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

Anonymous

ABSTRACT

Cyber-attacks are becoming more persistent and complex. State-of-the-art attack forensics techniques either require annotating and instrumenting software applications or rely on high quality execution profile 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 log provides low-level fine-grained information; and the two share a lot of common elements. ALchemist is hence a log fusion technique that organically couples application logs and audit log 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*), even in the presence of complex asynchronous execution models, and causality 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 causality), and providing high level semantics in causal graphs with very small overhead, compared to state-of-the-art techniques based on instrumentation. It also outperforms NoDoze, a state-of-art technique that does not require instrumentation but rather substantial execution profile.

1 INTRODUCTION

Advanced Persistent Threat (APT) is a complex form of attack that contains multiple phases and targets specific organization or institute [7]. A popular method for attack investigation is to perform causality analysis on system audit logs. System audit logs record the events and states during the execution of applications and component in a system. And these logs are widely used in many software engineering tasks such as runtime failures diagnosis [92, 95], performance bottlenecks identification [19, 68] and security analysis [23, 70]. Also, system audit logs can be used to reconstruct attack provenance. In [12, 42, 43], 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* [49, 64]. 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 [49, 64]. However, these approaches require 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 and is often conducted by third parties, it shifts the responsibility of maintaining the correctness of software from its original producer to the third party, which is undesirable. In fact, many companies provide mechanisms to proactively prevent their software from being instrumented such as the Kernel Patch Protection by Microsoft [4]. Another line of work tries to solve the dependency explosion problem by pruning graphs with heuristics such as prioritizing low frequency events [30, 54]. 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).

Our goal is to develop a new attack investigation technique that can achieve the same accuracy as instrumentation based methods without requiring instrumentation. Application log analysis is an important research area in software engineering. As debugging tools are usually inapplicable in production settings [97], application logs have become the main source of information when diagnosing application problems. Specifically, application logs have been widely used in various reliablity enhancement tasks such as anomaly detection [25, 88], fault diagnosis [85, 99], program verification [22, 77], etc. Based on our observations, those applications 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 [64, 65]. On the other hand, the low level system 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, ALchemist, that seamlessly 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 their canonical forms such that their correlations can be inferred. The canonical log entries are loaded into a Datalog engine [39] to derive new relations based on a set of pre-defined rules, which we call the *log fusion rules*. Precise causal 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 individual types of logs. According to our study (Appendix B), log formats rarely change, much less frequently compared to software releases. Note that each software release entails re-instrumentation for instrumentation based methods.
- We develop a comprehensive set of log fusion rules. We study the execution models of a set of popular applications from [49, 63–65] and their built-in application logs, and determine that their executions can be properly partitioned to units by our log fusion rules. 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 93.1% 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 [30], a state-of-the-art technique that does not require instrumentation.

Threat Model. ALchemist has a threat model similar to that in many existing works [15, 30, 49, 50, 65, 71, 75]. 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 [49, 64] that require application instrumentation also trust the (instrumented) applications. Note that as pointed out in [30, 49, 64, 71], 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., [15, 28]) 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 [13, 17, 60–62] such that tampering efforts can be detected.

2 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 the attacker's server *x.x.x.x*. Later, the user executes *fcopy* without realizing that it has been replaced with a malware. 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 from attacks discussed in existing works [30, 64] as it involves JS execution as part of its attack chain, which is difficult for many existing works.

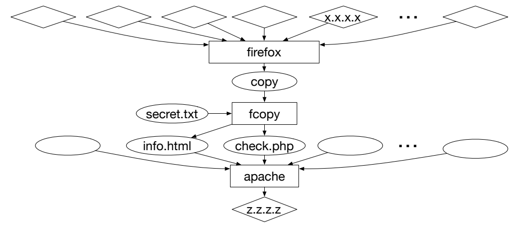


Figure 1: Causal graph by syscall only methods (e.g., [42])

```
16:30:12 [23104] nsObserverService (TranID=0xf4a1d5b80)
UpdateCurrentTopLevelOuterTabId id=200000001
```
(a)

```
21:17:46 [8890] mozStorage ATTACH '~/.thunderbird/INBOX'
21:17:46 [8890] mozStorage (TranID=0x603c9b80)
SELECT * FROM messageAttributes (folderID, messageID)
VALUES (2, piMN2BuJQb)
```
(b)

```
192.168.143.130 [05:09:23] "GET /index.html
HTTP/1.1" 200 151 "-" "Wget/1.19.2 (linux-
gnu)"
```
(c)

```
PATH 21:17:46 name=~/.thunderbird/INBOX
SYSCALL 21:17:46 syscall=open exit=130
ppid=8262 pid=8890 exe=thunderbird
```
(d)

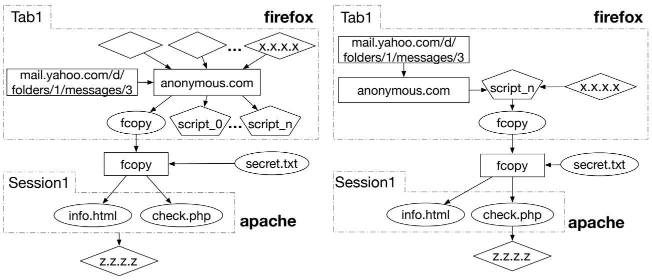
Figure 2: (a) *firefox* tab switch log (b) *thunderbird* email open log (c) *apache* request log (d) *thunderbird* email open audit log

2.1 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) [26, 40, 42]. 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* [49]. 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 (e.g., tabs in *firefox*). 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 back-trace 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*.

2.2 Instrumentation Based Approaches

The root cause of inaccuracy in syscall-only approaches is that these techniques are not aware of application semantics. Therefore, there is a line of work that leverages program instrumentation to expose application semantics to the system logging components [49, 51, 64]. Specifically, they partition an execution to multiple independent *units (tasks)*. For example, MPI [64] requires the user to annotate the tab data structure such that it can instrument all the accesses to this data structure to identify tab switches at runtime. An output syscall is considered only dependent on all the preceding input syscalls *within the same unit*, avoiding dependence explosion. Although these techniques are quite effective, they are either language dependent (e.g., MPI [64]) or cannot deal with virtual environments (e.g., BEEP [49]). Fig. 3 (a) shows the provenance graph of the motivation



(a) MPI graph (b) ALchemist graph

Figure 3: Causal graphs by (a) MPI and (b) ALchemist

example by MPI. Observe that 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. However, the graph only tells that the website has downloaded (i.e., written to) n JS files (*script_0* to *script_n*) and *fcopy*. It is invisible that the execution of *script_n* downloads *fcopy* from *x.x.x.x* due to MPI's difficulty of analyzing and instrumenting JS code. In addition, according to our experience, in many deployment scenarios, instrumentation is not an option, per the policy of the customer (i.e., cannot modify pre-installed systems). Instrumentation-based techniques are also difficult to maintain. Each time a new version of application is released, the annotations and instrumentation have to be redone.

