an URI "`CNN.com/index.html`", it is resolved to an IP 151.101.129.67, which allows (R1) to apply. Similarly, we have $project\ (F_5, A_5)$, $project(F_9, A_9)$, $project(F_{10}, A_{10})$, $project(F_{12}, A_{12})$, and so on. We also have $atomicL(A_1, A_2)$ due to the atomicity of the two operations, by (A2). As such, we have $project(F_2, A_1)$ by rule (R4). By (R10), we have $sameUnitL(A_2, A_5)$, which further entails $sameUnitL\ (A_2, A_6)$ by (R11), i.e., the forward attribution rule. Similarly, we have $sameUnitL(A_2, A_7)$. At the end, we correctly partition the audit events to two units, namely, the plain events and the shaded events. Furthermore, through rules (R14), we get $depL(A_{16}, A_{13})$, which correctly captures the dependence that the JS file was received from network and written to a file. And due to execution partitioning, the false dependence from A_{16} to A_{12} is avoided. In contrast, NoDoze [31] cannot distinguish the true dependence between A_{16} and A_{13} from the false dependence between A_{16} and A_{12}. OmegaLog [33] cannot identify the false dependence either as it cannot distinguish the subtasks from different units in a worker thread. In fact, as shown by our results in Section V-E, they do not work well when asynchronous behaviors are intensive.

Example 2: Request Serving in Apache. To show the generality of log fusion, we use another example in which *apache* serves a request. Recall that it has a class IV thread pool execution model, in which a thread from the pool is being reused to serve multiple requests. In Table III, an attacker from IP `172.16.163.1` requests a file `payload.php` from the *apache* server. The request is recorded in the *apache* log at line 5. The audit log column shows the low level events invisible at the application level. They all belong to a same thread (from the pool). Note that audit log entries contain thread id, allowing us to separate them by threads. Lines 1-10 belong to the request and lines 11-12 belong to another later request served by the same thread. Lines 1 and 2 are atomic. Lines 2, 3, 9, and 10 (in the audit log) share the common IP field with the application log entry (at line 5). As such, rules (R1), (R3), and (R4) allow us to project lines 2, 3, 9, and 10 in audit to line 5 in the application log. They hence belong to the same unit. According to rule (R11), lines 4-8 are attributed to the same unit too although they are not projected to any application log. This precisely reflects that tasks within a thread are processed sequentially.

We use the above two examples to show how the fusion rules can be applied in real scenarios to remove false dependence. However, it is possible that the attackers can craft special application workloads to attack our rules. We will explore this in our future work.

Apply to New Programs. ALchemist is designed in a way that aims to minimize the efforts of extending the technique to a new application. The canonical form, atoms, and fusion rules are all general and shared by all applications which can be expressed by our execution models. For such a new application, the analyst only needs to provide the parser to parse its built-in log to the canonical form. Note that it is completely fine if certain fields of the canonical form are vacant when the raw log does not have such information explicitly. ALchemist will infer it by log fusion if the raw log has sufficient (implicit) information to reflect the underlying execution model, which is the case for most the applications we studied. As discussed earlier, writing parsers is largely one-time efforts. However, it is possible that our execution model study do not cover some execution models, leading to incompleteness of the fusion rules. We plan to address this issue in the future work.

C. Demand-driven Datalog Inference, Graph Construction

ALchemist relies on the underlying Datalog inference engine to perform log fusion. However, according to our experiment in Section V, on average three million audit events and thirty thousand application log events can be generated everyday with a regular workload. Complex attacks may span over days, weeks and even months. It is infeasible for a Datalog engine to operate on the logs of such a long period. We leverage the observation that although attack span may be long, the attack behaviors may only be a very small portion of overall logged behaviors. Given that ALchemist is capable of avoiding bogus dependencies, we propose a *demand-driven* Datalog inference algorithm. Particularly, for a backward forensic task that tries to identify the root cause of an attack, we start with the raw logs (of a long period of time) and the symptom event. We separate the log entries by processes. We then perform log fusion on the process of the symptom to construct its causal graph, e.g., through rules (R12)-(R15) in Fig. 9. With the dependence relations, the provenance graph is constructed as follows.

Provenance Graph Construction. A *unit node* is created for each unit. It contains all the application log events and audit log events in the unit based on the $sameUnitH$ and $sameUnitL$ relations (in Fig. 9). An *event node* is created for each event such that each unit node contains a set of event nodes. *Dependence* edges are introduced between event nodes according to the $depH$ and $depL$ relations. *Projection* edges are introduced between an application event node and an audit event node according to the $projection$ relation. Examples can be found in Section VI.

After the dependence graph is constructed, it is traversed backward from the symptom event. Through the traversal, ALchemist identifies the other processes that are causally related to the symptom through direct or transitive dependencies. Then, only the logs of those processes are fused and further traversed. Section V shows that such a demand-driven strategy substantially reduces the workload for the Datalog engine.

TABLE IV: Attack overview

No.	Platform	Duration	Attack Surface	Scenario Name	Attack Reference
1	Ubuntu 14.04	0d8h	TightVNC-1.3.9	Ransomware	Case3.5(Engagement 4)
2	Ubuntu 14.04	0d3h	Nginx-1.2.9	Backdoor	Case3.8(Engagement 3)
3	Ubuntu 14.04	0d20h	Firefox54.0	Phishing Email Link	Case4.5(Engagement 3)
4	Ubuntu 14.04	0d10h	Firefox54.0	Exfiltration	Case4.9(Engagement 3)
5	Ubuntu 14.04	1d23h	Firefox54.0	Phishing Email Exec	Case4.8(Engagement 3)
6	Mint 17.1	0d7h	OpenSSH-6.6	Metasploit	Case3.6(Engagement 4)
7	Mint 17.1	0d5h	OpenSSH-6.6	Azazel Injection	Case3.2(Engagement 4)
8	Mint 17.1	1d0h	Nginx-1.2.9	SSHD Injection	Case3.14(Engagement 3)
9	Mint 17.1	0d3h	Nginx-1.2.9	Web-Shell	Case3.1(Engagemnet 3)
10	Mint 17.1	0d10h	Firefox54.0	RAT Malware	Case4.4(Engagement 3)
11	Ubuntu 14.04	0d19h	Apache-2.4.7	ShellShock	NoDoze[31]
12	Ubuntu 14.04	1d20h	OpenSSH-6.6	passwd-gzip-scp	High Fidelity[71]
13	Ubuntu 14.044	1d18h	Apache-2.4.7	Cheating Student	ProTracer[54]
14	Ubuntu 14.04	0d23h	OpenSSH-6.6	Data Theft	PrioTracker[49]

V. Evaluation

ALchemist supports both Linux 64 bits and 32 bits systems. Its code base includes approximately 600 lines of parser specification, 11500 lines of Python code, and 1900 lines of Datalog rules. We focus on the following research questions.

