# High Accuracy Attack Provenance via Binary-based Execution Partition

Kyu Hyung Lee Xiangyu Zhang Dongyan Xu
Department of Computer Science and CERIAS, Purdue University, West Lafayette, IN 47907, USA
{kyuhlee,xyzhang,dxu}@cs.purdue.edu

***Abstract*—An important aspect of cyber attack forensics is to understand the provenance of suspicious events, as it discloses the root cause and ramifications of cyber attacks. Traditionally, this is done by analyzing audit log. However, the presence of long running programs makes a live process receiving a large volume of inputs and produce many outputs and each output may be causally related to all the preceding inputs, leading to dependence explosion and making attack investigations almost infeasible. We observe that a long running execution can be partitioned into individual units by monitoring the execution of the program's event-handling loops, with each iteration corresponding to the processing of an independent input/request. We reverse engineer such loops from application binaries. We also reverse engineer instructions that could cause workflows between units. Detecting such a workflow is critical to disclosing causality between units. We then perform selective logging for unit boundaries and unit dependences. Our experiments show that our technique, called BEEP, has negligible runtime overhead ($< 1.4\%$) and low space overhead (12.28% on average). It is effective in capturing the minimal causal graph for every attack case we have studied, without any dependence explosion.**

## I. Introduction

Acquiring attack provenance is an important capability in cyber attack investigation. For example, given a symptom of an attack (e.g., a malicious process detected), we want to determine the "entry point" of the attack, such as viewing a malicious URL or opening an email attachment, that ultimately leads to the detected symptom. Such capability is helpful to attack attribution and future attack prevention. Upon detecting an attack, it is also important to understand its ramifications, that is, what damage has been caused on the victim system. Both require attack provenance information. However, achieving accuracy in attack provenance has been proved challenging. For example, assume a malware program first gets into the victim system through opening a social-engineered email. It then lurks in the system for weeks performing stealthy actions such as installing key loggers, profiling the victim machine and/or user, and setting up a backdoor. This attack is not detected until one month later when the attacker tries to remotely log in through the backdoor to harvest the collected information. Unfortunately, by this time, the trace-back from the symptom to the entry point will be difficult, even if the corresponding log entries still exist, because a causality analysis may conservatively indicate numerous causal chains – formed by causal relations between log entries – to the backdoor process. Even if email access is identified as the initial step of the attack, it is still difficult to pin-point the culprit email.

Most causal analysis techniques use logging to record important events during system execution and then correlate these events during investigation. The logging of events can be at the network (e.g. messages being sent or received), OS (e.g. system calls), or program (e.g. memory reads and writes) level. However, many existing logging techniques are too coarse-grained by attributing events to individual processes. For example, system call logging-based techniques (e.g., [18], [15]) treat processes as subjects and files, sockets, and other passive entities as objects. A system call will create a causal relation/edge between a subject and an object (e.g., a process reading a file), or between two subjects (e.g., a process spawning a child process.) We argue that, for root cause analysis, the granularity of "process" is too coarse and will suffer from the *dependence explosion problem* [19], [12], [17]: a subject (i.e., process) is causally related to *all the objects* it has accessed so far, making it difficult to identify the *small subset of objects* that are truly relevant to an attack. In Section II we will present such a real case where a web browser process has dependences with numerous external connections while only one of them is malicious.

The dependence explosion problem is mainly caused by the non-trivial lifetime and iterative input/output processing of processes. For example, a server process may accept and respond to a large number of requests during its lifetime, whereas an email client process may send/receive a large number of emails during its lifetime. Theoretically, an output activity of a process may be causally related to all preceding inputs to the same process. Unfortunately, for off-the-shelf applications, especially those that come *without source code*, automatically determining the real causality is highly challenging.

In this paper, we present a new scheme for *efficient, dependence explosion-free logging for binary programs*, called BEEP[1]. BEEP is based on a new, finer-grain type of subjects, called *units*. More specifically, a unit is a segment of execution of a process that processes a specific object (e.g., an email message, URL, or request). And a process is "partitioned" into multiple units. BEEP also involves the identification of a small set of critical memory

[1]It stands for "Binary-based ExEcution Partition".

dependences that denote high-level causality between units. With subjects being units instead of processes, a subject is likely to be causally related to one or a very few objects (processed by that unit), which will significantly mitigate the dependence explosion hence improving the accuracy of log-based attack provenance. Furthermore, the identification of units is performed *automatically without source code and with very small instrumentation overhead.*

BEEP is based on the following key observation: The execution of a wide range of applications (both server-side and client-side) is dominated by a small number of loops (with large number of iterations for each loop) – for example, loops that receive and dispatch input events and loops in worker threads that process individual tasks. Each iteration of such a loop can be considered a semantically autonomous *unit* as it often operates on an individual input object. As such, BEEP works by reverse engineering such loops from a binary program as well as dependences between iterations of such loops.

Our contributions are summarized as follows.

- We propose to reduce the granularity of executing subjects for log-based attack provenance analysis. Instead of having processes as subjects, BEEP partitions the execution of a process into individual units for effective dependence explosion mitigation.
- We perform an empirical study to illustrate that process-based logging leads to dependence explosion, as well as to confirm the existence of units during the execution of many programs.
- We propose novel techniques to reverse engineer critical loops whose iterations denote the natural boundaries for units, and memory access instructions that induce causal workflows between units. Application binaries are then instrumented at these places to log such events.
- We propose a log analysis algorithm that can cohesively reason about both the system log and our log to construct precise causal dependence graph for an attack.
- We have built a prototype that successfully instruments a set of commonly used Linux applications for more accurate attack investigation. We evaluate the performance of these applications and our results show that the runtime overhead is trivial ($< 1.4\%$) and the space overhead is low (12.28% on average). We also conduct three case studies that emulate different real attack scenarios to illustrate the effectiveness of BEEP, under which the causal graphs are on average 40.93 times smaller than those generated by a standard system call-based audit system.

## II. Motivating Example : Backtracking A Trojan Attack

In this section, we present a concrete, realistic attack scenario to motivate BEEP. This attack involves the *pine* email client and the *firefox* web browser. A number of other processes such as *procmail*, *bash*, *sendmail* are also involved. We compare the causal graphs generated by analyzing the traditional system call log and by BEEP.

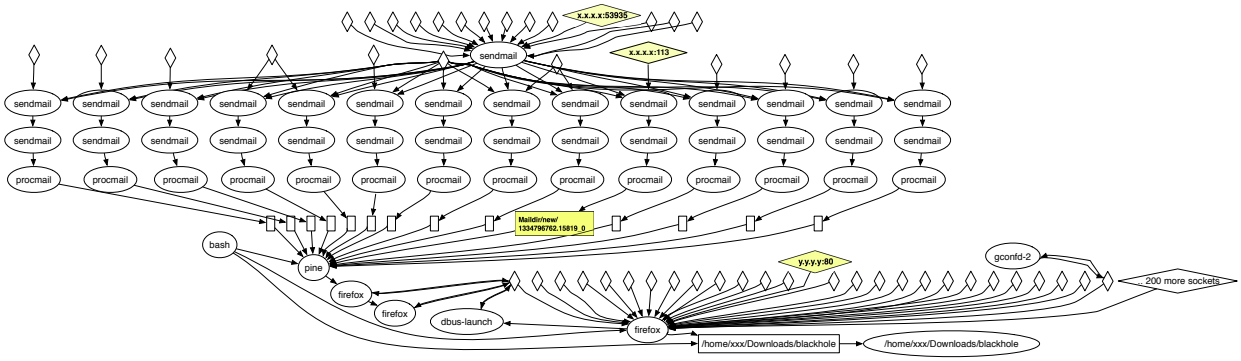
**Attack Scenario:** Suppose a user has the habit of using *pine* to access e-mails and *firefox* to browse the web. He has kept these two processes alive while using his computer. In a duration of 29 minutes, he has visited 11 different web sites and received/viewed 14 emails. Among the set of emails, there is a phishing email that contains a malicious URL. The user has clicked the URL to open a remote malicious page. Consequently, a file is downloaded to his computer and executed. The file is a backdoor trojan named *blackhole* [2]. Upon execution, *blackhole* opens a backdoor port that can be used for denial of service attacks, installing additional trojans and stealing private information.

**Forensic Analysis:** The system administrator of the machine by accident detects that a strange process (i.e. the trojan) is running in the background and wants to backtrack to the entry point of the attack. However, the opening of the culprit email and visiting of the malicious URL happened in the past and hence the current state of the system does not provide any obvious hint on the attack entry point. She then applies one of the existing host attack investigation techniques [18], [12] that detects dependences between processes, files, and sockets by analyzing the audit log that records system calls and signals.