2.3 Our approach

Built-in Application Log Providing Critical High Level Semantics. We observe that built-in application logs provide rich semantics regardless of the programming languages. In our example, the three applications involved, *firefox*, *thunderbird*, and *apache* all have built-in logs that provide sufficient information for precise execution partitioning, which is the key to the success of existing instrumentation based methods. Specifically, *firefox* logs any tab creation and switch, allowing precise identification of execution unit boundaries. As pointed out by existing work [51, 64], *firefox* execution is highly asynchronous. A tab's execution is broken down to smaller tasks (e.g., requesting a page, rendering an image, and executing a JS code blob) that are dispatched to various worker threads, which may further break down the tasks into sub-tasks. To help developers debug and maintain the code base, *firefox* uniquely identifies each atomic task/sub-task internally and logs their creation. 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*. .

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 causality). 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 idea 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. 4, 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 almost a 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 $switchTo(X)$ means that the current *firefox* tab is switched to a new tab X. These basic relations are provided to a Datalog inference engine [39] (in the center of Fig. 4), 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 at the syscall level (in the audit log). Our technique projects it to the high-level email access relation (corresponding to the application log entry in Fig 2 (b)). Since the high level application logs provide a clear execution unit structure, by projecting such a 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 causal graph generated by ALchemist. Observe that it avoids dependence explosion, similar to the graph by MPI (Fig 3 (a)). In addition, it precisely identifies that *fcopy* is generated by *script_n*, which downloads from *x.x.x.x*, as the JS execution is correctly partitioned by ALchemist. In addition to the advantage of without requiring instrumentation, according to our experiments in Section 4.2, our approach introduces much less space overhead and trivial runtime overhead, compared with MPI.

A Study of Built-in Logs for Popular Linux Applications. We conduct a study of 30 most popular Linux applications from [2]. Among them, 15 complex applications that are widely used in the APT attack literature, including *firefox*, *Thunderbird*, *Chromium*, *OpenOffice*, *LibreOffice*, and *Apache*. Our study shows that 28 applications are long running, and 29 have built-in logs. All the 29 applications' built-in logs record critical events which are used for execution partitioning. A more in-depth study of the 15 complex applications in Section 3.3 shows that the fusion of built-in logs and the audit log even allows precisely tracking causality in complex asynchronous execution models such as worker threads and thread pools. Our study also shows that the design of logging component tends to be stable. For example, *firefox* has been using the same logging facility for 14 years while it has 64 different releases in the duration. More details can be found in Appendix C and Appendix B.

3 SYSTEM DESIGN

In this section, we discuss the design details, including how to normalize various logs to basic relations and how to fuse them

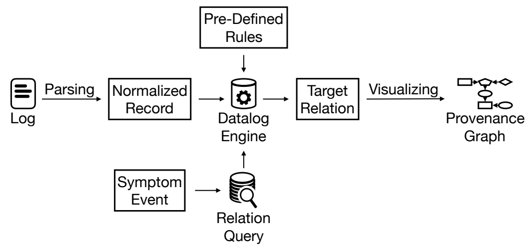


Figure 4: ALchemist's workflow

Figure 5: Examples of raw application logs (left) and the corresponding audit logs (right). ***Firefox*** **saves `main.c` from `a.com` to `/tmp` and opens it with** ***vim***

```
1  15:54:25 [2553] nsHostResolver (TranID=0xfc9c51b0)
2  a.com has 192.168.143.1
3  ...
4  15:54:29 [2553] Socket (TranID=0xfc9acb80) request
5  [TabID = 0, uri=a.com/main.c, referrer=a.com]
6  http request [
7     GET /main.c
8     Host: a.com
9     ...
10    Referrer: a.com
11    Connection: keep-alive ]
12 ...
13 15:54:48 [2553] mozStorage (TranID=0xfd9b3380)
14 INSERT INTO moz_annos(attribute_id, content) VALUES
15 ('FileURI', 'file:/tmp/mozilla/main.c')
```

```
16 type=SOCKADDR 15:54:25 host:127.0.1.1 serv:53
17 type=SYSCALL 15:54:25 syscall=connect exit=0 ppid=2275
18 pid=2553 exe=firefox
19 ...
20 SOCKADDR 15:54:29 host:192.168.143.1 serv:80
21 SYSCALL 15:54:29 syscall=connect exit=0 a0=23 ppid=2275
22 pid=2553 exe=firefox
23 ...
24 PATH 15:54:48 name=/tmp/mozilla/main.c
25 SYSCALL 15:54:48 syscall=open exit=34 ppid=2275 pid=2553
26 exe=firefox
27 ...
28 type=EXECVE a0=vim a1=/tmp/mozilla/main.c
29 type=SYSCALL 15:59:57 syscall=execve exit=0 ppid=2553
30 pid=2842 exe=firefox
```

Table 1: Normalized audit records (top) and ***firefox*** **records (bottom),** A_1 **denotes** ***firefox*** **and** A_2 ***thunderbird***, $I\!P_0$ **denotes 127.0.0.1,** $I\!P_1$ **192.168.143.1**

Index	Time	PID	PPID	PNAME	IP	Port	File	ActionL	ActionRet
S1	15:54:25	2553	2275	A_1	$I\!P_0$	53	-	connect	0
S2	15:54:29	2553	2275	A_1	$I\!P_1$	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

Index	Time	PID	PNAME	IP	File	ActionH	TabID	TranID	URL
F1	15:54:25	2553	A_1	$I\!P_1$	-	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	-

by performing inference and deriving new relations. At the end, we particularly discuss how invisible asynchronous background behaviors can be properly handled.

3.1 Log Normalization

Our overarching design is to fuse all the available application logs and the audit log (at the system level). A prominent challenge lies in understanding the various log formats and deriving the corresponding parsers. The parsers parse these heterogeneous logs to their canonical formats which provide the foundation for the later log fusion step. In the following, we show sample logs and then explain how they can be normalized.

Application Log and the Corresponding Audit Log For Sample Operations. Fig. 5 shows some (simplified) sample application logs and the corresponding audit logs. Specifically, it shows the log for *firefox* downloading a C file and then invoking *vim* to edit it. The application logs are on the left and the corresponding audit logs are on the right.

In Fig. 5, 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 4-11). Next, *firefox* saves the file to `/tmp/mozilla/main.c` (lines 13-15). In contrast, at the low level, we see the socket connections to a local port `127.0.1.1:53` for name resolution (lines 16-18) and then to `192.168.143.1:80` for file download (lines 20-22). As audit log does not have semantic information, it is difficult to know that the network connection at lines 20-22 is for sending the HTTP request and receiving `main.c`. On the other hand, the audit log discloses *firefox* opens *vim* (lines 28-30), which is invisible in 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, file name and similar timestamps.

Normalizing Logs. From the previous examples, we can observe that application logs have their own format and semantics, which are quite different from the audit log and from each other. Hence, as the first step, we develop a set of parsers that transform these logs to their canonical format. Each type of log has its own canonical format, which can be intuitively considered as a database schema. Fields across different logs with the same semantics must have the same field name to facilitate log fusion (e.g., IP, PID, and timestamp). Table 1 shows the canonical representation of audit log entries and appliation log entries, each consisting of 10 fields. Most fields are self-explaining. For audit log entries, `Index` field is a global ID; `PNAME` is the process name; `ActionL` represents the type of the syscall and `ActionRet` the return value. For application log entries, observe that it share 5 common fields with audit log entries. The other fields are application specific. For instance, a canonical *firefox* log entry contains `TabID`, `TranID` and `URL` that are specific to *firefox*, representing tab id, transaction id, and resource URL, respectively[1]. *Firefox* transactions are introduced by developers to denote an atomic operation within a thread (e.g., sending a network request). More detailed discussion can be found later in the section.