RQ1 What is the runtime and space overhead of ALchemist(Section V-B)?

RQ2 What is the performance of Datalog module when analyzing real world attacks (Section V-C)?

RQ3 How effective is our execution partitioning scheme based on log fusion (Section V-D)?

RQ4 How effective is ALchemist when analyzing real world attacks? How does it compare to the state-of-the-art techniques that do not require instrumentation: NoDoze [31] and OmegaLog [33] (Section V-E)?

A. Experiment Setup

To evaluate the efficiency of ALchemist (RQ1), we use 12 popular applications collected from the literature [53], [45]. To answer RQ3, we construct a few most commonly seen use cases for each application, which involve intensive background behaviors, and demonstrate that ALchemist can correctly partition these executions and attribute the background activities. Also, to show the effectiveness of ALchemist (RQ4) and study the performance of Datalog inference (RQ2), we emulate 10 advanced attacks collected from various public resources including the DARPA TC engagements [6] and the 4 real world attacks in NoDoze [31]. We are not able to acquire the implementation of NoDoze or OmegaLog. We hence reimplement them based on the papers and validate the correctness of reimplementation by comparing the results of our implementation with their published results.

Our evaluation environments include the Ubuntu 14.04 64-bit operating system (as a few attacks require exploiting vulnerabilities on 64-bit applications) and the Mint 17.1 32-bit operating system. These systems have the audit logging module running, with the configuration of collecting 48 security related syscalls. The built-in application logging components are all activated. Several attacker machines with different IPs launch remote attacks and generate benign workload. NoDoze

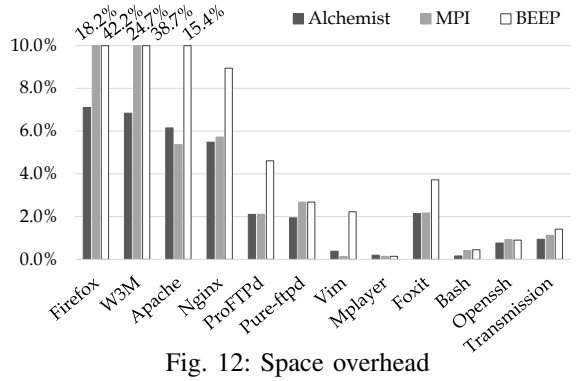

Fig. 12: Space overhead

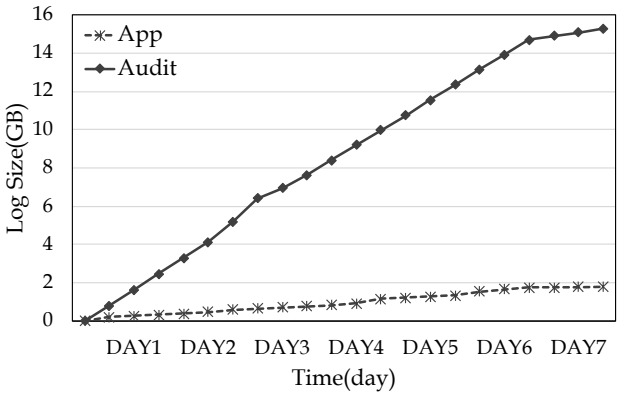

Fig. 13: Space consumption over a week

requires collecting event frequency in normal workload in order to determine outlier events during deployment. To collect such profile, we collect audit logs of 4 weeks from 10 work-stations in our institute (running typical end-user workloads) and calculate the frequency of each dependence edge. These work-stations are used exclusively by the authors of this paper and they all agree to use the collected data for their own research. Besides, we anonymized the identity related information including account names, private file names, and private domain names.

To answer the research questions, we run 8 systems for seven days. Most of the time, the systems are dealing with normal workloads, e.g., as the primary machine for daily usage. The fourteen attacks are conducted during the seven-day period on various machines (some machines having more than one attacks conducted). We assume that we know the symptom events and we conduct backward analysis to understand the root cause. The details of fourteen attacks (nine on the 64-bits platform and the other five on 32-bits) are shown in Table IV. They are reproduced based on reports at [6] and description in [31]. Observe in column 3, each attack procedure is distributed in a long duration of time (within the 7-day period), in order to simulate real attacks and test how well ALchemist can identify attack provenance from benign workload. The Datalog inference module and visualization module are deployed on a separate server with Intel i7-9700 CPU 4.7GHz and 64 GB memory running Ubuntu 14.04 OS.

B. System Overhead (RQ1)

Space Overhead. In the overhead experiments, we acquire the implementations of MPI [53] and BEEP [45] from their

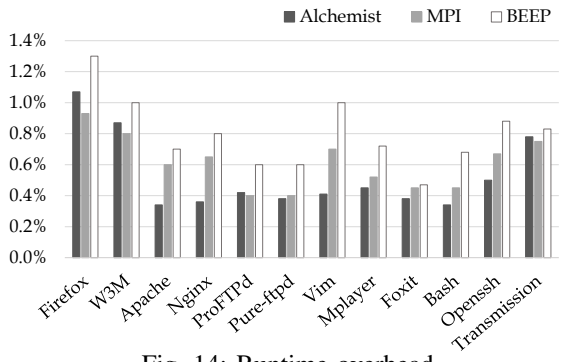

Fig. 14: Runtime overhead

authors and compare ALchemist with them. To measure space overhead, we use the logs from the one-week experiments on the 8 systems. We have turned on ALchemist, MPI and BEEP. The results are shown in Fig. 12. For ALchemist, the space overhead denotes the ratio of aggregated application log size over the audit log size. For MPI and BEEP, the space overhead denotes the size of the additional events emitted by instrumentation over the audit log size. Observe that for complex applications such as *firefox*, our system introduces much less overhead compared to MPI and BEEP. For *firefox*, our system introduces 7.11% overhead while MPI introduces 18.20% overhead and BEEP introduces 42.16% overhead. This is because the instrumentation is quite low level such that a high level event (i.e., one entry in the application log) may give rise to a large number of instrumented events. We also evaluate the whole system overhead in real world scenario. With one week of normal workload, our system on average generates 15.8GB logs with 1.7GB application logs. Fig. 13 shows the space consumption over time for one of the machines.