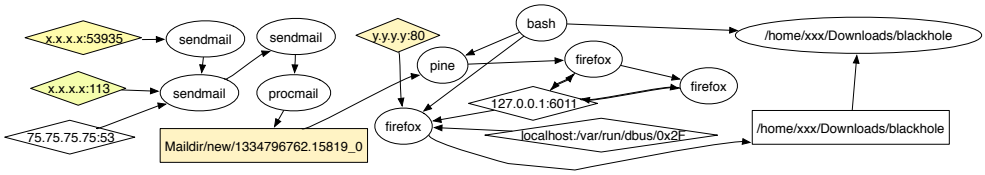
Causal analysis of the audit log results in a causal graph as illustrated in Fig. 1(a), in which ovals, diamonds, and boxes represent processes, sockets, and files, respectively. An edge denotes a system call event that causally correlates two entities and the arrow indicates direction of information flow. For example, the path from process *bash* to process *blackhole* (via file *blackhole*) at the bottom of the figure represents that the *bash* shell opens the binary file of *blackhole* and spawns a process from it. The edge from process *firefox* to the binary file means that the file is created by the browser.

Now the administrator knows that the malicious binary comes from *firefox*. However, she wants more specific information about the *source* of the malicious file. Unfortunately, from the graph, she only observes that *firefox* has received inputs from more than 200 sources as the process has been running for quite some time. Each of the diamonds connected to *firefox* denotes that the browser has visited web pages through a socket. In fact, the graph contains 225 different connections. Any of them could be the source of the malicious file whereas the malware is actually from the link through **y.y.y.y**. Observe that the browser is also causally connected to the *pine* process. When the user clicked the phishing link in the email, *pine* spawned a *firefox* process, which does nothing but forwarding the link to the existing long running *firefox* process.

Assume the administrator is able to track back to the




(a) Graph generated from syscall log. It is a much smaller reduced version. The original graph contains 200 more communication channels connected to *firefox*.




(b) Generated graph by fine-grained forensic analyzer

Figure 1. Back-tracking an attack: diamond, box, and oval represent communication channel (e.g. socket), file and process, respectively.

*pine* process. She again faces the difficulty of identifying *which email* has caused the creation of the *firefox* process. The small boxes connected to the *pine* node denote individual emails, each of which is generated from a path of "*sendmail*→*sendmail* →*sendmail* →*procmail*", meaning that initially the *sendmail* daemon receives an incoming mail request, and then it (transitively) spawns two *sendmail* processes and a *procmail* process to handle the request and deliver the message. There are many such paths, among which there is only one for the malicious email.

The second row of Table I shows the graph statistics.

**Existing Heuristics:** Other researchers have also observed this problem. In existing efforts [18], [12], [19], simple heuristics have been proposed to mitigate the problem, such as using timestamps to approximate causality and using a white list to preclude unnecessary dependences. However, the asynchronous design of long running programs, parallel tasks, and delay in user interactions render timestamps ineffective in many cases. Defining a white list often demands judicious human efforts and may lead to false negatives.

For example, to detect which connection in *firefox* causes the creation of the malicious file, one could detect the socket read/writes events that are close to the file write event along the time dimension. Unfortunately, *firefox* is a multi-threaded program that allows opening multiple tabs to show different web pages at the same time. The reads and writes from/to sockets in different threads interleave. In particular, the closest connection was a connection to *Google*, not the malicious connection. Furthermore, file writes are conducted in an asynchronous way in *firefox* to improve responsiveness. Specifically, when a thread (corresponding to a tab) wants to perform a file write, it posts this request to a work queue. The request is later dequeued and processed by an idle thread specialized in file writing. In other words, there may be substantial delay between the event of viewing the page and the file write. In fact, there are activities in 21 different non-

| | process # | file # | com. channel # | edge # |
|---|---|---|---|---|
| Linux audit | 51 | 15 | 251 | 354 |
| File Offset [26] | 51 | 15 | 251 | 354 |
| Socket Interval [12] | 50 | 14 | 39 | 145 |
| BEEP | 10 | 2 | 6 | 23 |

Table I
COMPARISON OF CAUSAL GRAPHS VIA FOUR APPROACHES.

malicious connections between these two events in our case study.

Another possible heuristic is to treat a *thread* as an autonomous subject. However, in *firefox*, unless the workflow between the page viewing thread and the file write thread is tracked, the connection between the file and the malicious URL cannot be properly established. Also, large programs such as *firefox* and *apache* make heavy use of thread pools such that events in the same thread may not be causally related as a thread may be used/reused to process multiple independent input requests.

In [26], file offsets are logged additionally to distinguish processes that operate on different segments of a file. Only the processes that operate on the same file segment are considered causally related. In our example, such a technique is not able to reduce any false dependencies because partial file accesses did not happen. For example, when the user reads an email message, *pine* always reads the entire file. *Firefox* and *bash* also access the *blackhole* file as a whole.

The third row of Table I shows the statistics of the file offset approach.

TASER [12] separates an execution to segments bounded by two consecutive socket reads to reduce false dependences. Subjects/objects falling into different segments are not considered as causally related. However, it does not work for programs that are not network oriented (e.g. an editor). Furthermore, a segment in their notion is not semantically autonomous and hence the technique may have false negatives. In the *firefox* case, the technique misses the causality between the file write and the socket read as they are performed in different threads. The fourth row of Table I shows the statistics of this approach. It reduces more than 200 dependencies, mainly from *firefox*. However, the resulting causal graph is still quite large because it is not able to achieve reduction from *pine* as *pine* reads from email files instead of sockets. As a result, all emails are considered causally related and hence all the corresponding socket reads by the *sendmail* processes. It also undesirably removes key dependencies such that the root cause email is not even reachable from the symptom.

**Overview of Our Approach:** We propose BEEP to facilitate more accurate causality analysis with a finer-grain subject: the unit. BEEP dynamically partitions the execution of a process into autonomous execution segments called units. A unit is essentially an iteration of an event processing loop. For a program driven by external events, its execution is dominated by such iterations and the functions directly/indirectly invoked by the iterations. On the other hand, units are not completely independent. So we detect causality between units by observing *memory dependences.* For example, in *firefox*, the processing of an event, such as a connection opening event, page loading event, or page rendering event, is carried out by an iteration of a loop. An iteration (i.e., unit) may further post more continuation tasks in a work queue which are handled by other loop iterations (i.e., units.)

To help identifying units, BEEP first *automatically* detects the event processing loops in the *firefox* binary and the instructions that cause dependences between units by running the binary on a set of training runs. Then, it instruments the binary at the identified instructions to log unit boundaries and unit dependences during production runs. A causal graph is generated by analyzing both the standard audit log and our log. In the generated causal graph, a process is decomposed into many autonomous units with the corresponding partitioning of the objects (e.g., files, sockets) accessed by each unit, hence avoiding the dependence explosion as demonstrated earlier in the section.

Fig. 1(b) shows the graph generated by BEEP-induced log. In the graph, an oval represents a division of a process composed of a unit or a set of inter-dependent units. This causal graph accurately captures the minimal causal path between the root cause – the initial phishing email – to the symptom – the back door process. Observe that the three sockets connected to *sendmail* processes include the two connections that deliver the phishing email[2] and the one to the name server (75.75.75.75). The two sockets connected to the *firefox* process denote the local address (127.0.0.1) used for receiving the link from the *firefox* process spawned by *pine*, and the malicious web site (y.y.y.y:80), respectively. Observe that the phishing email is precisely pinpointed (the file between *procmail* and *pine*). The fifth row of Table I shows the statistics of BEEP.

The observation is that BEEP is much more accurate as it precisely captures the attack causal path. In the following sections, we further discuss the loop-based unit concept and technical details.

## III. Design Patterns of Long Running Programs

A key concept in BEEP is execution units derived from loop iterations. In this section, we conduct a study of the design patterns of (relatively) long running programs to illustrate the generality of the unit concept and explain the intuitions behind it.

### A. Unit

Short running programs unlikely cause dependence explosion as their entire execution is usually a cohesive semantic unit. In contrast, long running programs often process many independent inputs during their lifetime. Hence it is undesirable to use a single node to represent a long running process as in traditional causal analysis techniques because that'd force causal relations from different independent inputs to be undesirably related. Instead, we should partition it into autonomous sub-executions corresponding to independent inputs. The essence of BEEP is to leverage *top-level* event handling loops to perform such partitioning.

To identify the program structure and characteristics of long running programs, we study a large pool of popular open-source programs. We collect a list of 94 long running Linux applications in different categories, including 51 server programs in 10 categories and 43 UI progrmas in 12 categories. These applications are written in various languages (C, C++, Java, Python, Perl or Tcl). We summarize the results in Table II. Column 2 presents the category and column 3 shows the number of applications we collect for each category. Column 4 presents the number of applications with top level event loops. For example, in the 13 web server applications we study, all of them have their execution dominated by event loops.