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. The most complex parser is that for *firefox*, containing 1800+ lines of code.

3.2 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 causality 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 [9]. Intuitively, one can consider these rules are performing relational operations (e.g., `select` and `join`) on various kinds of logs, which can be considered as tables of different schemas. The inference starts from the *basic relations*, which are the normalized logs, proceeds iteratively until no new relations can be derived.

[1] In ALchemist, we mainly focus on logging the networking module (e.g. nsHttp) and the storage module (e.g. mozStorage) for *firefox*.

Types:

$FirefoxEvent \quad HR$	$:=$	$\langle Time, IDX, PID, PPID, PNAME, IP, Port, File, TabID, TranID, ActionH,$ $ActionRet, URI:$ uniform resource identifier, $NetworkInfo:$ detailed info of network request, $StorageInfo:$ detailed info of local file$\rangle$
$AuditEvent \quad LR$	$:=$	$\langle Time, IDX, PID, PPID, PNAME, IP, Port, File, FileID, SysNum:$ syscall number, $ActionL:$ syscall name, $ActionRet\rangle$
$ActionH$	$:=$	`switch` | `request` | `init` | `end` | `readSegment` | `writeSegment` | $\cdots$
$ActionL$	$:=$	`open` | `close` | `socket` | `connect` | `read` | `write` | $\cdots$
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
/* relation shorthands for firefox log */		
(A7) $switchTo(HR, TabID)$	:	at HR, firefox switches to tab $TabID$ (i.e., $HR.ActionH$ = `switch`)
(A8) $initTran(HR, TranID_1, TranID_2)$	:	HR with $TranID_1$ starts a transaction with $TranID_2$
(A9) $request(HR, TabID, TranID, URI)$	:	In tab $TabID$, HR with $TranID$ requests URI
(A10) $readNtwkData(HR, TranID, NetworkInfo)$	:	HR with $TranID$ reads network data denoted by $Networkinfo$
(A11) $writeNtwkData(HR, TranID, NetworkInfo)$	:	HR with $TranID$ writes network data denoted by $NetworkInfo$
(A12) $createFile(HR, TranID, File)$	:	HR with $TranID$ creates $File$
(A13) $readLocData(HR, TranID, StorageInfo)$	:	HR with $TranID$ reads from a file/data-segment denoted by $StorageInfo$
(A14) $writeLocData(HR, TranID, StorageInfo)$	:	HR with $TranID$ writes to a file/data-segment denoted by $StorageInfo$
/* relation shorthands for audit log */		
(A15) $connect(LR, IP, SocketID)$	:	at LR, the system connects to IP through socket $SocketID$.
(A16) $open(LR, File, FileID)$	:	at LR, the system opens $File$, which has a $FileID$.

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 records 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 firefox records with the same tab id belong to the same tab */
(R6) $sameTab(HR_1, HR_2)$:- $HR_1.TabID = HR_2.TabID$
/* two firefox records with the same transaction id belong to the same tab
(R7) $sameTab(HR_1, HR_2)$:- $sameTran(HR_1, HR_2)$
/* two firefox records with different transaction id belong to the same tab if the first transaction intializes the second one
(R8) $sameTab(HR_1, HR_2)$:- $initTran(HR_1, TranID_1, HR_2.TranID_2)$
/* two high level records belong to the same unit if they belong to the same tab */
(R9) $sameUnitH(HR_1, HR_2)$:- $sameTab(HR_1, HR_2)$
/* two low level records 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 $StorageInfo$) updated by HR_1 */
(R12) $depH(HR_1, HR_2)$:- $writeLocData(HR_1, TranID_1, StorageInfo)\ \&\ readLocData(HR_2, TranID_2, StorageInfo)\ \&\ HR_1.File = HR_2.File\ \&\ HR_1.IDX < HR_2.IDX$
/* HR_2 reads from the network resource (denoted by $NetworkID$) updated by HR_1*/
(R13) $depH(HR_1, HR_2)$:- $writeNtwkData(HR_1, TranID_1, NetworkInfo)\ \&\ readNtwkData(HR_2, TranID_2, NetworkInfo)\ \&\ HR_1.ip = HR_2.ip\ \&\ HR_1.port = HR_2.port\ \&\ HR_1.IDX < HR_2.IDX$
/* two low level records have dependence as long as the corresponding high level records have dependence */
(R14) $depL(LR_1, LR_2)$:- $depH(HR_1, HR_2)\ \&\ project(HR_1, LR_1)\ \&\ project(HR_2, LR_2)$
/* in the same unit, a record of the output type depends on preceding records of the input type, regardless of the resources they access */
(R15) $depL(LR_1, LR_2)$:- $sameUnitL(LR_1, LR_2)\ \&\ LR_1.IDX < LR_2.IDX\ \&\ inputType(LR_1.ActionL)\ \&\ outputType(LR_2.ActionL)$
/* for read/write of regular file, a file read depends on preceding writes to the same file regardless of their units */
(R16) $depL(LR_1, LR_2)$:- $LR_1.File = LR_2.File\ \&\ LR_1.IDX < LR_2.IDX\ \&\ outputType(LR_1.ActionL)\ \&\ inputType(LR_2.ActionL)$

Figure 6: Log fusion rules

In the following, we use the inference and fusion of *firefox* log (the most complex application log) with audit log as an example to illustrate the procedure.

Firefox Execution Model and Its Logging Component. Before we explain the detailed fusion procedure, we have to discuss the unique execution model of *firefox* and its logging component. *Firefox* is a very complex browser, containing numerous functional components for network operations, page parsing/rendering, comprehensive style support, a uniform database resource management system, advanced virtual environment for JavaScript (JS), and a complicated security system. For the sake of cost-effectiveness, its

Figure 7: *Firefox* Asynchronous Download

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

Figure 8: Log for *Firefox* Asynchronous Download

execution is highly asynchronous. In particular, individual functionalities are provided by highly specialized worker threads. For example, there are different threads for socket creation and network communication, file I/O, page rendering, and JS execution, respectively. The main thread serves as a coordinator, distributing sub-tasks to individual worker threads. Each worker thread may serve multiple pages/tabs. Such an asynchronous execution model creates lots of difficulties for the low-level audit logging system [64] as syscalls serving different pages, tabs, JS blobs, and other background tasks are interleaving, without any hints about their origins. *Firefox* uses the NSPR logging module [5] 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.

Example. Fig. 7 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 resulted syscall interleaving makes causality inference difficult. We will use this example to demonstrate how log fusion allows dis-entangling the complex interleaving.

Fusing Firefox Log and Audit Log. The log fusion procedure is formally described in the Datalog language [9], which is a Prolog-like representation for relation computation. The inference rules on these relations are shown in Fig. 6. We use form $p(x_1, x_2, \cdots, x_n)$ to represent relations, 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. At the beginning of Fig 6, we first define a number of types. Specifically, a *firefox* event *HR* is a relation of 15 fields and an audit event *LR* is a relation of 12 fields. They correspond to normalized log entries (some simplified examples can be found in Table 1). We also define *ActionH* and *ActionL* as the type of event of *firefox* and audit, respectively. For example, switch means switching to a tab, init and end denote transaction initialization and termination.