Runtime Overhead. To measure runtime overhead, we created a set of normal workloads for individual applications, representing typical use cases, such as browsing websites and downloading files in *firefox*. We use *ab* [3] to simulate *apache* workload and a UI input simulation tool `xdotool` [9] to scriptize keyboard and mouse activities. The results are shown in Fig. 14. Here the original applications with audit logging turned on serve as the baseline. Observe that for most applications ALchemist has the lowest overhead as its runtime overhead comes only from application logging. For 4 applications such as *firefox* and *transimission*, MPI has lower overhead as it only instruments places that are critical for causality, whereas the application logs record additional information such as application performance statistics. The more important message here is that all these methods, including ALchemist, have very small overhead.

C. Datalog Inference Overhead (RQ2)

The analysis overhead of ALchemist is dominated by the Datalog engine. Recall that ALchemist is demand-driven and only performs inference on log entries related to attacks. Table V shows the important statistics for Datalog inference for the 14 attacks. The first column shows the attacks. The second column shows that how many raw log entries, including both audit log entries and application log entries, are consumed, with and without demand-driven analysis. For instance, 6.6M/291K (1st row) means that without demand-driven, 6.6M tuples have to be processed and with demand-driven, they

TABLE V: Datalog inference details of attacks

Attack	Tuple(#)	Rules(#)	Relations(#)	Time(s)	Memory(MB)
1	6.6M/291K	73.8M/3.2M	16.4M/1.2M	40.0 / 1.1	262 / 40
2	313K/95K	1.74M/1.16M	554K/429K	0.6 / 0.5	23 / 17
3	83M/1.76M	-/347.26M	-/11.96M	- /88.5	- /217
4	39M/800K	-/59.48M	-/3.74M	- /23.2	- / 95
5	149M/3.06M	-/221.85M	-/22.2M	- /70.6	- /384
6	3.6M/181.8K	494M/9.18M	11.7M/1.89M	106.0 / 2.7	178 / 46
7	2.58M/155.36K	106M/6.61M	1.50M/854K	42.3 / 2.3	38 / 28
8	10.8M/697.82K	544M/15.48M	79.4M/2.49M	136.0 / 3.3	1180 / 53
9	5.37M/154.83K	184.7M/31.65M	14.2M/12.2M	46.4 /12.5	191 /154
10	17.5M/730.82K	-/46.57M	-/3.45M	- /12.1	- / 79
11	7.37M/2.12M	-/281.62M	-/6.87M	- /97.3	- /409
12	5.07M/1.25M	-/223.71M	-/9.33M	- /99.0	- /370
13	4.30M/1.05M	-/151.9M	-/4.25M	- /84.5	- /267
14	4.02M/521K	-/161.6M	-/5.72M	- /95.4	- /145

are reduced to 291K. The third column reports the number of applications of inference rules (with and without demand-driven). Symbol '-' means timeout (10 hours) or out of memory. The fourth column shows the number of derived relations; the fifth column time consumed and the last column memory consumed. The results indicate the necessity of the demand-driven strategy. Observe that in a complex attack 5 (involving complex *firefox* behaviors), the inference engine applies over 220 million rules, deriving 22.2 million new relations. The corresponding runtime overhead is only 70 seconds while the space overhead is only 384MB, demonstrating the practicality of ALchemist in attack forensics.

D. Effectiveness in Execution Partitioning (RQ3)

We conduct an experiment to evaluate the effectiveness of log fusion for execution partitioning. For each application, we craft a special workload that represents the most commonly seen background activities of the application. Each workload represents multiple independent tasks (i.e., units), each task having substantial background activities. We first run the tasks one by one with complete separation to acquire the ground-truth (i.e., which unit an event belongs to). Then we execute these tasks again in parallel, inducing maximum interleaving, and then evaluate the precision and recall of ALchemist. Here, precision means that how many unit attributions identified by ALchemist are correct and recall means that how many correct unit attributions are reported by ALchemist. In order to compare with the ground truth, we suppress non-determinism by hosting resources on local servers and avoiding dynamic contents (e.g., dynamic Ads in *firefox*).

For instance in the workload of *firefox*, we use *HTTrack* to crawl 10 common websites, e.g., `yahoo.com` and `CNN.com`, including all the content pages, CSS, and JS to a local folder and then host these sites locally. We then use a command line to open each site in a tab, e.g., "`firefox -new-tab -url CNN.com`" to open `CNN.com`, and collect the corresponding logs. All the log entries belong to the same unit. We do this for the ten sites and acquire the ground truth.

Then, we use "`firefox -new-tab -url CNN.com -new-tab -url ...`" to open the 10 sites simultaneously, causing maximum interleaving. Then we use ALchemist to partition application logs and attribute audit events. Recall in Section I we discussed that OmegaLog [33] also partitions execution by using repetitive control flow paths (recovered from application logs) to approximate units. We hence also run the same experiment on OmegaLog for comparison.

The second row in Table VI presents the results for *firefox*. The second and third columns denote the audit log size and the corresponding application log size. The 4th, 5th, 6th, 7th, and 8th columns denote the number of raw event entries, rule applications, derived relations, inference time, and memory overhead. The 9th and 10th columns denote precision and recall for ALchemist, whereas the 11th and 12th columns for OmegaLog. NoDoze does not partition execution. Observe that ALchemist's precision for *firefox* is 99.7%, with only 16.9k events mis-attributed, while OmegaLog does not support applications with asynchronous background behaviors. Further inspection shows that *firefox* regularly updates backup files like `sessionstore.jsonlz4`. Our system attributes these updates to a tab instead of the *firefox* process. The recall of *firefox* is 96.8%. The reason for missing entries is the non-determinism of *firefox* execution beyond our control. Specifically, different *firefox* executions use different temporary files to communicate with other applications (e.g., `/tmp/dbus-XXX`, with `XXX` a random string). Hence, the file names in the re-execution are different from those in the ground truth. The other applications have similar results. Observe that ALchemist has 100% precision and recall for many of them, denoting perfect partitioning. In contrast, for the applications it supports, OmegaLog has good precision and recall for many of them except two, where the log messages do not have sufficient information to recover precise paths. The details of the workloads, the raw logs, and the derived relations are posted on [2] for reproduction.