Overall, all of the 94 applications have the loop-based, event-processing structure. In most cases, we observe that the event processing loop is inside the application code. In some case, we find that the event loop can reside in special libraries. For example, 12 of the UI programs use event

[2]The two connections have the same IP but different ports.

```
337  int main(..) {
..
1063 [UNIT] for(;;) {
..
1081     w3mFuncList[..].func(..);
..
1110     mouse_active();
..
1144     c=getch();
..
1178  } // for end
1179 } // main function end
```

[main.c]

(a) W3M-0.5.2

```
1093 void server_accept_loop(..) {
..
1118 [UNIT] for(;;) {
..
1168     newsock=accept(..);
..
1241     if((pid=fork()) == 0) {
1242       //Child process handles the request
..
1264       break; // child done
1265     } // if end, parent stays in loop
..
1304  } // for end
1305 } // server_accept_loop function end
```

[sshd.c]

(b) sshd-5.9

```
593  static void *listener_thread(..) {
..
631  [UNIT] while(1) {
..
742      rv=accept_func(..);
..
768      ap_queue_push(..);
..
798    } // while end
..
810  } // listener_thread function end
```

[server/mpm/worker/worker.c]

(c) Apache-2.2.21 Listener thread

```
820  static void *worker_thread(..) {
..
842  [UNIT] while(!workers_may_exit) {
..
862      ap_queue_pop(..);
..
894      process_socket(..);
..
899    } //while end
..
906  } // worker_thread function end
```

[server/mpm/worker/worker.c]

(d) Apache-2.2.21 Worker thread

Figure 2. Event Processing Loop Examples

| | Category | Total Applications | Loop structured Applications |
|---|---|---|---|
| Servers | Web server | 13 | 13 |
| | Mail server | 8 | 8 |
| | FTP server | 6 | 6 |
| | SSHD server | 2 | 2 |
| | DNS server | 9 | 9 |
| | Database server | 4 | 4 |
| | Proxy server | 2 | 2 |
| | Media server | 5 | 5 |
| | Directory server | 3 | 3 |
| | Version control server | 2 | 2 |
| | Remote desktop server | 2 | 2 |
| UI Programs | Web browser | 5 | 5 |
| | E-mail client | 5 | 5 |
| | FTP client | 5 | 5 |
| | Office | 2 | 2 |
| | Text Editor | 3 | 3 |
| | Image tool | 4 | 4 |
| | Audio player | 2 | 2 |
| | Video player | 4 | 4 |
| | P2P program | 6 | 6 |
| | Messanger | 2 | 2 |
| | File manager | 2 | 2 |
| | Shell program | 3 | 3 |

Table II
PROGRAMMING STRUCTURE OF LONG RUNNING APPLICATIONS

processing library such as GTK+, QT or KDE, and the main loop is located inside these libraries.

Through this study, we observe that a long running program is most likely driven by external events such as remote requests, keyboard and mouse inputs. The execution of such a program is dominated by event processing loop(s) and the functions that are directly/indirectly invoked by the loop body. For further investigation, we reduce our sample set and study more detailed structure of the selected long running programs. We pick target programs based on their popularity and category. Table III shows the programs we select.

In Fig. 2, we show the event handling loops from three representative applications that are single-process, multi-process, and multi-threaded, respectively. A loop with the **[UNIT]** annotation denotes that it is an event processing loop. Fig. 2 (a) shows the request handling loop of *W3M*, a text-based web browser. *W3M* is a single-process application: The main process receives and also handles input requests. At line 1063, *W3M* begins an infinite loop and each iteration handles a single user request. At lines 1110 and 1144, *W3M* receives user inputs from mouse and keyboard and processes the request at line 1081. The next request cannot be processed until the current one is done. Fig. 2 (b) shows a multi-process application *sshd*, the SSH daemon program. *Sshd* starts a loop at line 1118 and receives network requests at line 1168. Then it spawns a child process by calling `fork()` at line 1241. The spawned child process handles the received request and then terminates. After the parent process forks the child, it immediately starts the next iteration to receive the next request. Fig. 2 (c) and (d) show the event processing loops from *Apache*, a web server program. *Apache* is a multi-threaded application and it creates several threads to handle simultaneous requests. *Apache* creates two different classes of threads, listeners and workers. A listener thread receives a network request and then pushes the connection information into a queue. Worker threads keep checking the queue. Whenever the queue is not empty, a worker thread dequeues the connection information and handles the request. Fig. 2 (c) shows the code snippet of a listener thread and (d) shows a worker thread.

In all three applications, one iteration of an event loop always initiates a semantically independent sub-execution. In case (a), an iteration precisely corresponds to an independent sub-execution. In case (b), since the child process does not have event loops, its entire execution can be considered as one unit, which is causally induced by the iteration of the event loop in the main process. In other words, by treating an event loop iteration as a unit and the child process another unit and correlating them through the spawn event, we can isolate the independent sub-executions. In case (c), an iteration of the listener thread's event loop causes the execution of another iteration in the worker thread's event loop. Again, by treating the *top-level* event loop iterations (in the two threads) as units and correlating them through queue operations, we can isolate the independent sub-executions.

In fact, our study on the set of applications listed in Table III shows that all of them follow one of the three afore-

mentioned patterns. Note that these applications represent a set of commonly used programs that could be long running. This strongly supports our idea of treating top-level event loop iterations (or the entire execution in the absence of any event loop) as units to isolate independent sub-executions and the input/output data they process.

| | Applications | Type | Unit dependence |
|---|---|---|---|
| Servers | Sshd-5.9 | multi-process | Fork, Socket |
| | Sendmail-8.12.11 | multi-process | Fork, Socket |
| | Proftpd-1.3.4 | multi-process | Fork, Socket |
| | Apache-2.2.21 | multi-thread | Memory, Socket |
| | Cherokee-1.2.1 | multi-thread | Socket |
| UI Programs | Wget-1.13 | single | No |
| | W3m-0.5.2 | single | Socket |
| | Pine-4.64 | single | File |
| | MidnightCommand-4.6.1 | single | File |
| | Vim-7.3 | single | Memory, File |
| | Bash-4.2 | multi-process | Fork, Pipe |
| | Firefox-11 | multi-thread | Memory, Socket |
| | Yafc-1.1.1 | single | Socket |
| | Transmission-2.6 | multi-thread | Memory, Socket |

Table III
APPLICATION DESCRIPTION

### B. Inter-Unit Dependence

From previous discussion, we know that a unit alone may not correspond to a sub-execution handling an independent input, such as in the last two cases in Fig. 2. Instead, a few inter-dependent units together constitute an autonomous sub-execution. It is hence critical to detect dependences between units.

We further study the set of applications in Table III and observe the patterns that can induce dependences between units. As shown in the last column of the table, one popular pattern is through system-level events, such as file/socket reads and writes and process forking. These dependences can be easily detected from a standard audit log. For complex applications, such as *apache*, *firefox*, and *vim*, dependences caused by memory accesses (i.e. one unit writes to some memory and another unit reads it) are critical to detecting inter-unit dependences. For example, a listener thread in *apache* writes connection information to a queue, which is a data structure in memory, and a worker thread reads it from the queue memory. We also observe that the memory footprint of the queue is small and the queue is heavily reused by many threads. Hence, page-level memory dependence detection as in [20] cannot disambiguate the dependencies belong to different autonomous sub-executions. Hence, we need to instrument the memory access instructions corresponding to enqueue and dequeue operations to detect the dependences. In *vim*, files being edited are loaded to memory buffers and these buffers serve as the communication channels between different iterations of the event loop.

Another important observation is that besides the dependences denoting high-level workflow, there are also a large volume of low level dependences that cross unit boundaries including those caused by memory management, caching, and statistics collection. They are not interesting from the causal analysis perspective. We will discuss details of detecting inter-unit workflow dependences and precluding low level dependences of no analysis interest in Section IV-B.

## IV. IDENTIFYING UNIT LOOPS AND UNIT DEPENDENCES

Ideally, we would like to have source code and programmer annotations about units and unit dependences. However in practice, since programs on a system usually come from vastly diverse sources, it is very difficult to enforce the availability of such annotations or even the source code. In cases that we have the source code, it is still very difficult to identify units and unit dependences *statically* because the task entails a lot of static analysis that are hard to achieve good precision. For example, it is very challenging to statically identify program dependences representing high level workflow. A conservative approximation will result in a lot of unnecessary instrumentation, leading to high runtime overhead. Hence in this paper, we propose dynamic analysis to detect unit and unit dependences without source code. In the remainder of this section, we present these analysis and discuss why dynamic analysis is an appropriate design choice due to the nature of units and unit dependences.