In the middle of Fig. 6, we define a number of basic relations called *atoms*. These relations are directly acquired from the normalized log entries without inference. For example, (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 *firefox* and audit log entries. For example, (A7) $swithTo(HR,\ TabID)$ denotes that a *firefox* event *HR* is essentially a switch to a new tab denoted by $TabID$, and (A15) $connect(LR,\ IP,\ SocketID)$ denotes that an audit log entry *LR* is a socket connection request. These short-hands are used in defining inference rules later in the section. Note that we only present a subset of the atoms that are sufficient to explain our example and technique. We have 258 atoms in total.

After the atoms, we define a set of inference rules that derive additional relations from atoms and fuse *firefox* 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 $B_1, B_2, \cdots, B_n$ 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. An execution unit is similar to that in MPI [64], denoting an autonomous sub-task (e.g., a tab in *firefox*). According to [64], 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 having dependences on all the preceding input events *within the same unit*, even when they operate on different system resources. Various studies [49, 64, 90] have demonstrated that such a strategy yields high precision. In general, units are application specific, determined by the aforementioned four inference rules for that application.

The first two rules (R1) and (R2) of $correlated(HR, LR)$ correlate a *firefox* event and an audit event through their shared fields, indicating that they operate on the same resources. 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* and 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 relations are captured by the aforementioned atom (A2) *atomicL*.

The next rule (R5) $sameTran(HR_1, HR_2)$ is specific to *firefox*. It identifies all the *HR*'s that belong to the same transaction. The next three rules (R6)-(R8) $sameTab(HR_1, HR_2)$ infer the *firefox* events that belong to the same tab. Note that a tab may have many transactions, such as requesting a page, downloading a file, and executing a piece of JS code. In the first rule (R6), there are some *firefox* log entries that contain explicit tab information. 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 (R9) $sameUnitH(HR_1, HR_2)$ derives *firefox* events that belong to the same unit. Here the rule reflects that we treat a tab as a unit. Note that while $sameTab$ is a relation specific to *firefox*, $sameUnitH$ is generic for all applications. For example, we have rules that derive $sameUnitH$ from *thunderbird* based on individual emails. 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 discuss in Section 3.3, this rule is particularly important for proper attribution of background activities.

The last five rules (R12)-(R16) derive causality/dependencies between events. The first *depH* rule (R12) specifies that read and write on the same storage entry induces dependence. This rule allows us to have fine-grained dependencies. For example, *thunderbird* stores all emails in the same INBOX file. Without the storage entry information, any email read is an INBOX file read and hence has to be considered dependent on any preceding writes/updates (on other emails). The second *depH* rule (R13) specifies dependence caused by network read and write. Note that applications may use sockets to perform local I/O. The first *depL* rule (R14) inherits dependence from high level log entries. The second *depL* rule (R15) 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 (R16) derives cross-unit and even cross-process dependence by the common resource that is operated on.

Example Continued. Fig. 8 shows the runtime information of the example execution in Fig. 7 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 sends a request for the main page of `CNN.com` (line 2) in transaction #0 in tab A, then the `Main` thread switches to tab B. In the background, the `Resolver` thread resolves the host of the JS file, `a.com`, in transaction #1, and then initializes a child transaction #2 that will download the JS file (line 8). 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). 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.

In the following, we show how the two logs are fused to derive unit structure and causality. We use F_t and A_t to denote the *firefox* event and the audit event at line t in Fig. 8.

According to rule (R5) in Fig. 6, we can derive $sameTran(F_5, F_8)$ and $sameTran(F_{13}, F_{14})$. By rule (R7), these pairs are in the same tab. By rule (R8), we have $sameTab(F_8, F_9)$ and $sameTab(F_{14}, F_{15})$.

```
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)

Figure 9: (a) Source code and log for *vim 7.3* (b) Source code and log for *sshd 7.4*

Table 2: Execution Models

	Application	I. Sequential	II. Fork Process	III. Task Queue	VI. Thread Pool	V. Virtual Environment
UI Program	firefox			✓	✓	✓
	thunderbird			✓	✓	✓
	chromium			✓	✓	✓
	libreoffice			✓	✓	
	openoffice			✓	✓	
	vim	✓				
	bash		✓			
	foxit			✓	✓	
	mplayer				✓	
Server	nginx				✓	✓
	apache				✓	✓
	vsftpd		✓			
	pure-ftpd		✓			
	sshd		✓			
	tightvnc		✓			

By rule (R6), we have $sameTab(F_2, F_9)$ due to the same tab id A. At the end of inference, we can determine that all the plain *firefox* log entries in Fig. 8 belong to the same tab and hence the same unit, by rule (R9). The shaded entries belong to another unit.

Following rules (R1) and (R3), we have $project(F_2, A_2)$. Note that although F_2 has an URI "CNN.com", 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 (R15), 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. □

3.3 Execution Partitioning by Log Fusion

In the previous section, we have demonstrated how the complex asynchronous execution model of *firefox* can be properly handled by ALchemist. In this section, we study the execution models for the set of popular applications considered, including *firefox*, *Chromium*, *LibreOffice*, *OpenOffice*, and *Apache*, especially their background (asynchronous) activities, and demonstrate that the same partitioning scheme is equally effective. The applications and their execution models are shown in Fig. 2. These models can be divided into five different categories, with each application leveraging 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 Fig. 9 (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. 9 (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 causality of individual tasks 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. Fig. 2 shows that there are quite a number of applications in this class. For these applications, a unit consists of a chain of causally related processes.

Class III: Asynchronous Task Queue. As shown in Fig. 2, 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. The log fusion rules and example in Section 3.2 have demonstrated that how ALchemist can correctly handle the execution model.

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, the listener thread first acquires a pointer to the thread pool . Then it listens to any requests through a while loop . Each time a request is received, it finds an idle thread from the pool or wait when such threads are not available. The request is then served by the

Table 3: *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	`GET(·,·,·)`*	`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)`

* `GET(172.16.163.1, payload.php, 200)`

```
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)

Figure 10: (c) Request processing in `apache 2.4.20`; (d) Source code `firefox 60.0` DOM thread and its app log

thread. After handling a request, it logs the request . Note that although the unit structure is clear in the execution model, we cannot simply consider all behaviors in a worker thread belong to the same unit as worker threads are being recycled. ALchemist leverages the application log events to support correct partitioning of the audit events of a worker thread. Next, we use an example to illustrate. In Table 3, 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 the same worker thread. 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 worker thread (due to thread reuse). 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. Note that the tasks within a worker thread are processed sequentially.

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 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. ALchemist 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. 10), 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 by rule (R5). Other event handling has a similar mechanism.

3.4 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 4, 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)-(R16) in Fig. 6. 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. 6). 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 5. After the causal 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 4 shows that such a demand-driven strategy substantially reduces the workload for the Datalog engine.

4 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, how does it compare to the state-of-the-art instrumentation based techniques MPI [64] and BEEP [49] (Section 4.2)?

RQ2 What is the performance of Datalog module when analyzing real world attacks (Section 4.3)?