E. Effectiveness in Attack Forensics (RQ4)

To answer RQ4, we use ALchemist, the reimplemented NoDoze [31] and OmegaLog [33] to generate provenance graphs for the 14 APT attacks. During NoDoze attack forensics, each event is assigned an anomaly score based on its frequency (when compared to the normal profile). Then the anomaly score is propagated during causal path traversal. In this way, an anomaly score can be computed for each path. Paths having a high score (i.e., likely anomaly) are reported. Then we compare the generated graphs with the precise ground truth attack graphs (manually marked based on the attack steps) to calculate the True Positive (TP), True Negative (TN), False Positive (FP), False Negative (FN), Precision and Recall values. In contrast, OmegaLog recovers repetitive control flow paths from application logs and use these paths as the approximation of the execution units. An output event in a unit is considered dependent on all the preceding inputs in the unit. We tune our reimplementations (e.g., the threshold for anomaly) until we can achieve similar performance as their published results. The results are summarized in Table VII. The table contains the following information: columns 2∼3 for the ground truth number of attack related units and normal units (note that attack steps may interleave with benign activities); columns 4∼7 for the number of attack related and normal units determined by ALchemist (a unit is considered attack related if it is in the attack provenance graph), FP and FN compared with the ground truth; columns 8∼9 for the ground truth numbers of attack related events and normal events, each including both the number of audit events (L) and the number of application log events (H); columns 10∼13 the number of FPs and FNs (of the events in attack provenance graph) by ALchemist, the precision, and recall; columns 14∼17 the FPs and FNs (only at the audit log level), precision and recall by NoDoze, which does not use application log. The last four

TABLE VI: Execution partitioning on asynchronous normal workloads ('-' entries are because OmegaLog does not support applications with asynchronous background behaviors)

Program	Audit Size	App Size	Tuples(#)	Rules(#)	Relations(#)	Time(s)	Memory(MB)	ALchemist Result		OmegaLog Result	
								Precision	Recall	Precision	Recall
Firefox	2.6GB	241.8MB	5.4M	139.0M	22.0M	107.1	525	99.7%	96.8%	-	-
Chromium	1.6GB	71.1MB	1.9M	77.6M	12.7M	99.7	477	99.8%	96.2%	-	-
LibreOffice	513.0MB	5.7MB	1.1M	46.2M	8.3M	87.5	381	99.8%	97.7%	-	-
OpenOffice	487.6MB	3.1MB	382K	13.7M	2.6M	61.7	167	99.6%	99.5%	-	-
Vim	389.0MB	2.5MB	774K	8.9M	2.0M	15.9	174	100.0%	100.0%	93.1%	93.1%
Apache	282.4MB	15.3MB	529K	4.8M	1.4M	10.5	119	100.0%	100.0%	100.0%	100.0%
Nginx	205.2MB	11.2MB	401K	2.1M	512K	5.0	92	100.0%	100.0%	100.0%	100.0%
Pure-ftpd	388.1MB	6.5MB	832K	5.9M	1.5M	12.5	168	100.0%	100.0%	100.0%	100.0%
Vsftpd	491.1MB	9.2MB	1.1M	12.6M	2.5M	21.0	191	100.0%	100.0%	100.0%	100.0%
Proftpd	338.5MB	4.7MB	717K	7.4M	1.6M	14.9	130	100.0%	100.0%	100.0%	100.0%
TightVNC	402.4MB	7.9MB	839K	6.2M	2.2M	13.7	176	100.0%	100.0%	100.0%	100.0%
Foxit	63.6MB	1.1MB	110K	757K	243K	1.2	29	99.3%	98.0%	-	-
Openssh	186.1MB	1.6MB	425K	3.0M	1.3M	8.1	110	100.0%	100.0%	100.0%	100.0%
Transmission	1.2GB	17.6MB	2.6M	20.2M	7.1M	42.0	358	98.9%	97.6%	74.1%	72.8%

TABLE VII: Forensic results (APG, L, and H stand for attack provenance graph, audit level, and application level, respectively)

Attack No.	|Unit| Attack	(g-truth) Normal	|Unit| TP	ALchemist TN	FP	FN	|Event| Attack	(g-truth) Normal	(L/H)	|Event| FP	FN	in ALchemist APG Precision	Recall	|Event| FP	FN	in NoDoze APG (L only) Precision	Recall	|Event| FP	FN	in OmegaLog APG (L/H) Precision	Recall
1	2	24	2	24	0	0	1043/10	290K/618		25/0	0/0	97.6%/100%	100%/100%	12	0	98.9%	100%	92/4	37/2	91.9%/71.4%	96.6%/83.3%
2	3	17	3	17	0	0	65/13	95K/212		5/0	0/0	92.8%/100%	100%/100%	3	0	95.6%	100%	16/0	0/0	80.2%/100%	100%/100%
3	4	158	4	158	0	0	2687/1132	1.78M/33K		279/58	0/0	90.6%/95.1%	100%/100%	1191	785	69.3%	77.4%	-	-	-	-
4	8	97	8	97	0	0	1028/530	1.19M/28K		150/36	0/0	87.3%/93.6%	100%/100%	792	565	56.5%	64.5%	-	-	-	-
5	12	354	12	354	0	0	3874/859	4.05M/58K		357/83	0/0	91.6%/91.2%	100%/100%	1516	698	71.9%	84.7%	-	-	-	-
6	5	45	5	45	0	0	114/40	181K/1463		12/0	0/0	90.4%/100%	100%/100%	18	0	86.4%	100%	12/0	0/0	90.4%/100%	100%/100%
7	5	74	5	74	0	0	112/37	155K/680		17/0	0/0	86.8%/100%	100%/100%	13	24	87.1%	78.6%	-	-	-	-
8	7	67	7	67	0	0	307/26	697K/297		26/0	28/0	91.4%/100%	90.8%/100%	22	37	92.5%	87.9%	-	-	-	-
9	5	267	5	267	0	0	73/19	154K/5349		5/0	0/0	93.5%/100%	100%/100%	4	0	94.8%	100%	14/0	0/0	83.9%/100%	100%/100%
10	7	136	7	136	0	0	926/283	1.07M/20K		115/39	0/0	89.0%/87.9%	100%/100%	405	213	69.6%	81.3%	-	-	-	-
11	11	541	11	541	0	0	285/35	1.36M/33K		7/0	0/0	97.6%100%	100% 100%	4	2	98.6%	99.3%	20/0	4/1	93.4%/100%	98.6%/97.2%
12	2	12	2	12	0	0	211/9	1.22M/21		13/0	0/0	94.2%/100%	100%/100%	21	0	90.9%	100%	13/0	0/0	94.2%/100%	100%/100%
13	5	43	5	43	0	0	101/20	1.50M/44		4/0	0/0	96.2%/100%	100%/100%	2	2	98.1%	98.1%	9/0	0/0	91.8%/100%	100%/100%
14	6	12	6	12	0	0	656/17	511K/47		0/0	0/0	100%/100%	100%/100%	0	0	100%	100%	0/0	0/0	100%/100%	100%/100%
Avg.	6	132	6	132	0	0	820/216	987K/13K		73/13	2/0	92.8%/97.7%	99.6%/100%	286	166	86.4%	90.8%	22/1	5/1	90.7%/96.4%	99.4%/97.6%