### A. Reverse Engineering Unit Loops

In this section, we describe a technique to detect loops from application binaries that can produce units at runtime, called *unit loops*. We leverage two observations: (1) such loops (e.g. event handling loops) are most likely top level loops, that is, loops that are not nested in any other loops; (2) their loop body must make some system calls to receive inputs or produce outputs. Due to the difficulty of static binary analysis, we use a number of training runs to detect unit loops. For instance, it is difficult to statically determine whether a loop in a function is a top level loop as the function may be the target of some indirect function call inside another loop.

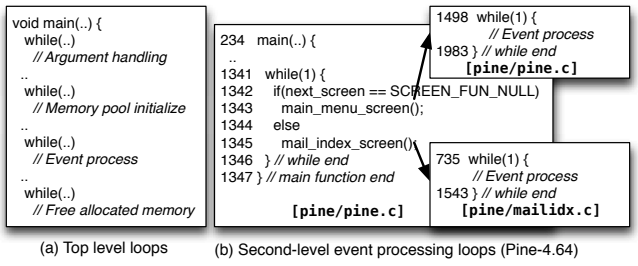



Figure 3. Loops in applications.

We cannot simply treat all top level loops as unit loops. Applications tend to have other top level loops for program initialization (e.g. argument parsing) and finalization. Fig. 3 (a) shows a typical design pattern for the applications we have studied. The key characteristic is that unit loops must interact with external environment. Most unit loops receive

inputs. Unit loops that retrieve and process tasks from a work queue may not receive inputs, but they produce outputs.

In some cases, top level loops are not sufficient as event handling loops may nest inside other loops. *Pine* is such an example. Fig. 3 (b) shows that a top level loop at line 1341 calls either `main_menu_screen()` or `mail_index_screen()` depending on if the next screen that needs to be displayed is the main menu or the email index. Both `main_menu_screen()` and `mail_index_screen()` have their own event processing loop at line 1498 of *pine.c* and line 735 of *mailidx.c*, respectively. When the user stays in the mail index screen and browses e-mail messages, the execution stays in the loop at line 735. *Pine* starts a new iteration of the top level loop at line 1341 when the user exits from the mail index screen. Similarly, execution stays in the loop at line 1498 when the user stays in the main menu.

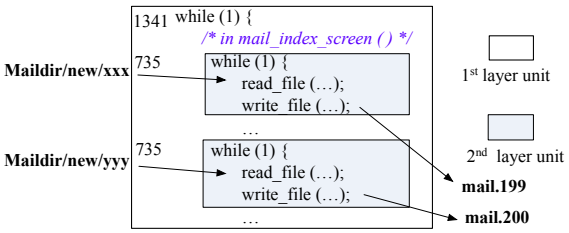



Figure 4. Two layers of units in *pine*. The 1st layer node represents an iteration of the main loop, in which the loop inside `mail_index_screen()` iterates multiple times. The two second layer nodes represent two consecutive iterations of the inner loop.

If we only consider the top level loop as the unit loop and miss the second level loops, spurious causal dependences will be introduced. Consider the partial causal graph in Fig. 4 for *pine*. There is a file read and a file write in each iteration of the inner loop. If the second layer units are not identified, all the iterations of the inner loop together are denoted by a single first layer node. Since each outgoing edge is considered causally dependent on all the incoming edges of the same node, the file write in the second iteration is undesirably dependent on the file reads in both iterations. With the second layer units, we can precisely attribute the file write to only the file read within the same iteration.

To tolerate such cases, our training system is configurable, allowing the user to specify the number of loop nesting layers to be considered. In this paper, we set it to 2, meaning that we look for unit loops in both the top level loops and those that directly nest in the top level loops. It is sufficient for the applications we consider.

In the training phase, our technique first constructs control flow graphs and call graphs for subject binaries using PEBIL [21], to identify loop heads and exits. Then we perform dynamic instrumentation using PIN [23] to log the beginning and ending of each iteration of all loops and system calls. We analyze the generated training log to filter out those loops that nest too deep or do not involve input/output syscalls. The remaining loops are considered as unit loops. Details are elided. Note that imprecision in static graph construction does not affect our technique as we rely on dynamic analysis to detect loop nesting and system calls.

The second to fourth columns of Table IV show the numbers of loops, high level loops, and unit loops in the applications we consider. Observe that despite the large number of loops, our technique can detect a very small number of unit loops. With the nesting configuration of 2, we identify three unit loops from *Pine* as discussed earlier. *Apache* has two different unit loops, one for the listener thread and the other the worker thread. *Firefox* has nine unit loops, one for each different type of thread.

**Discussion.** BEEP's unit loop detection is through dynamic analysis and heuristics. Hence, it is theoretically incomplete and unsound. However, incompleteness and unsoundness does not cause problems for us in practice. First of all, it is very unlikely for BEEP to miss the first layer event loops as applications rely on their event loops to operate. In contrast, we may miss unit loops in the lower layers if the configured number is too small. The effect of missing lower layer unit loops is illustrated by Fig. 4, which is undesirable unification of dependences. However, due to the modular design of most existing software, it is unlikely that developers put their event loops or dominant operation loops deep inside other loops. Nesting level of 2 is the maximum we have encountered.

In the presence of unsoundness, we may detect loops that should not be considered as unit loops. For example in Fig. 4, assume the second file write is indeed causally dependent on both file reads, it is hence not necessary to have the second layer units. However, despite the *redundancy*, having the second layer units does not affect the *correctness* of the causal graph as in the later cross-unit dependence analysis phase, we will detect the causal dependence between the two second layer units so that we are able to correlate the file write to both file reads. Since our technique is very lightweight, the overhead caused by redundancy may not matter. These arguments are supported by our empirical results in Section VII.

### *B. Reverse Engineering Inter-Unit Dependences*

From our previous discussion in Section III-B, capturing inter-unit dependence is important to isolating semantically independent sub-executions constituted by multiple units. Since dependence caused by system level events can be easily detected from audit log, We in this section focus on reverse engineering inter-unit dependence through memory.

**Workflow Dependence vs. Low Level Dependence.** In practice, there are lots of memory dependences crossing unit boundaries. Only some of them are helpful in separating units belonging to semantically independent sub-executions. We call them *workflow dependences* or high

level dependences. Other cross-unit dependences are not part of the workflow, but rather caused by low level behavior such as logging and memory management. They usually induce arbitrary dependences between units. Sometimes, all units are inter-connected by such dependences. Hence, the challenge lies in how to distinguish these two types of dependences.

In the following, we first use examples to illustrate these two types of dependences. We then determine the unique characteristics of workflow dependences. Finally, we present a dynamic analysis to reverse engineer such dependences.

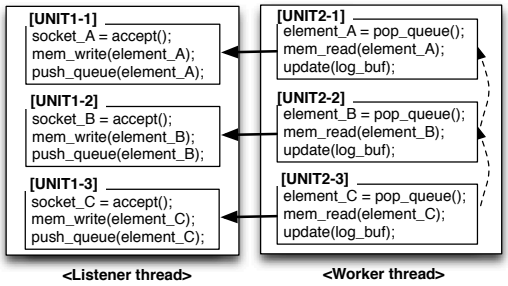



Figure 5. High level and low level memory dependences in *apache*. Unit1_2 means the second unit instance from loop 1.

*Apache* and *firefox* record each request processed in its lifetime. After a request is handled, they write the request information to a memory buffer which is flushed periodically. We observe that log buffer updates cause dependences between most units in the process. In particular, there is a meta data structure for the buffer that maintains the current space occupied by messages, the number of messages, etc. Each update has to first read the data structure written in the last update, and write to it afterwards.

Fig. 5 shows a number of units in an *Apache* execution and their memory accesses. In this example, we observe two kinds of memory dependences. One is through the queue data structure (solid arrows) and the other is through the log buffer (dotted arrows). Observe that the first type of dependences capture the causality between units that handle the same request, and thus workflow dependences. The second type correlates all units in a worker thread regardless of the request, and thus low level dependences.

Our study of other applications in Table III show that low level dependences may have various forms. For instance, a program using object pools tends not to directly allocate an object from heap, but rather from a pool, and returns it to the pool after using it. Hence, dependences can be induced between units through the meta data structure of the pool regardless of the input. A program that needs to report its own execution status constantly updates some global statistical variables, leading to low level dependences. We summarize the characteristics of workflow dependences as follows.