RQ3 How effective is our execution partitioning scheme based on log fusion (Section 4.4)?

RQ4 How effective is ALchemist when analyzing real world attacks? How does it compare to NoDoze [30] that does not require instrumentation (Section 4.5)?

4.1 Experiment Setup

To evaluate the efficiency of ALchemist (RQ1), we collect 12 popular applications from the literature [49, 64]. We have also acquired the implementations of BEEP and MPI from their authors for comparison. 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 [1] and the 4 real world attacks in NoDoze [30].

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 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.

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 are shown in Table 4. 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.

Table 4: 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 4)
11	Ubuntu 14.04	0d19h	Apache-2.4.7	ShellShock	NoDoze[30]
12	Ubuntu 14.04	1d20h	OpenSSH-6.6	passwd-gzip-scp	High Fidelity[89]
13	Ubuntu 14.044	1d18h	Apache-2.4.7	Cheating Student	ProTracer[65]
14	Ubuntu 14.04	0d23h	OpenSSH-6.6	Data Theft	PrioTracker[54]

4.2 System Overhead

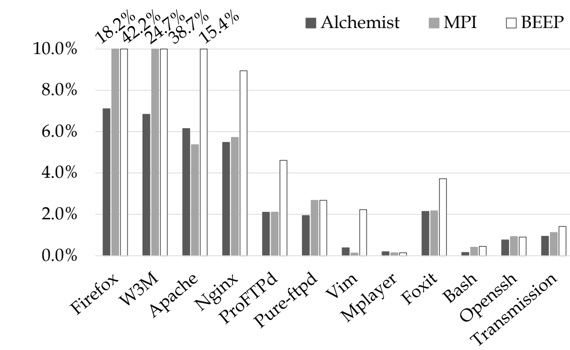


Figure 11: Space overhead

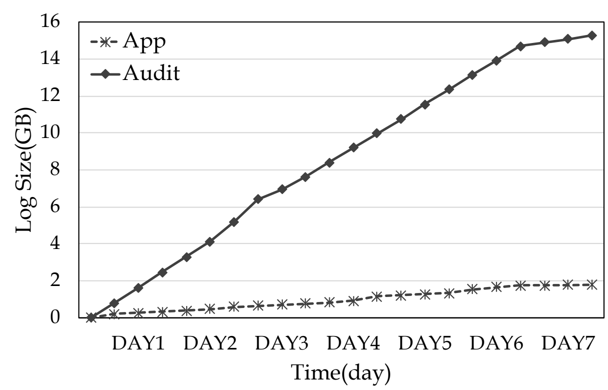


Figure 12: Space consumption over a week

Space Overhead. To measure space overhead, we used the logs from the one-week experiments on the 8 systems. We have turned on ALchemist, MPI and BEEP. The results are shown in Figure 11. 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. 12 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* [6] to simulate *apache* workload and a UI input simulation tool `xdotool` [10] to scriptize keyboard and

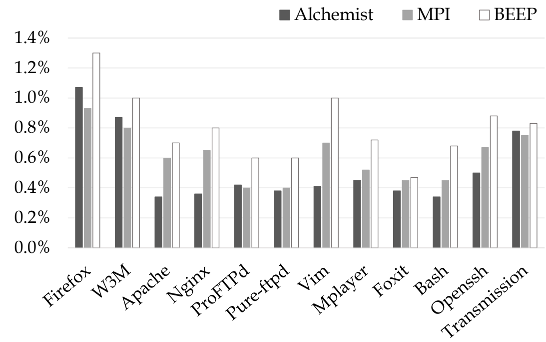


Figure 13: Runtime overhead

mouse activities. The results are shown in Fig. 13. 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.

4.3 Datalog Inference Overhead

Table 5: 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

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 5 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 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.

Table 6: Execution partitioning on asynchronous workloads

Program	Audit Size	App Size	Tuples	Rules	Relations	Time(s)	Mem(MB)	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%
Apache	282.4MB	15.3MB	529K	4.8M	1.4M	10.5	119	100.0%	100.0%
Nginx	205.2MB	11.2MB	401K	2.1M	512K	5.0	92	100.0%	100.0%
Pure-ftpd	388.1MB	6.5MB	832K	5.9M	1.5M	12.5	168	100.0%	100.0%
Vsftpd	491.1MB	9.2MB	1.1M	12.6M	2.5M	21.0	191	100.0%	100.0%
Proftpd	338.5MB	4.7MB	717K	7.4M	1.6M	14.9	130	100.0%	100.0%
TightVNC	402.4MB	7.9MB	839K	6.2M	2.2M	13.7	176	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%
Transmission	1.2GB	17.6MB	2.6M	20.2M	7.1M	42.0	358	98.9%	97.6%

4.4 Effectiveness in Execution Partitioning

We conduct a controlled experiment to evaluate the effectiveness of log fusion in 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. `CNN.com`) and then host these sites locally. Each time, we open one site in a tab using command line, e.g., ”`firefox 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. The second row in Table 6 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. Observe that the precision is 99.7%, with only 16.9k events misattributed. 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 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 files names in the re-execution are different from those in the ground truth. The other applications have similar results. Observe that many of them have 100% precision and recall, denoting perfect partitioning. The details of their workloads can be found in Appendix A. These workloads are posted on [3] for reproduction.

Table 7: Forensic results (L stands for audit level and H stands for application level)

Attack No.	Ground Truth (#units) Attack	Normal	Partition Result (# units) TP	TN	FP	FN	Ground Truth (#records) Attack(L/H)	Normal(L/H)	ALchemist Result (L/H) FP	FN	Precision	Recall	NoDoze Forensic Result (L only) FP	FN	Precision	Recall
1	2	24	2	24	0	0	1043/10	290818/618	25/0	0/0	97.6%/100%	100%/100%	12	0	98.9%	100.0%
2	3	17	3	17	0	0	65/13	95024/212	5/0	0/0	92.8%/100%	100%/100%	3	0	95.6%	100.0%
3	3	115	3	115	0	0	2666/984	1763648/25719	228/50	0/0	92.1%/95.2%	100%/100%	0	263	100.0%	90.1%
4	8	86	8	86	0	0	866/236	799497/14039	107/22	0/0	89.0%/91.5%	100%/100%	105	58	88.5%	93.3%
5	12	288	12	288	0	0	1628/526	3062833/29633	132/39	0/0	92.5%/93.1%	100%/100%	57	21	96.6%	98.7%
6	5	45	5	45	0	0	114/40	181676/1463	12/0	0/0	90.4%/100%	100%/100%	18	0	86.4%	100.0%
7	5	74	5	74	0	0	112/37	155251/680	17/0	0/0	86.8%/100%	100%/100%	13	24	87.1%	78.6%
8	7	67	7	67	0	0	307/26	697515/297	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	154752/5349	5/0	0/0	93.5%/100%	100%/100%	4	0	94.8%	100.0%
10	7	127	7	127	0	0	726/183	730103/14496	82/23	0/0	89.9%/88.8%	100%/100%	69	32	90.9%	95.6%
11	11	541	11	541	0	0	285/35	1365100/33517	7/0	0/0	97.6%100%	100% 100%	4	2	98.6%	99.3%
12	2	12	2	12	0	0	211/9	1222889/21	13/0	0/0	94.2%/100%	100%/100%	21	0	90.9%	100%
13	5	43	5	43	0	0	101/20	1051610/44	4/0	0/0	96.2%/100%	100%/100%	2	2	98.1%	98.1%
14	6	12	6	12	0	0	656/17	511480/47	0/0	0/0	100%/100%	100%/100%	0	0	100%	100%
Avg.	6	123	6	123	0	0	630/154	863014/9009	47/9	2/0	93.1%/94.5%	99.6%/100%	24	32	96.3%	95.2%