columns are the same information for OmegaLog that uses both application and audit logs. Our experiments show that ALchemist can precisely identify all the attack-related units in the 14 attacks. At the individual event level, ALchemist can achieve 92.8% precision and 99.6% recall for audit records, whereas NoDoze can achieve 86.4% precision and 90.8% recall and OmegaLog achieves 90.7% precision and 99.4% recall (for the subset of attacks it supports). ALchemist has some false positives for attacks 4 and 7 that involve *firefox* and rootkit *Azazel* with highly asynchronous execution and sophisticated process injection techniques. As such, some of the low level events are attributed to the wrong unit by the forward attribution rule. These events do not have clear projection to the application log. In contrast, NoDoze does not have good accuracy for attacks 3, 4, 5, and 10 that involve *firefox* and *libreoffice* with a lot of background behaviors. As illustrated by the example in Fig. 11, NoDoze cannot prevent bogus dependencies caused by asynchronous background behaviors. It mistakenly introduces dependencies between foreground unit and the completely unrelated background unit as it cannot distinguish them. NoDoze misses important events in 6 out of 10 attacks and the recall for attacks 3, 4, 5 are low. This is because the attacks leveraged frequently executed apps and dependencies, which are mistakenly excluded by NoDoze. These events are critical in understanding attack provenance. We also want to point out that NoDoze requires collecting execution profile and training whereas ALchemist does not. Omega does not handle applications that have background

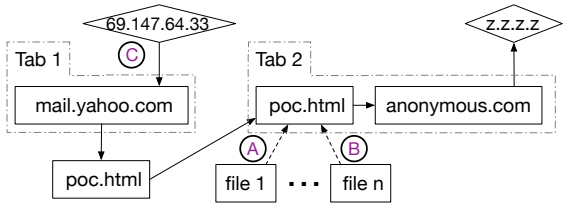

Fig. 15: Causal graph of attack #4 by ALchemist

behaviors and hence the provenance graphs for attacks 3, 4, 5, 7, 8 and 10 explode (and hence the '-' values). It works well for the other attacks. Observe that ALchemist consistently outperforms.

VI. Case Studies

We use two representative attacks: Exfiltration (attack # 4) and Azazel Injection (attack # 7) to demonstrate the effectiveness of ALchemist in comparison with NoDoze. They demonstrate how NoDoze misses critical attack steps such as those related to commonly visited *mail.yahoo.com* and the */home/user* folder.

A. Attack #4: Exfiltration

Attack Scenario. The attacker sends an email with a malicious attachment to the victim. The victim downloads the malicious HTML file to "`file:///home/user/poc.html`". Then the victim opens the HTML file in *firefox*. Based on the *same origin policy* [12] by *firefox*, `poc.html` has the access to all files in a folder if one of its DOM objects has access to these

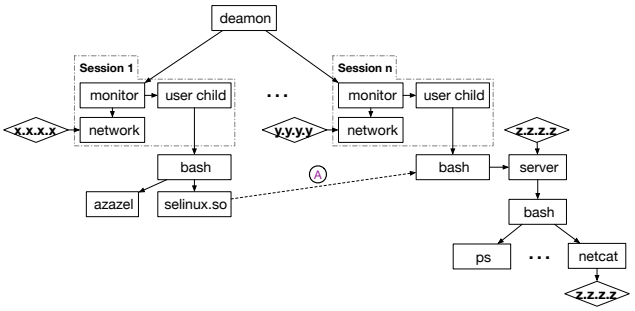

Fig. 16: Causal graph for the Azazel attack by ALchemist

files. Then attacker uses ClickJacking [5] to deceive the victim into clicking a button on the malicious HTML. The victim believes he clicks on a link to a remote page, but in fact he is clicking on the iframe's directory "`file://home/user/`", allowing `poc.html` to gain access to all the files in the directory. Finally, the malicious page sends requests with the stolen information and navigates to the attacker's website `anonymous.com` (with IP `z.z.z.z`).

Threat Alert. The suspicious connection to `z.z.z.z` is subsequently detected by a local network monitoring software *Nogios*, which leads to the investigation of the attack.

Attack Investigation. Fig. 15 presents the causal graph by ALchemist. Observe that it precisely captures the attack provenance with tab 1 downloading the attachment from `mail.yahoo.com` (IP 69.147.64.33) and tab 2 exfiltrating files. In contrast, NoDoze misses the root cause Ⓒ and the exfiltration of files (e.g., Ⓐ and Ⓑ). In particular, the IP of `mail.yahoo.com` is frequently visited by *firefox* and hence the network connection is precluded. Furthermore, the exfiltration happens on preference files in the `/home/user` folder frequently visited by *firefox* during normal operation. As such, they are precluded as well. Hence, from NoDoze's graph, the inspector may not understand the damages caused by the attack, nor does he understand where the attack was from. In addition, the navigation from `poc.html` to `anonymous.com` is also unclear from NoDoze's graph. In ALchemist, rules (R1) is used during the Datalog inference phase to correlate *firefox* event "`GET anonymous.com/index.html*?data=...`" with the system event "`connect(z.z.z.z)`". Through *firefox* events, ALchemist reconstructs the navigation relation from `/home/user/poc.html` to `anonymous.com`. The accesses to "`file 1`" and "`file n`" are invisible at the application log level. But they can be seen at the audit level. Our forward attribution rule (R11) allows attributing these audit events to the appropriate tab.

B. Attack #7: Azazel Attack

Attack Scenario. The attacker connects to a host via SSH using stolen credentials. Then an open-source rootkit *Azazel* and its shared object package *selinux.so* were uploaded using *scp*. In order to avoid creating a large number of events during a short period, the attacker terminates the current sshd session. Sometime later, the attacker uses other stolen credentials to start a new sshd session and executes command line "`export LD_PRELOD = selinux.so`" in *bash* to set the $LD\_PRELOAD$ environment variable to the downloaded *selinux.so*. Then the attacker starts a *server* process listening on port 4444. As such, *selinux.so* is injected to the newly launched process. By hooking the commonly used function `accept()`, *selinux.so* enables the attacker to drop a shell remotely from IP `z.z.z.z`. Then the attacker can execute multiple recon commands to collect and send back credential information a few times.