- Two units that have workflow dependence often share a common heap object, called the *workflow object*. This object is often closely tied with the common input that the two units are processing. It is a *heap* object instead of a *global* or *stack* object because the number of inputs varies a lot at runtime, demanding different numbers of such objects. In Fig. 5, memory structure `element_A` is a workflow object containing request information and it is shared by **[UNIT1-1]** and **[UNIT2-1]**.
- Units spawned from *the same loop* have unique workflow objects. In particular, the various units from the same loop correspond to different inputs and hence should operate on different workflow objects. In contrast, data structures causing low level dependences, such as the log buffer in *apache*, are not exclusively owned by any unit or subsets of units, and they can be accessed by any units. In Fig. 5, the three units from the same event loop in the listener thread operate on three different workflow objects, denoting three different input requests.

**Reverse Engineering Algorithm.** Based on the characteristics of workflow dependences, we devise the following dynamic analysis to reverse engineer the instructions that can produce workflow dependences at runtime. They will be instrumented for production runs.

1) We instrument libc memory allocation functions to detect all heap objects and their sizes.
2) We instrument all memory accesses to check if an access targets on any of the allocated heap object. If so, we log the access.
3) We instrument all the unit loops identified in the previous phase to log the begin and the end of a unit. Essentially, we log each instance of a unit loop head. The execution between two consecutive instances of the loop head denotes a unit.
4) We then associate all the heap objects to the units in which they are accessed. For a heap access instruction inside a unit loop, if it accesses unique heap objects in different units and these objects cause inter-unit dependences, we consider it a unit dependence inducing instruction. In other words, if an instruction ever accesses the same heap object in multiple units, it is excluded; if the object accessed by an instruction can never cause cross-unit dependence, it is excluded.

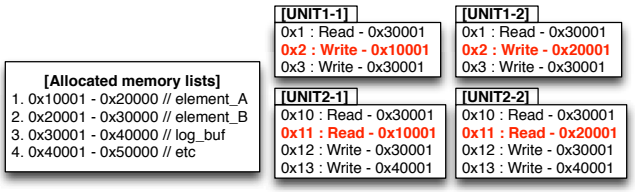



Figure 6. Training log example of *apache*. Allocated heap objects on the left; the access log by units on the right.

Fig. 6 shows a simplified training log of *Apache*. In this example, four different heap regions are allocated, two for the elements storing input requests, one for the log buffer, and the other (etc) is used internally for statistics

collection. Our algorithm determines that instructions 0x1, 0x10, and 0x12-13 are excluded as multiple units from the same loop access the same object. In contrast, instructions 0x2 and 0x11 are identified as workflow dependences related instructions.

**Optimization.** Our algorithm may identify multiple instructions that cause workflow dependences on the same heap object. For example, multiple fields of a request data structure in *Apache* are defined by different instructions in a unit of the listener thread, and then used by a worker thread unit. Although these field accesses are by different instructions and with different addresses, they access the same allocated region and cause the same workflow dependence between the same two units. Hence, we can select one representative field access and ignore the others. In particular, if a set of accesses of the same object always appear together, we select the first one.

Columns 5-7 in Table IV present the results of memory dependences. The fifth column shows the number of memory instructions accessing heap objects that can cause dependences across unit boundaries, including both workflow and low level dependences. The sixth column shows the instructions causing workflow dependences and the last column shows the number after the optimization. They are the ones that get instrumented.

| Applications | # of loops | | | # of memory instructions | | |
|---|---|---|---|---|---|---|
| | total | 1 and 2 level | syscall included | total | high level | after opt. |
| Sshd-5.9 | 1,079 | 29 | 1 | 0 | 0 | 0 |
| Sendmail-8.12.11 | 798 | 61 | 1 | 0 | 0 | 0 |
| Proftpd-1.3.4 | 812 | 86 | 1 | 0 | 0 | 0 |
| Apache-2.2.21 | 1,294 | 31 | 2 | 67 | 31 | 2 |
| Cherokee-1.2.1 | 7 | 3 | 1 | 0 | 0 | 0 |
| Wget-1.13 | 519 | 32 | 1 | 96 | 0 | 0 |
| W3m-0.5.2 | 1,471 | 121 | 1 | 0 | 0 | 0 |
| Pine-4.64 | 5,513 | 155 | 3 | 6 | 0 | 0 |
| MC-4.6.1 | 1,215 | 153 | 1 | 0 | 0 | 0 |
| Vim-7.3 | 4,359 | 292 | 1 | 412 | 6 | 6 |
| Bash-4.2 | 1,840 | 179 | 1 | 38 | 0 | 0 |
| Firefox-11 | 25,564 | 1,863 | 9 | 671 | 12 | 2 |
| Yafc-1.1.1 | 469 | 7 | 1 | 0 | 0 | 0 |
| Transmission-2.6 | 1,117 | 10 | 2 | 14 | 4 | 2 |

Table IV
IDENTIFIED UNIT LOOPS AND INTER-UNIT MEMORY DEPENDENCES.

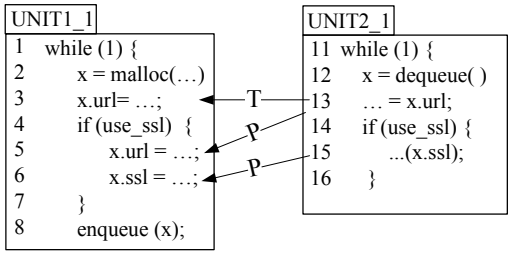



Figure 7. Missing dependences in training runs. During training, `use_ssl` is never set. The memory dep with a 'T' label is the workflow dep detected in training. The deps with a 'P' label are those happening in production runs.

**Discussion.** As we detect workflow dependence inducing instructions through training runs, depending on the coverage we may not detect the complete set of such instructions and hence we may theoretically miss workflow dependences at runtime. However this is unlikely in practice. The reason is that workflow dependences denote how a program works at a very high level so that they are very stable. For instance, *Apache*'s workflow is that a request is received by the listener thread and put in a work queue. It is later dequeued and processed by the worker thread. This workflow does not change with different inputs. Therefore, as long as we are able to instrument at least one of the many memory writes of the workflow object in the listener thread, and one of the many memory reads of the workflow object in the worker thread, we can detect the workflow dependence.

Consider an example in Fig. 7, abstracted from *Apache*. Assume in the training runs, the variable `use_ssl` is never set. So the dependence between lines 3 and 13 is the only workflow observed and only instructions 3 and 13 are instrumented. Assume during production runs, variable `use_ssl` is set. Hence the real memory dependences become $13 \rightarrow 5$ and $15 \rightarrow 6$. However, as we instrument lines 3 and 13, we log a write to `x.url` at 3 and later a read of `x.url` at 13. Our log analysis concludes that there is a memory dependence between 13 and 3, which does not precisely reflect the true memory dependence, but sufficiently reflect the workflow. If in the training runs, `use_ssl` is always set, both instructions 3 and 5 will get instrumented as both access a heap object unique to an iteration and the object causes inter-unit dependences (Rule 4 of the reverse engineering algorithm). As such, even when `use_ssl` is not set in a production run, the same workflow can be detected nonetheless.

This illustrates the detection of workflow dependence is not by accident, as `x.url` is such an important field that it must be written by the first unit and read by the second unit, in both training and production runs. In Section VII-C, we will show that different coverage of training runs have little impact on the effectiveness of BEEP.

In the worst scenario, if the entire workflow is not covered by any training runs, our technique is unlikely effective. However, since our training runs are the regular use cases provided by the user, we argue that the technique is sufficiently effective for the particular user. Also improving coverage of training runs is an orthogonal challenge that we can leverage other researchers' solutions to mitigate [7], [5]. We leave it as our future work.

We may also fail to exclude some low level memory accesses if they happen to demonstrate the same characteristics as workflow dependences in training. This may lead to bogus inter-unit dependences. In practice, we rarely find such problems. From Table IV, we can reduce to a small set of instructions during training, indicating the criteria we use are distinctive enough.

| rule | instruction | instrumentation |
|---|---|---|
| $R_{entry}$ | entry of unit loop $l$ [1] | nesting++; |
| $R_{head}$ | head of unit loop $l$ | kill(UNIT, $l$, $inst(l)$++, nesting) [2]; |
| $R_{exit}$ | exit of unit loop $l$ | nesting- -; |
| $R_{wrt}$ | memory write to $a$ [3] | kill(MEM_WRT, $a$); |
| $R_{rd}$ | memory read from $a$ [3] | kill(MEM_RD, $a$); |

[1]Loop entry is before the first iteration.
[2]It is implemented by multiple `kill()` calls that take 2 args.
[3]These are reads/writes that may cause workflow deps.

Table V
LOGGING INSTRUMENTATION. UNIT, MEM_WRT, AND MEM_RD ARE NEW SYSCALL IDS FOR OUR ANALYSIS.