4.5 Effectiveness in Attack Forensics

To answer RQ4, we used ALchemist and the technique described in NoDoze [30] to generate provenance graphs for the 14 APT attacks. The graphs by MPI are very similar to ours and hence the comparison is elided. Recall MPI requires code annotations and instrumentation. 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. The results are summarized in Table 7. The table contains the following information: columns 2∼3 for ground truth number of attack units and normal units (note that attack steps may interleave with or involve benign activities); columns 4∼7 for the number of attack units and normal units predicted by ALchemist, FP and FN; columns 8∼9 for the ground truth number of attack records and normal records, each including both the audit record number and the corresponding application record number; columns 10∼13 the number of FPs and FNs by ALchemist, precision, and recall; columns 14∼17 the FPs and FNs (at the audit log level) by NoDoze, precision and recall. 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 93.1% precision and 99.6% recall for audit records, whereas NoDoze can achieve 96.3% precision and 95.2% recall. The reason for the lower accuracy is that ALchemist does not leverage execution profile to eliminate common cases and most TC engagement attacks and the attacks from NoDoze indeed induce rarely happened dependencies. For example, there are some benign subjects and objects involved in attack paths such as "sudo" and application preference files. ALchemist added them to the causal graphs. In some sense, ALchemist and NoDoze are complementary such that execution profile in NoDoze can be leveraged to improve our results. We want to point out that after grouping the events into units, our graphs are fairly small, the precision difference (with NoDoze) at the low level does not cause practical quality loss. In contrast, our graphs are easy to interpret due to their inclusion of high level semantics.

ALchemist has FNs in only one case whereas NoDoze misses important events in 8 out of 14 attacks. In attack 8, the adversary exploits daemon services to achieve privilege escalation. Although the relation between *sshd* and payload is detectable, how the payload exploits *sshd* and injects code are invisible to ALchemist. In contrast, NoDoze misses 10%∼20% malicious events for complex attacks such as the Azazel injection (attack 7). This is because the attacks leveraged frequently executed apps and dependencies. These events are critical in understanding attack provenance.

5 CASE STUDIES

5.1 Attack #4: Exfiltration

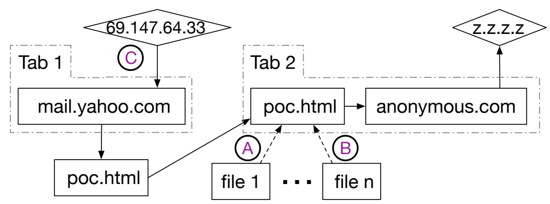


Figure 14: Causal graph of attack #4 by ALchemist

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* [11] by *firefox*, `poc.html` has the access to all files in a folder if one of its DOM objects has access to these files. Then attacker uses ClickJacking [8] 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. 14 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 C and the exfiltration of files (e.g., A and B). 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 `anony- mous.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.

5.2 Attack #7: Azazel Attack

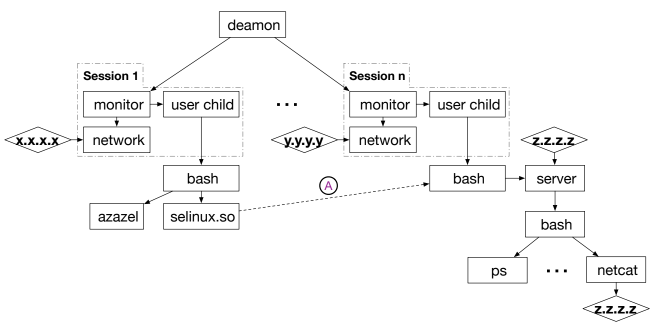


Figure 15: Causal graph for the Azazel attack by ALchemist

Attack Scenario. The attacker connects to the 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.

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. 15. 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 A 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.

6 RELATED WORK

Log analysis. Log analysis is an important research area that has been studies for decades due to its practical importance. Since logs record the events and states that occur in a system or other software runs, many works focus on leveraging information extracted from logs for anomaly detection, problem diagnosis, runtime verification and performance modeling [16, 20, 21, 25, 27, 32–34, 36, 44, 52, 53, 55, 57–59, 68, 70, 72, 74, 77, 81–83, 88, 91, 93, 96].

Datalog-based analysis. Datalog is a declarative query language which adds logical inference to relational queries. Because of the feature of inferring new facts based on the pre-defined rules, it has been widely used in program analysis [14, 18, 24, 38, 45, 46, 48, 56, 69, 73, 76, 78–80, 84, 84, 94], including points-to analysis [14, 18], alias analysis [84], side channel mitigation [73], etc.

Data provenance and casuality analysis. Data provenance [29, 66, 75] provides a historical record of the data and its origins. Audit logging [15, 26, 37, 41, 67, 71, 75] is a special type of data provenance which provides a chronological record of security-relevant events. From it, one can trace back to identify the root cause of a attack-related event. Many causality analysis techniques [35, 41–43, 47, 49, 63–65, 98] have been proposed to improve attack investigation process. Some of them suffer from the *dependency explosion problem* [42, 43]. Some require instrumentation [49, 64, 65], which is not practical for deployment in enterprises. Many techniques utilize learning/profiling to derive a reference model to detect abnormal events [23, 30, 31, 35, 53, 54, 68, 70, 86–89]. 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.

7 CONCLUSIONS

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 causal graphs, without requiring any instrumentation. It also outperforms state-of-the-art techniques with and without instrumentation.

REFERENCES

[1] 2018. darpa-i2o/Transparent-Computing: Material from the DARPA Transparent Computing Program. https://github.com/darpa-i2o/Transparent-Computing.