TABLE VIII: Comparison with closely related works that aim to solve the dependency explosion problem

	BEEP [45]	MPI [53]	MCI [43]	UIScope [73]	OmegaLog [33]	NoDoze [31]	ALchemist
Instrumentation	✓	✓	✗	✗	✗	✗	✗
Training Run	✓	✗	✓	✗	✗	✗	✗
Server App.	✓	✓	✓	✗	✓	✓	✓
GUI App.	✓	✓	✓	✓	✗	✓	✓
App. Semantics	✗	✗	✗	✗	✓	✗	✓

Threat Symptom. The suspicious connection to `z.z.z.z` is subsequently detected by a local network monitoring software (e.g. *Nogios*), which leads to the attack investigation.

Attack Investigation. To investigate this attack, the inspector first obtains the logs (including both app and audit logs), apply Datalog inference and construct the graph from the symptom event (i.e., the connection to `z.z.z.z`). The graph is shown in Fig. 16. Note that in this attack, the daemon forks a monitor process for each external connection. For the purpose of avoiding privilege escalation, the monitor further forks child processes to handle individual tasks (e.g., network authentication/communication). The application log helps ALchemist to group sshd processes into sessions (one for each connection request). In this way, starting from the symptom `z.z.z.z`, ALchemist first back-traces to a sshd session n. NoDoze can also achieve this due to the rarely visited IP. However, setting $LD\_PRELOAD$ is invisible at the audit log level while it is recorded by applications (e.g., *bash* and *firefox*). As a result, NoDoze misses this attack step due to the missing dependence in Ⓐ whereas ALchemist precisely captures it and then the root cause. Furthermore, since loading *selinux.so* is considered a normal activity by NoDoze according to the execution profile, it misses the root cause as well.

VII. Related Work

ALchemist is related to data provenance [65], [17], [62], [21], [15], [64], [37], [55], [30] [69], audit logging [57], [41], [40], [62], [18], [61], [36], [75], [74], log parsing [79], [70], [58], [34], [66], [26] and causality analysis [80], [42], [41], [40], [35], [45], [52], [53], [54], [43] [77], [78]. Some of them suffer from the *dependency explosion problem* [41], [42]. Some require instrumentation [53], [45], [54], which is not practical for deployment in enterprises. Many techniques utilize learning/profiling to derive a reference model to detect abnormal events [49], [31], [59], [70], [22], [31], [32], [47], [68], [67], [71], [35], [23], [63], [72], [60], [76], [48]. As discussed in Section I, these methods may be bypassed if the attacker uses spoofing techniques to hide their activities. UIScope [73] intercepts UI events and correlates them with audit events to construct attack graphs. It focuses on UI apps and does not leverage application specific semantics. It is Windows based and hence cannot be empirically compared with ALchemist. In contrast, ALchemist does not require instrumentation or pre-trained models. It performs log fusion on application logs

and audit log to address the dependency explosion problem. Table VIII summarizes the differences between ALchemist and a few closely related works that address the dependence explosion problem. Some works propose to support better forensics analysis using graph queries [61], [25], [24] and efficient storage [29]. Forensic analysis can be extended to other tasks [56]. Such techniques are complementary to ALchemist.

Zhang et al. [77], [78] use rule- and learning-based methods to infer causal relationship between network events. Then they devise user-intention based security policies to pinpoint stealthy malware activities based on the relations. These rules can potentially be rewritten in datalog to enhance ALchemist's capabilities. Xu et al. [69] propose an efficient cryptographic protocol that ensures the correct origin or provenance of critical system information and prevents adversaries from utilizing host resources.

Application logs have been studied in recent years. Ghoshal et al. [27] utilize a rule specification to generate structured provenance events by processing application log. The proposed approach, however, only focuses on application logs without considering system logs or log fusion. The experiments were only conducted on 5 simple applications. The provenance system designed by Chen et al. [20] uses a graph recorder, which extracts provenance for applications written in a specific declarative language or instrumented in source code. ALchemist does not require any specific language or instrumentation.

VIII. Conclusion

We propose a novel forensics technique ALchemist. It leverages that built-in application logs and audit log are complementary and in the mean time share a lot of common elements, which can be utilized for log fusion. A set of parsers are developed to parse various kinds of logs to their canonical representations. Datalog based fusion rules are applied to bind these logs and more importantly, to derive new information that is invisible from either kind of the logs. Our evaluation shows that ALchemist is highly effective in partitioning execution to units and producing precise attack provenance graphs, without requiring any instrumentation. It also outperforms state-of-the-art techniques with and without instrumentation.

Acknowledgment

We thank our shepherd, Daphne Yao, and those anonymous reviewers for their comments and suggestions. This research was supported, in part by NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947, and Sandia Lab MOD1-18046142. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] "50 essential linux applications," https://tinyurl.com/wcj5og2.

[2] "Alchemist2020/workload," https://github.com/ALchemist2020/Workload.

[3] "Apache http server benchmarking tool," https://tinyurl.com/onkcat3.

[4] "Apt groups," https://tinyurl.com/y2tqt74o.

[5] "Clickjacking," https://tinyurl.com/62thhvp.

[6] "darpa-i2o/transparent-computing: Darpa transparent computing program," https://github.com/darpa-i2o/Transparent-Computing.

[7] "Datalog," https://tinyurl.com/cam64up.

[8] "Elasticsearch," https://tinyurl.com/y3uv7342.

[9] "jordansissel/xdotool: fake keyboard/mouse input, window management, and more," https://github.com/jordansissel/xdotool.

[10] "Kernel path protection," https://tinyurl.com/7ttl5h2.

[11] "Nspr log modules," https://tinyurl.com/gmlmtnz.

[12] "Same-origin policy," https://tinyurl.com/pp86n9a.

[13] "Splunk," https://www.splunk.com.

[14] "Sysdig," https://tinyurl.com/nzrexz5.

[15] S. T. Ali, V. Sivaraman, D. Ostry, G. Tsudik, and S. Jha, "Securing first-hop data provenance for bodyworn devices using wireless link fingerprints," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 12, pp. 2193–2204, 2014.