## V. LOGGING INSTRUMENTATION AND LOG ANALYSIS

Through the training phase, we reverse engineer the unit loops and instructions that could induce inter-unit memory dependences. In the next step, we instrument the applications accordingly to collect runtime log. Attack investigation is performed by co-analyzing the audit log and the additional log generated by BEEP.

**Logging Instrumentation.** To enable easy log analysis, we leverage the existing audit system to store our log. Specifically, we instrument an application binary at the instructions of interest such that special system calls are triggered. The system calls are then captured and logged by the audit system. The detailed instrumentation rules are presented in Table V. Specifically, the nesting level is increased by one before entering the first iteration of a unit loop (rule $R_{entry}$) and decreased when exiting the loop (rule $R_{exit}$). A unit event is emitted upon each instance of a unit loop head instruction (rule $R_{head}$), containing the id of the loop, the instance, and the nesting level. Note that loop entries and exits are statically identified, regardless of the coverage of training runs. We rely on the underlying binary instrumentation tool PEBIL [21] to achieve this goal. In some extremal cases (e.g. strange loop structures), PEBIL may miss some loop entries or exits. The consequence is that we may coalesce multiple units into one, losing precision. We have not encountered such cases in practice. In the generated log, a unit is delimited by two consecutive unit events of the same loop. Note that the audit system automatically annotates each kill signal it receives with the id of the signaling process.

**Log Analysis.** We develop an algorithm to analyze the enhanced audit log to generate a causal graph. This algorithm is applied when the user observes a symptom and wants to identify its root cause. It is undesirable to expose the user to units or memory dependences because these are application specific artifacts. Therefore, our design goal is to have a precise causal graph at the system level, that is, the graph is composed of system level entities such as files, sockets, and processes. However, a process node does not represent the entire process, but rather just a sub-execution constituted by the relevant units, i.e. *all the units that are reverse-reachable from the symptom through inter-unit dependences.* All the system calls occur in those units are connected to the process node.

**Algorithm 1** Log Analysis Algorithm

Input: $L$ - the audit log
$n$ - the nesting level considered.
$e_c$ - symptom event.
Output: $G$ - the generated causal graph.
Def: *memUse* - memory uses that look for definitions.
*sysObj* - system objects relevant to the symptom.
$unitRel[u]$ - if unit $u$ is relevant.
$curUnit[pid]$ - the current unit of process *pid*.

1: $memUse \leftarrow \{\}$
2: $sysObj \leftarrow \{$ the system objects of $e_c$ $\}$
3: **for** each event $e \in L$ in reverse order, starting from $e_c$ **do**
4: **if** $e$ is unit event UNIT($pid_e$, $uid_e$, $inst_e$, $n_e$) **then**
5: **if** $n_e \equiv n$ **then**
6: $curUnit[pid_e] \leftarrow (uid_e, inst_e)$
7: **if** $e$ is mem write in *pid* to $a \wedge a \in memUse$ **then**
8: $memUse \leftarrow memUse - \{a\}$
9: $unitRel[curUnit[pid]] \leftarrow true$
10: **if** $e$ is sys event in *pid* on $s \wedge s \in sysObj$ **then**
11: $unitRel[curUnit[pid]] \leftarrow true$
12: $G \leftarrow G +$ *edge* $(pid, e, s)$
13: **if** $e$ is mem read in *pid* to $a \wedge unitRel[curUnit[pid]]$ **then**
14: $memUse \leftarrow memUse \cup \{a\}$
15: **if** $e$ is sys event in *pid* on $s \wedge unitRel[curUnit[pid]]$ **then**
16: $sysObj \leftarrow sysObj \cup \{s\}$
17: $G \leftarrow G +$ *edge* $(pid, e, s)$

Furthermore, recall that depending on the configuration, we may have nesting units. Our algorithm allows the user to generate causal graphs at different nesting levels. For example, if two layers of units are logged, the user can choose to generate a causal graph with the first layer units only (less precise but concise) or the second layer units (more precise but larger).

Algorithm 1 describes the process. It takes as input the log file, a symptom event (e.g. a file or a process), and the level of units to be considered, and generates the causal graph relevant to the symptom. Initially, it populates the *sysObj* set with the system object(s) of the symptom event (line 2). Any events in the past that operate on the same object(s) have dependence with the symptom event. The main body of the algorithm traverses the logged events in the reverse order, starting from the symptom event. If it encounters a unit event, it updates the current unit if the nesting level is equivalent to the specified value (lines 4-6). For example, if the specified level is one, all the second layer units are ignored. Note that they must be nesting in some first layer unit. All of them are considered part of the first layer unit. In lines 7-9, if a memory dependence is detected, the current unit is considered relevant. Similarly in lines 10-12, if a system level dependence is detected, the current unit is considered relevant and an edge is added to the causal graph. After the current unit is determined relevant, all the

preceding memory reads and system objects are put into the memory use set and the system object set to look for further dependences (lines 13-17).

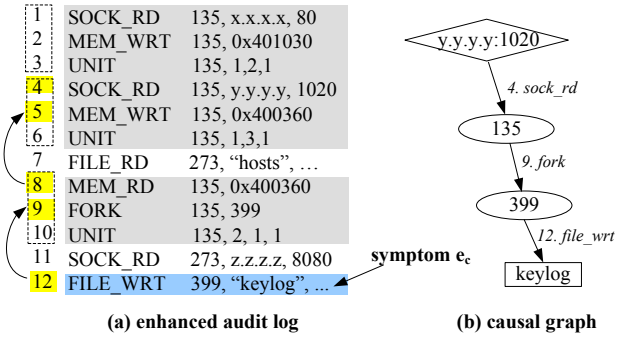



Figure 8. Log analysis example. The first value of an event is process id. The events of different processes have different shades. The line numbers of relevant events are highlighted. Event line numbers in the same unit are boxed. "Unit 135,2,1,1" denotes the 1st instance of unit loop 2 in process 135, and nesting level 1.

Fig. 8 presents an example. The symptom is the last file write event of process 399. It is dependent on the fork event at line 9, which is in a unit of loop 2 in process 135. The unit is marked relevant, entailing the memory read at line 8 being inserted to the memory use set to look for its definition, which is found at line 5, making the enclosing unit relevant and the socket read in the unit at line 4 being added to the graph. Observe the events in the first unit are not relevant although it is in the same process 135. Also observe that units from different processes may interleave.

While the aforementioned algorithm traverses the log *backward* to identify all the events leading to a symptom, a similar algorithm is also developed to perform a *forward* traversal to identify all the events that are caused by a root cause. This is particularly beneficial in understanding the ramifications of an attack. The details are omitted.

## VI. Implementation

We have implemented a BEEP prototype composed of the training, instrumentation, and log analysis components. The training analysis is implemented on PIN [23], instrumentation is through a binary rewriting tool PEBIL [21]. Log analysis is implemented in C++. Both the backward and forward log analysis algorithms are implemented.

In Linux, syscall `clone()` is used for spawning both a process and a thread. On one hand, this allows individual threads across processes have unique ids. Consequently, we can attribute an event to its thread through the unique pid and then to its unit. On the other hand, we need to distinguish a thread from a process to detect memory dependences across threads. Note that threads in the same process share the same virtual space so that when we match a memory use with a definition, we need to ensure the corresponding threads belong to the same process; we should not match a use with a definition in a different process. We parse the argument of a clone syscall and see if `CLONE_THREAD` or `CLONE_VM` is set. If either one is set, the child is a thread, otherwise a process.

## VII. Evaluation

Part of the experimental results have been presented in previous sections. In this section, we present the time and space overhead and two additional case studies. For training runs, we use 1-3 executions for each application. They are the most common use cases of those applications.

The training overhead is about 10x-200x. Next we focus on reporting the more interesting logging overhead.

### A. Logging Overhead

In this experiment, we measure execution time of individual applications. For server programs, we use test inputs provided with the program if available, random inputs otherwise. For UI programs with batch mode support, such as *vim* or *W3M*, we use scripts to feed inputs. For *firefox*, we use the *SunSpider* browser benchmarking tool [3]. It is difficult to measure runtime for programs such as *pine* and *bash* because they are highly interactive without batch mode support. We hence preclude them in the time overhead study.

Fig. 9 shows the logging overhead of BEEP under different conditions. The "*Audit Off*" bars show the overhead of BEEP over the execution of the original binaries without turning on the Linux audit system. The "*Audit On*" bars show the overhead of BEEP over the execution of the original binaries with the Linux audit system running. In the later case, the baseline includes the running time of the Linux audit operations.

In most cases, the overhead is less than 1%.