[2] 2019. 50 Essential Linux Applications. https://tinyurl.com/wcj5og2.
[3] 2019. ALchemist2020/Workload. https://github.com/ALchemist2020/Workload.
[4] 2019. Kernel Path Protection. https://tinyurl.com/7ttl5h2.
[5] 2019. NSPR LOG MODULES. https://tinyurl.com/gmlmtnz.
[6] 2020. Apache HTTP server benchmarking tool. https://tinyurl.com/onkcat3.
[7] 2020. APT Groups. https://tinyurl.com/y2tqt74o.
[8] 2020. ClickJacking. https://tinyurl.com/62thhvp.
[9] 2020. Datalog. https://tinyurl.com/cam64up.
[10] 2020. jordansissel/xdotool: fake keyboard/mouse input, window management, and more. https://github.com/jordansissel/xdotool.
[11] 2020. Same-origin policy. https://tinyurl.com/pp86n9a.
[12] 2020. Sysdig. https://tinyurl.com/nzrexz5.
[13] Sepideh Avizheh, Tam Thanh Doan, Xi Liu, and Reihaneh Safavi-Naini. 2017. A Secure Event Logging System for Smart Homes. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*. ACM, 37–42.
[14] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *International Static Analysis Symposium*. Springer, 84–104.
[15] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. 2015. Trustworthy whole-system provenance for the Linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*. 319–334.
[16] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*. 468–479.
[17] Erik-Oliver Blass and Guevara Noubir. 2017. Secure logging with crash tolerance. In *2017 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 1–10.
[18] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262.
[19] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 217–231.
[20] Rui Ding, Qiang Fu, Jian-Guang Lou, Qingwei Lin, Dongmei Zhang, Jiajun Shen, and Tao Xie. 2012. Healing online service systems via mining historical issue repositories. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 318–321.
[21] Rui Ding, Qiang Fu, Jian Guang Lou, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Mining historical issue repositories to heal large-scale online service systems. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 311–322.
[22] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 139–150.
[23] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1285–1298.
[24] Azadeh Farzan and Zachary Kincaid. 2012. Verification of parameterized concurrent programs by modular reasoning about data and control. *ACM SIGPLAN Notices* 47, 1 (2012), 297–308.
[25] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*. IEEE, 149–158.
[26] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. 2005. The taser intrusion recovery system. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 163–176.
[27] Zhongxian Gu, Earl T Barr, Drew Schleck, and Zhendong Su. 2012. Reusing debugging knowledge via trace-based bug search. *ACM SIGPLAN Notices* 47, 10 (2012), 927–942.
[28] Ragib Hasan, Radu Sion, and Marianne Winslett. 2009. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance.. In *FAST*, Vol. 9. 1–14.
[29] Ragib Hasan, Radu Sion, and Marianne Winslett. 2009. Preventing history forgery with secure provenance. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 12.
[30] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage.. In *NDSS*.
[31] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*.
[32] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and detecting log-related issues. *Empirical Software Engineering* 23, 6 (2018), 3248–3280.
[33] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 60–70.
[34] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 207–218.
[35] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott D Stoller, and VN Venkatakrishnan. 2017. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *Proc. USENIX Secur.* 487–504.
[36] Hennie Huijgens, Robert Lamping, Dick Stevens, Hartger Rothengatter, Georgios Gousios, and Daniele Romano. 2017. Strong agile metrics: mining log data to determine predictive power of software metrics for continuous delivery teams. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 866–871.
[37] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2018. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *27th USENIX Security Symposium (USENIX Security 18)*. 1705–1722.
[38] Zhao Jianhua and Li Xuandong. 2013. Scope logic: An extension to hoare logic for pointers and recursive data structures. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 409–426.
[39] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430.
[40] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *Acm Sigplan Notices*, Vol. 47. ACM, 121–132.
[41] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2010. Intrusion recovery using selective re-execution. (2010).
[42] Samuel T King and Peter M Chen. 2003. Backtracking intrusions. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 223–236.
[43] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality.. In *NDSS*.
[44] Thomas Krismayer, Michael Vierhauser, Rick Rabiser, and Paul Grünbacher. 2020. Comparing Constraints Mined From Execution Logs to Understand Software Evolution. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 491–495.
[45] Markus Kusano and Chao Wang. 2016. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 799–809.
[46] Markus Kusano and Chao Wang. 2017. Thread-modular static analysis for relaxed memory models. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 337–348.
[47] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, et al. 2018. 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*.
[48] Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 1–12.
[49] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *NDSS*.
[50] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1005–1016.
[51] Bo Li. 2017. *Enabling fine-grained reconstruction and analysis of web attacks with in-browser recording systems*. Ph.D. Dissertation. uga.
[52] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 583–588.
[53] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 102–111.
[54] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a timely causality analysis for enterprise security. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS). The Internet Society, San Diego, California, USA*.
[55] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2019. Which Variables Should I Log? *IEEE Transactions on Software Engineering* (2019).
[56] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis.. In *USENIX Security Symposium*, Vol. 14.

18–18.

[57] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 557–566.

[58] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining Invariants from Console Logs for System Problem Detection.. In *USENIX Annual Technical Conference*. 1–14.

[59] Kasper S Luckow and Corina S Pasareanu. 2016. Log2model: inferring behavioral models from log data. In *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 25–29.

[60] Di Ma. 2008. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 341–352.

[61] Di Ma and Gene Tsudik. 2007. Forward-secure sequential aggregate authentication. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 86–91.

[62] Di Ma and Gene Tsudik. 2009. A new approach to secure logging. *ACM Transactions on Storage (TOS)* 5, 1 (2009), 2.

[63] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, low cost and instrumentation-free security audit logging for windows. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 401–410.

[64] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. MPI: Multiple perspective attack investigation with semantics aware execution partitioning. In *USENIX Security*.

[65] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting.. In *NDSS*.

[66] Patrick McDaniel. 2011. Data provenance and security. *IEEE Security & Privacy* 9, 2 (2011), 83–85.

[67] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. 2006. Provenance-aware storage systems.. In *USENIX Annual Technical Conference, General Track*. 43–56.

[68] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 26–26.

[69] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319.

[70] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. 2015. 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*. IEEE, 45–56.

[71] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. 2018. Runtime analysis of whole-system provenance. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1601–1616.

[72] Karthik Pattabiraman, Giancinto Paolo Saggese, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2010. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing* 8, 5 (2010), 640–655.

[73] Brandon Paulsen, Chungha Sung, Peter AH Peterson, and Chao Wang. 2019. Debreach: Mitigating Compression Side Channels via Static Analysis and Transformation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 899–911.

[74] Ernest Pobee and WK Chan. 2019. AggrePlay: efficient record and replay of multi-threaded programs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 567–577.

[75] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 259–268.

[76] Ramy Shahin, Marsha Chechik, and Rick Salay. 2019. Lifting datalog-based analyses to software product lines. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 39–49.

[77] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Brain Adams, Ahmed E Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 402–411.

[78] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. 2016. Static DOM event dependency analysis for testing web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 447–459.

[79] Chungha Sung, Markus Kusano, and Chao Wang. 2017. Modular verification of interrupt-driven software. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 206–216.

[80] Chungha Sung, Shuvendu K Lahiri, Constantin Enea, and Chao Wang. 2018. Datalog-based scalable semantic diffing of concurrent programs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 656–666.

[81] Mark D Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2013. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *2013 IEEE international conference on software maintenance*. IEEE, 110–119.

[82] Chakkrit Tantithamthavorn and Arnon Rungsawang. 2012. Knowledge discovery in web traffic log: A case study of Facebook usage in Kasetsart University. In *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 247–252.

[83] Bipin Upadhyaya, Ran Tang, and Ying Zou. 2013. An approach for mining service composition patterns from execution logs. *Journal of Software: Evolution and Process* 25, 8 (2013), 841–870.

[84] John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. 131–144.

[85] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2011. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability* 61, 1 (2011), 149–169.

[86] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell DE Long. 2012. A hybrid approach for efficient provenance storage. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 1752–1756.

[87] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Darrell DE Long, Ahmed Amer, Dan Feng, and Zhipeng Tan. 2011. Compressing Provenance Graphs.. In *TaPP*.

[88] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 117–132.

[89] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 504–516.