[16] S. Avizheh, T. T. Doan, X. Liu, and R. Safavi-Naini, "A secure event logging system for smart homes," in *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, 2017, pp. 37–42.

[17] M. Backes, S. Bugiel, and S. Gerling, "Scippa: system-centric ipc provenance on android," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 36–45.

[18] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 319–334.

[19] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *USENIX Security Symposium*, 2015, pp. 319–334.

[20] A. Chen, Y. Wu, A. Haeberlen, B. T. Loo, and W. Zhou, "Data provenance at internet scale: Architecture, experiences, and the road ahead," in *Proceedings of 8th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[21] C. Collberg, A. Gibson, S. Martin, N. Shinde, A. Herzberg, and H. Shulman, "Provenance of exposure: Identifying sources of leaked documents," in *2013 IEEE Conference on Communications and Network Security (CNS)*, 2013, pp. 367–368.

[22] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298.

[23] T. Dumitras and I. Neamtiu, "Experimental challenges in cyber security: A story of provenance and lineage for malware." in *Proceedings of 4th Workshop on Cyber Security Experimentation and Test (CSET)*, 2011.

[24] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "{SAQL}: A stream-based query system for real-time abnormal system behavior detection," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 639–656.

[25] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "{AIQL}: Enabling efficient attack investigation from system monitoring data," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 113–126.

[26] Y. Gao, S. Huang, and A. Parameswaran, "Navigating the data lake with datamaran: Automatically extracting structure from log datasets," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 943–958.

[27] D. Ghoshal and B. Plale, "Provenance from log files: a bigdata problem," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 2013, pp. 290–297.

[28] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara, "The taser intrusion recovery system," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, 2005, pp. 163–176.

[29] Z. Guo, H. Lin, M. Yang, D. Zhou, F. Long, C. Deng, C. Liu, and L. Zhou, "G2: A graph processing system for diagnosing distributed systems," 2011.

[30] R. Hasan, R. Sion, and M. Winslett, "The case of the fake picasso: Preventing history forgery with secure provenance." in *FAST*, vol. 9, 2009, pp. 1–14.

[31] 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.

[32] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *Network and Distributed Systems Security Symposium*, 2018.

[33] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis," in *NDSS*, 2020.

[34] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2017.

[35] 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 *Proc. USENIX Secur.*, 2017, pp. 487–504.

[36] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1705–1722.

[37] Y. Ji, S. Lee, and W. Lee, "Recprov: Towards provenance-aware user space record and replay," in *International Provenance and Annotation Workshop*, 2016, pp. 3–15.

[38] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*, 2016, pp. 422–430.

[39] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Acm Sigplan Notices*, vol. 47, no. 7, 2012, pp. 121–132.

[40] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," 2010.

[41] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 223–236, 2003.

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

[43] 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 *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS). The Internet Society, San Diego, California, USA*, 2018.

[44] K. H. Lee, X. Zhang, and D. Xu, "Loggc: garbage collecting audit log," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1005–1016.

[45] ——, "High accuracy attack provenance via binary-based execution partition." in *NDSS*, 2013.

[46] B. Li, "Enabling fine-grained reconstruction and analysis of web attacks with in-browser recording systems," Ph.D. dissertation, uga, 2017.

[47] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 102–111.

[48] F. Liu, Y. Wen, D. Zhang, X. Jiang, X. Xing, and D. Meng, "Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1777–1794.

[49] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS). The Internet Society, San Diego, California, USA*, 2018.

[50] D. Ma, "Practical forward secure sequential aggregate signatures," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 2008, pp. 341–352.

[51] D. Ma and G. Tsudik, "Forward-secure sequential aggregate authentication," in *2007 IEEE Symposium on Security and Privacy (SP'07)*, 2007, pp. 86–91.

[52] 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 *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 401–410.

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

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

[55] P. McDaniel, "Data provenance and security," *IEEE Security & Privacy*, vol. 9, no. 2, pp. 83–85, 2011.

[56] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," *arXiv preprint arXiv:1810.01594*, 2018.

[57] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 43–56.

[58] M. Nagappan, K. Wu, and M. A. Vouk, "Efficiently extracting operational profiles from execution logs using suffix arrays," in *2009 20th International Symposium on Software Reliability Engineering*, 2009, pp. 41–50.

[59] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 26–26.

[60] A. Oprea, Z. Li, T.-F. Yen, S. H. Chin, and S. Alrwais, "Detection of early-stage enterprise infection by mining large-scale log data," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 45–56.

[61] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime analysis of whole-system provenance," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1601–1616.

[62] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: collecting high-fidelity whole-system provenance," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 259–268.

[63] A. Ramachandran, K. Bhandankar, M. B. Tariq, and N. Feamster, "Packets with provenance," Georgia Institute of Technology, Tech. Rep., 2008.

[64] A. Ramachandran and M. Kantarcioglu, "Smartprovenance: a distributed, blockchain based dataprovenance system," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 35–42.

[65] M. Stamatogiannakis, E. Athanasopoulos, H. Bos, and P. Groth, "Prov 2r: practical provenance analysis of unstructured processes," *ACM Transactions on Internet Technology (TOIT)*, vol. 17, no. 4, p. 37, 2017.

[66] S. Thaler, V. Menkonvski, and M. Petkovic, "Towards a neural language model for signature extraction from forensic logs," in *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*, 2017, pp. 1–6.

[67] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long, "A hybrid approach for efficient provenance storage," in *Proceedings of the 21st ACM international conference on Information and knowledge management*, 2012, pp. 1752–1756.

[68] Y. Xie, K.-K. Muniswamy-Reddy, D. D. Long, A. Amer, D. Feng, and Z. Tan, "Compressing provenance graphs." in *TaPP*, 2011.

[69] K. Xu, H. Xiong, C. Wu, D. Stefan, and D. Yao, "Data-provenance verification for secure hosts," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 2, pp. 173–183, 2011.

[70] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[71] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 504–516.

[72] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu, "Using provenance patterns to vet sensitive behaviors in android apps," in *International Conference on Security and Privacy in Communication Systems*, 2015, pp. 58–77.

[73] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen, "Uiscope: Accurate, instrumentation-free, deterministic and visible attack investigation," in *NDSS*, 2020.

[74] A. A. Yavuz, P. Ning, and M. K. Reiter, "Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging," in *International Conference on Financial Cryptography and Data Security*, 2012, pp. 148–163.