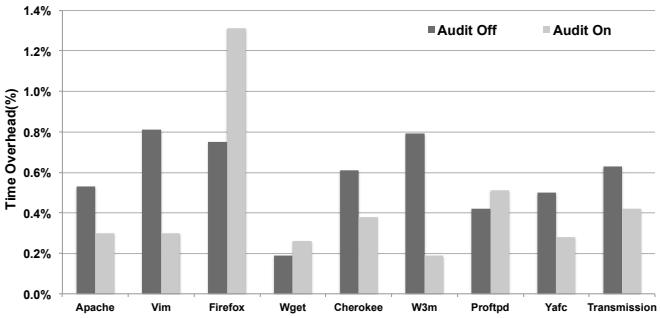



Figure 9. Runtime overhead.

Table VI shows space overhead. It is measured over executions of about one minute. Note that since we use batch mode, the executions are highly intensive. "*Original*" log means the audit log without our technique. Observe that our technique has small space overhead for most of applications except *firefox* and *apache*. *Firefox* and *apache* have relatively high space overhead because a single request from the user or network is handled by multiple units from multiple

threads and each of them logs units and also memory writes and reads.

| App | Execution time(sec) | Log size(MB) | | Space overhead(%) |
|---|---|---|---|---|
| | | Original | Instrumented | |
| Apache | 58.48 | 0.227 | 0.278 | 22.4% |
| Vim | 59.95 | 85.75 | 89.4 | 4.2% |
| Firefox | 95.1 | 46.12 | 63.25 | 37.1% |
| Cherokee | 61.01 | 1.1 | 1.126 | 2.2% |
| W3M | 58.92 | 26.88 | 28.39 | 5.6% |
| Proftpd | 58.65 | 0.54 | 0.56 | 2.3% |
| Wget | 60.37 | 17.47 | 17.48 | 0.6% |
| Yafc | 59.16 | 7.29 | 7.41 | 1.6% |
| Transmission | 60.51 | 4.92 | 5.37 | 9.1% |
| Pine | - | 5.62 | 6.21 | 10.5% |
| Bash | - | 0.63 | 0.64 | 0.6% |
| MC | - | 1.56 | 1.67 | 6.9% |
| Sshd | - | 2.2 | 2.22 | 0.8% |
| Sendmail | - | 0.214 | 0.22 | 2.8% |

Table VI
SPACE OVERHEAD

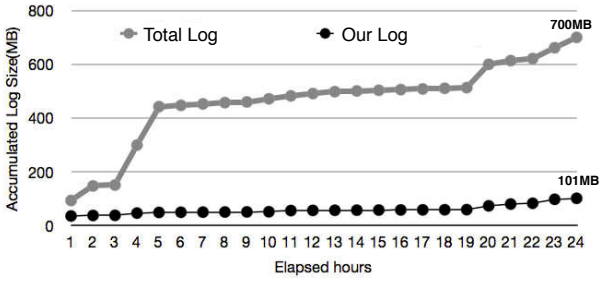



Figure 10. Accumulated log size from one-day execution.

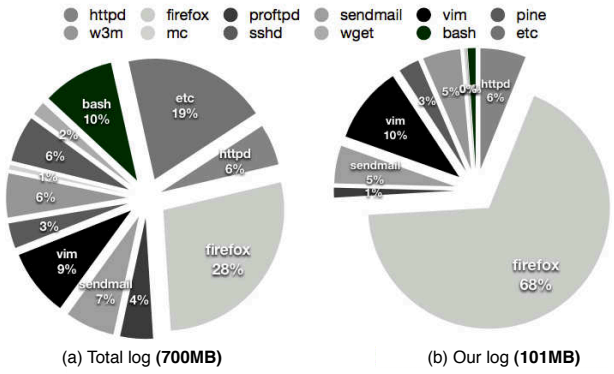



Figure 11. Log size of applications.

**Scalability.** To shed some light on the scalability of BEEP, we conduct the following experiment. We created a VM image with all the instrumented applications and installed it on a (non-author) user's laptop. He used the system for 24 hours with 8 hours sleep time. We then analyze his log. Fig. 10 shows the log size changes over time. Observe that the log generated by our technique is consistently a small portion of the whole audit log. The growth is linear. Finally, about 700MB log was generated with 100MB from our technique. Fig. 11 shows the breakdown for individual applications. Observe that a program that generates a lot of audit log may not generate a lot of fine-grained log through BEEP, such as *bash*, and vice versa (e.g. *firefox*).

| | process # | file # | com. channel # | edge # |
|---|---|---|---|---|
| Linux audit | 95.29 | 188.31 | 201.12 | 785.98 |
| BEEP | 7.19 | 18.08 | 10.86 | 67.29 |

Table VII
CASUAL GRAPH COMPARISON FOR 100 FILES.

**Systematic Evaluation of Effectiveness.** In this experiment, we randomly select 100 files accessed in the 24-hour experiment, backtrack from these files, and compare the causal graphs generated by BEEP and standard system call based techniques. We observe dependence explosion ($>$ 10 times difference in graph size) in 74 out of the 100 files. Table VII compares the averages. Observe that the causal graphs by BEEP are on average 13.5 times smaller.

### *B. Case Studies*

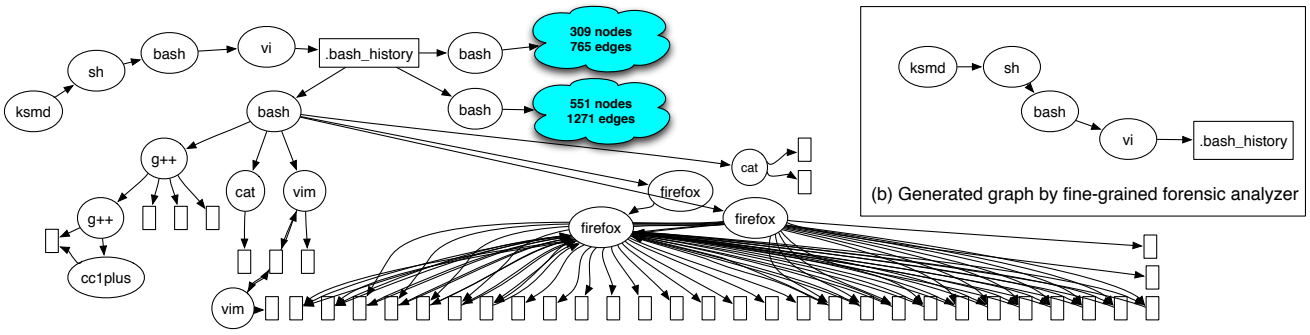
We have shown one case study in Section II. In this section we show two more case studies to demonstrate the effectiveness of BEEP.

**Understanding Attack Ramifications.** During the 24 hours scalability study, we have another colleague using the offensive BackTrack Linux VM [1] on a different machine to attack the machine with the audit system running at the 13th hour. BackTrack provides a set of pre-installed hacking tools.

We provide the IP address of the "victim" machine to the (emulated) "attacker." The "attacker" first used a scanning tool to scan all the open ports and the application versions on the "victim" machine and was able to find that the *Proftpd* ftp server that has opened port 21 has a vulnerability. He used another tool to exploit the vulnerability and acquired a root shell. He then installed a backdoor in the "victim" machine and made another connection to the backdoor to collect system and user information. After that he modified *.bash_history* file to remove the footprints of the attack.

At the end of the 24-hour period, the user of the "victim" machine was notified of the backdoor program. He first performed a backward causal analysis using both the standard audit log and the enhanced log. Both generate a concise causal graph that precisely identifies the root cause. Audit log works well in this case as none of the processes along the path from the exploit to the back door process is long running, and hence no explosion.

The user also wanted to understand the ramifications of the attack, namely to identify the objects and processes that get compromised by the attack. This can be done by generating a causal graph involving all the events *reachable from* the root cause through dependences. This time, the Linux audit log leads to dependence explosion, whereas BEEP produces the precise causal graph. Fig. 12(a) shows the graph generated by Linux audit log. It contains 348 processes and 512 files, including three *bash* sessions which the user connected to after the attack. Fig. 12(b) shows the




(a) Graph generated from syscall log. It is a much smaller reduced version.

Figure 12. Identifying Attack Ramification.

graph generated by BEEP and it only shows 4 processes and 1 file, clearly identifying that the *.bash_history* file gets compromised by someone sneaking in through the back door (i.e. *sh*→ *bash*). The reason of the explosion is that whenever the user logged into the system, the corresponding *bash* process is tainted because it first reads the compromised *.bash_history* file. Then all child processes of the *bash* process also get tainted and all objects accessed by these processes are also included in the graph. In contrast, BEEP partitions a *bash* execution into units and there is no workflow from the unit that reads the *.bash_history* to the units that spawn child processes.