[90] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. 2020. UISCOPE: Accurate, Instrumentation-free, Deterministic and Visible Attack Investigation. In *NDSS*.

[91] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. 2006. Automated known problem diagnosis with event traces. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 375–388.

[92] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 293–306.

[93] Tarannum Shaila Zaman, Xue Han, and Tingting Yu. 2019. SCMiner: Localizing System-Level Concurrency Faults from Large System Call Traces. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 515–526.

[94] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 239–248.

[95] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 19–33.

[96] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 683–694.

[97] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 415–425.

[98] Ningning Zhu and Tzi-cker Chiueh. 2003. Design, implementation, and evaluation of repairable file service. In *2003 International Conference on Dependable Systems and Networks (DSN)*. IEEE, 217.

[99] De-Qing Zou, Hao Qin, and Hai Jin. 2016. Uilog: Improving log-based fault diagnosis by log analysis. *Journal of computer science and technology* 31, 5 (2016), 1038–1052.

A ASYNCHRONOUS WORKLOAD FOR OTHER APPLICATIONS IN THE EXECUTION PARTITIONING EXPERIMENT IN SECTION 4.4

Server Applications. For *apache*, we place multiple files on the server (e.g. 1.txt and 2.txt). We send a request from `IP1` to access files 1.txt on the server and collect logs. All the high level *apache* log entries belong to the same request unit and all the corresponding audit logs should be attributed to the unit. We repeat this 10 times with different IP addresses and acquire the ground truth. In this process, we use a network record and replay tool *GoPlay* to record network traffic for each IP. Then we replay the traffic from different IPs at the same time, causing interleaving. Other server applications like *nginx*, *proftpd* and *tightvnc* follow a similar procedure.

Chromium. For *chromium*, we use *HTTrack* to crawl 10 popular websites, e.g., `yahoo.com` and `CNN.com`, and all the content pages, CSS and JS to a local folder and then host these sites locally. Each time, we then use chromium's headless mode to open one site in a tab, and collect the corresponding logs. All the log entries belong to the tab opening `CNN.com`. We do this for the ten sites and obtain the ground truth. Then, we open the 10 sites in parallel, log entries from different tabs hence interleave with each other. We use ALchemist to partition the application log and the audit log to units and attribute individual events.

Vim. For *vim*, we place a same set of files in both the "`workload`" and the "`test`" folders. We then use a *vim* script to simulate user behavior, for instance, "`:w >> 1.out`" to write buffer content to a file `1.out`. We generate a long sequence of random vim script which opens multiple file buffers and reads/writes files in the "`workload`" folder. Then we execute the commands in the script one by one with more than 30 seconds idle time in between. We take a snapshot before and after executing each command. The differences between two consecutive log snapshots should belong to current unit. Then we rerun the vim script in the "`test`" folder. Note that we duplicate the files in two folders as file updates may be persistent.

LibreOffice/OpenOffice. The workload setup and experiment of *LibreOffice/OpenOffice* are similar to that of *vim*. Specifically, we place a same set of files in both the "`workload`" and the "`test`" folders. We use snippets of python code to simulate various kinds of user behaviors such as creating a file and writing to a file, using the APIs provided by *LibreOffice/OpenOffice*. We then generate a long sequence of random user behaviors based on the primitive snippets. Specifically, it opens files in the "`workload`" folder and performs various operations on them (e.g. inserting data into a sheet cell). The files are created/accessed in order, with substantial idle time in between (to avoid interleaving of any background behaviors). As such, the idle durations serve as the unit boundaries (one unit for each file), which allow us to partition the logs and acquire the ground truth. Then we shuffle the operations in the script so that the operations on all the files interleave. We then execute it in the "`test`" folder to avoid interference from the ground truth execution.

Foxit. For *foxit*, we place 10 PDF files to a folder. We then use a command line to open each PDF in a tab, e.g., "`FoxitReader 1.pdf`" to open `1.pdf`, and collect the corresponding logs, which serve as the ground truth. Then we use "`FoxitReader 1.pdf 2.pdf ...`" to open 10 files simultaneously, causing interleaving.

Table 8: 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

Transmission. In the workload of *transmission*, we host 10 large files on a local server and create the corresponding magnet links. We then use a command to add a magnet link (once added, *transmission* will start to download) and collect the logs. For example, we use "`transmission-remote -a link1`" to add and download `link1`. Then, we use "`transmission-remote -a link1 link2 ...`" to add and download 10 magnet links simultaneously, causing interleaving.

B STABILITY STUDY OF APPLICATION BUILT-IN LOGGING MODULES

We study the stability of application built-in logging modules. The results are shown in Table 8. 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.

C STUDY OF TOP 30 LINUX APPLICATION BUILT-IN LOGGING

We study top 30 Linux applications from [2] to check if these applications have built-in logging module and if their logs contain information to indicate unit boundary, which is the most critical information for execution partitioning. Columns 1-2 show the applications and their brief description. Column 3 presents if the application has built-in logging facility. Column 4 presents the execution unit structure for the application. Column 5 shows if the application log contains information to separate different units. The study is done by manually inspecting the applications' source repository. From the table, 28 out of 30 applications are long running and 29 out of 30 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

Table 9: Built-in logging study for top 30 Linux applications (20 UI programs and 10 servers).

	Application	Description	Has Built-in Logging	Unit	Log of Unit Boundary
UI Program	Thunderbird	Email Client	Yes	Conversation Thread	Yes
	Geary	Email Client	Yes	Conversation Thread	Yes
	WizNote	Collaborative Note-taking	Yes	Note	Yes
	Chromium	Web Browser	Yes	Tab	Yes
	Firefox	Web Browser	Yes	Tab	Yes
	FileZilla	FTP Client	Yes	Connection	Yes
	OpenOffice	Office Suite	Yes	File/Window	Yes
	LibreOffice	Office Suite	Yes	File/Window	Yes
	KeePass*	Password Management	No	/	/
	gscan2pdf*	PDF Scanner	Yes	Document	Yes
	WINE	Compatibility Layer	Yes	Guest Application	Yes
	VirtualBox	Virtual Machine	Yes	Guest Environment	Yes
	Skype	Telecommunications	Yes	Chat Thread	Yes
	DropBox	Cloud Storage Client	Yes	Folder	Yes
	Gimp	Graphic Editor	Yes	Window	Yes
	Bash	Shell	Command History	Command	Yes
	Zsh	Shell	Command History	Command	Yes
	Nmap	Network Audit	Yes	Connection	Yes
	Vim	Text Editor	Yes	Buffer/Window	Yes
	Emacs	Text Editor	Yes	Buffer/Window	Yes
Server	Apache	Web Server	Yes	Connection	Yes
	Nginx	Web Server	Yes	Connection	Yes
	Lighttpd	Web Server	Yes	Connection	Yes
	TightVNC	VNC Server	Yes	Connection	Yes
	Openssh	SSH Server	Yes	Connection	Yes
	Pure-ftpd	FTP Server	Yes	Connection	Yes
	Vsftpd	FTP Server	Yes	Connection	Yes
	Proftpd	FTP Server	Yes	Connection	Yes
	FileZilla	FTP Server	Yes	Connection	Yes
	UFW	Firewall	Yes	Connection	Yes

* Not-long running applications.

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.