[75] A. A. Yavuz and P. Ning, "Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems," in *2009 Annual Computer Security Applications Conference*, 2009, pp. 219–228.

[76] T.-F. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. Robertson, A. Juels, and E. Kirda, "Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 199–208.

[77] H. Zhang, D. D. Yao, and N. Ramakrishnan, "Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 39–50.

[78] H. Zhang, D. D. Yao, N. Ramakrishnan, and Z. Zhang, "Causality reasoning about network events for detecting stealthy malware activities," *computers & security*, vol. 58, pp. 180–198, 2016.

[79] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 121–130.

[80] N. Zhu and T.-c. Chiueh, "Design, implementation, and evaluation of repairable file service," in *2003 International Conference on Dependable Systems and Networks (DSN)*, 2003, p. 217.

APPENDIX

A. Stability Study of Application Built-in Logging Modules

We study the stability of application built-in logging modules. The results are shown in Table IX. Column 1 presents the name for the logging facilities. Note that the same logging facility may be used by multiple applications. The second column shows the applications. Column 3 shows the number of regular expressions we implemented to parse the log. Columns 4-5 present the two versions whose built-in logs are compared. Column 6 indicates the new log types added (in the new version) and column 7 presents the number of regular expressions we have to change, that is, the log types are the same but the formats are changed. Observe that most of them are fairly stable. Even for *firefox* that has gone through major code change, the logging module has only small changes.

B. Study of Top 30 Linux Application Built-in Logging

We study 32 Linux applications, including 30 most popular applications listed in [1] and 15 complex applications widely used in the APT attack literature. We want to analyze their execution models and check if these applications have built-in logging module and if their logs contain information to disclose the underlying execution model, especially implicit/-explicit unit boundaries, which are the most critical information for execution partitioning. Here, implicit boundaries mean that they can be inferred by log fusion. Column 1 shows the applications. Column 2 presents if the application has built-in logging facility. Column 3 presents the execution unit structure for the application. Column 4 shows if the application log contains information to separate different units. Column 5 shows the execution model(discussed in Section III) used by the application. From the table, 28 out of 32 applications are long running and 29 out of 32 have built-in logging facility and support unit partitioning. For UI programs, their unit structures have the following categories. Web applications (e.g. firefox and chromium) have tabs as their execution units. For example, Chromium's built-in log uses a same connection id to denote all sub-tasks originated from the same tab, which is very similar to the transaction id in firefox, allowing tracking causality in its complex asynchronous execution model. Editor applications (e.g. office, text editor, and graphic editor) have individual windows and files as units. Shell programs (e.g. bash and zsh) have a history file that records all the interactive commands and individual commands can hence be considered as different units. For server programs, each connection is considered as a unit. Class I execution model (sequential single process) is widely used by editors and firewall applications. Class II (process forking) is used in simple UI programs (e.g. gscan2pdf) and ftp servers. Asynchronous models III, IV, and V are commonly used by complex UI programs such as Thunderbird and Geary.

TABLE IX: Change of application logging over years

Logging Facilities	Applications	Total	version1	version2	Semantic Change	Syntax Change
NSPR	firefox	719	42.0(2015)	60.0(2018)	28	52
	thunderbird	719	42.0(2015)	60.0(2018)	28	52
ChromeLog	chromium	657	46.0(2015)	64.0(2018)	47	84
OfficeLog	libreoffice	64	4.4(2015)	6.0(2018)	16	41
	openoffice	70	4.1.2(2015)	4.1.6(2018)	0	0
VimLog	vim	109	8.0.0(2016)	8.1.0(2019)	6	3
HttpLog	nginx httpd	20	1.9.0(2015)	1.15.0(2018)	0	0
	apache httpd	20	2.4.12(2015)	2.4.32(2018)	0	0
FtpLog	vsftpd	18	2.3.5(2011)	3.0.3(2015)	0	0
	pure-ftpd	18	1.0.37(2015)	1.0.47(2018)	0	0
SshLog	sshd	26	7.0(2015)	7.9(2018)	0	0
VncLog	tightvnc	30	2.7.10(2013)	2.8.11(2018)	0	0
ShellLog	bash	8	4.3.11(2013)	5.0(2018)	0	0
PdfLog	foxit	54	2.4.1(2015)	2.4.4(2018)	0	0
PlayerLog	mplayer	20	1.1.0(2012)	1.3.0(2016)	0	2

TABLE X: Built-in logging study for top 30 popular Linux applications in daily usage and top 15 Linux applications in APT attacks (13 are shared by the two sets). Execution models I,II,III,IV,V are those in Section III.

	Application	Has Built-in Logging	Unit	Log of Unit Boundary	Execution Model
UI Program	Thunderbird	Yes	Conversation Thread	Yes	III,IV,V
	Geary	Yes	Conversation Thread	Yes	III,IV,V
	WizNote	Yes	Note	Yes	III,IV
	Chromium	Yes	Tab	Yes	III,IV,V
	Firefox	Yes	Tab	Yes	III,IV,V
	FileZilla	Yes	Connection	Yes	III,IV
	OpenOffice	Yes	File/Window	Yes	III,IV
	LibreOffice	Yes	File/Window	Yes	III,IV
	KeePass*	No	/	/	/
	gscan2pdf*	Yes	Document	Yes	II
	WINE	Yes	Guest Application	Yes	III,IV,V
	VirtualBox	Yes	Guest Environment	Yes	III,IV,V
	Skype	Yes	Chat Thread	Yes	III,IV
	DropBox	Yes	Folder	Yes	III,IV
	Gimp	Yes	Window	Yes	III,IV
	Bash	Command History	Command	Yes	II
	Zsh	Command History	Command	Yes	II
	Nmap	Yes	Connection	Yes	II
	Zsh	Command History	Command	Yes	II
	MPlayer	Yes	Connection	Yes	IV
	Vim	Yes	Buffer/Window	Yes	I
	Emacs	Yes	Buffer/Window	Yes	I
Server	Apache	Yes	Connection	Yes	IV,V
	Nginx	Yes	Connection	Yes	IV,V
	Lighttpd	Yes	Connection	Yes	IV,V
	TightVNC	Yes	Connection	Yes	II
	Openssh	Yes	Connection	Yes	II
	Pure-ftpd	Yes	Connection	Yes	II
	Vsftpd	Yes	Connection	Yes	II
	Proftpd	Yes	Connection	Yes	II
	FileZilla	Yes	Connection	Yes	II
	UFW	Yes	Connection	Yes	I

* Not-long running applications.