The file-offset approach [26] does not reduce any false dependences because *bash* reads the whole *.bash_history* file when it executes. The socket interval technique [12] does not work because *firefox* is tainted from the beginning of its execution and then all the following socket accesses are also tainted. Table VIII compares the statistics of the four techniques.

Using timestamps to approximate causal relation hardly works here as the bash processes are long running. Precluding *.bash_history* from the white list and thus avoiding tracking dependences through it may prevent the explosion, but that would lead to missing the malicious action of modifying the file. Also, there are many other such configuration files and log files. The system administrator has to decide which ones should be precluded. It is in general not safe to do so either as the attacker can intentionally use configuration/log files as their communication channels.

| | process # | file # | com. channel # | edge # |
|---|---|---|---|---|
| Linux audit | 348 | 512 | 0 | 2,135 |
| File Offset [26] | 348 | 512 | 0 | 2,135 |
| Socket Interval [12] | 348 | 512 | 0 | 2,135 |
| BEEP | 4 | 1 | 0 | 4 |

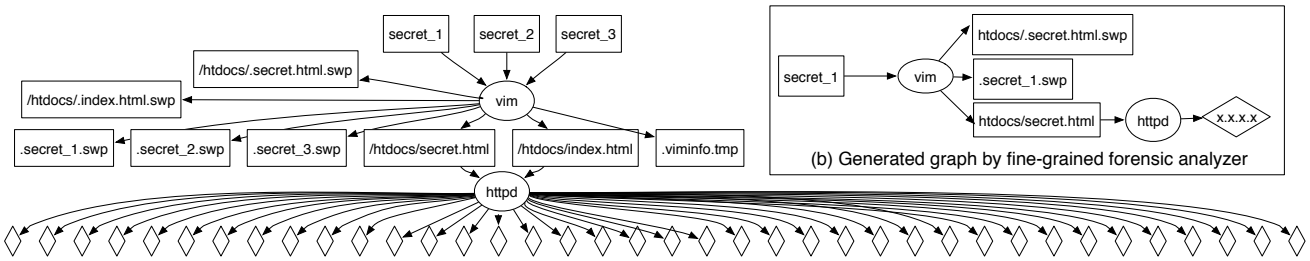
Table VIII
CASUAL GRAPH COMPARISON.

**Information Theft.** In this case study, we emulate the following scenario: an employee executes *vim* editor and opens three secret files (secret_1, secret_2 and secret_3) and two other html files(index.html and secret.html) on a server in his company. He copies secret information from secret_1 file and pastes it to secret.html file. Then he modifies the index.html file to generate a link to the secret.html file. After he gets home, he starts a web-browser on his home system and connects to the company's server to download the secret file. Note that the company server is set up in such a way that direct copying secret files is forbidden, but viewing them is allowed by users from the company's physical LAN.

Later, the company finds out some of the important information is leaked and the administrator tries to investigate how the secret was stolen and who shall be held responsible. She constructs a casual graph related to the three secret files. Fig. 13(a) shows the graph generated by the existing techniques. This graph indicates all three secret files are read by *vim* and two html files are written from the same *vim* process. Then those html files are read by the *httpd* web server daemon. The web server has served requests from 38 connections since the html files are read. From this graph, the only conclusion we can draw is that maybe all three secret files are leaked and the destination can be any of the 38 connections.

Neither the file-offset or the socket-interval approach reduces any false dependence. When the employee executes *vim* and opens a file, *vim* first reads the entire file and then generates a swap file. Also when he modifies the part of the html file, *vim* does not directly modify the original file, instead it modifies the corresponding swap file and renames it to the original file name when he saves modifications. That means even though the employee reads a part of the secret file and modifies a part of the html file, *vim* reads and writes entire files internally. The socket interval technique does not work because the *index.html* file is tainted and all the corresponding socket accesses are tainted.

One simple heuristic is to find the last secret file read that happens right before the write to the html file based on timestamps. However the user opens all three secret files in the beginning of *vim* execution and performs edits in arbitrary order. The two html files are written only when the user indicates so. In this case, the closest read is from secret_3 which is not leaked. Another heuristic is that the administrator manually inspects the secret files and the html files to find the secret source and the sink by comparing their contents. However, the audit system cannot afford




(a) Generated graph by previous forensic analyzer

Figure 13. Information stealing.

logging contents (e.g. packets that the web server has sent). Moreover, a smart attacker can apply packer or compressor to encode the stolen content.

Fig. 13(b) shows the graph generated by BEEP. It precisely pinpoints the insider's home machine IP and the secret that gets leaked. BEEP is able to detect all necessary workflows between units including the one between the *vim* unit that copies the content and the unit that pastes it. The secret.html file is only read by one unit in *httpd* and sent to an external IP through another unit. Table IX shows the comparison.

| | process # | file # | com. channel # | edge # |
|---|---|---|---|---|
| Linux audit | 2 | 11 | 38 | 51 |
| File Offset [26] | 2 | 11 | 38 | 51 |
| Socket Interval [12] | 2 | 11 | 38 | 51 |
| BEEP | 2 | 4 | 1 | 6 |

Table IX
CAUSAL GRAPH COMPARISON.

### C. Impact of Training Coverage

In this experiment, we study the effect of training run coverage. For each program, we create three test suites. Training #1 is the universal suite, containing the common use cases. Training #2 is a subset of training #1, we randomly select 30%∼50% of the test cases from training #1 so it covers a smaller portion of the code. Training #3 is a subset of training #2 in which we randomly include 30%∼50% from training #2. For example, in the *apache* case, we acquire a real-world web request log for 3 days and we write a script to regenerate the workload and use them as training #1. We randomly select a one-day log for training #2 and also select a five-hour log for training #3. Table X shows the function and instruction coverage of each training set. Observe that code coverage are quite different for the three sets of each program.

| Applications | Training # 1 Coverage | | Training # 2 Coverage | | Training # 3 Coverage | |
|---|---|---|---|---|---|---|
| | Fnc | Ins | Fnc | Ins | Fnc | Ins |
| Apache | 74.1% | 48.2% | 51.3% | 30.9% | 39.4% | 20.7% |
| Vim | 60.7% | 47.4% | 48.2% | 31.7% | 29.6% | 16.4% |
| Transmission | 59.6% | 41.5% | 51.4% | 32.1% | 39.8% | 21.3% |
| W3m | 64.2% | 50.4% | 58.7% | 43.7% | 41.8% | 29.0% |
| Pine | 52.1% | 27.4% | 29.6% | 18.4% | 21.8% | 11.1% |

Table X
EFFECT OF TRAINING.

We first verify correctness of unit loop detection and observe that all three training sets of each application correctly find the same unit loops.

Then we detect inter-unit workflow dependences using the different training sets. *W3m* and *Pine* do not have any workflow dependence and our technique correctly identify that for all the training sets. In cases of *Apache* and *Transmission*, inter-unit dependence detection results from all the three sets are the same. All of them correctly identify the same set of memory instructions which may cause dependences across units. In *Vim*, the detected instruction set by training #1 is different from that by training #2. Training #2 and #3 have the same result. However the two sets both access the same heap objects and both of them can be used to correctly detect inter-unit dependences in all scenarios we have evaluated.

From this study, we observe that training run coverage has little effect on BEEP.

## VIII. RELATED WORKS

**Coarse-grained Logging.** Our technique follows the line of work in tracking system-level dependence for forensic analysis including backward tracking to identify the source of an attack [18], [15], [4], [11] and forward tracking to detect the effect of an attack [12], [19], [17], [31], [25], [9]. Our contribution is to complement these techniques by partitioning an execution to autonomous units that handle independent inputs to avoid dependence explosion.

Some existing efforts try to mitigate the dependence explosion problem with system level techniques. In [20], page level memory dependences are detected and logged in addition to syscall logging. While it provides better precision, it is not generally effective due to its limitation to the page level granularity and its unawareness of the semantics of those dependences. In [26], file offsets are logged additionally for file related syscalls such that file dependences can be better disambiguated. However, it is only limited to file dependences.

**Whole System Replay.** To allow forensic analysis and to roll back a victim system after an attack, system level replay techniques have been proposed in the past [17], [8]. These techniques regularly create checkpoints and record non-deterministic factors in system-wide execution such as packets from remote sites, inputs from users, and hardware signals so that the whole system can be replayed from a

checkpoint. During replay, the part that is affected by the attack can be skipped [17]. BEEP can complement these techniques by providing better accuracy in causality analysis and supporting the replay of benign units of a tainted process.

**Combination of Coarse-grained Logging and Instruction Logging.** Techniques have been proposed to detect causality between system calls by checking if there are instruction level dependences between system calls [29]. These techniques require instrumenting a lot of instructions as they do not distinguish low level and workflow memory dependences. Furthermore, causal relations induced by low level dependences – thus of no forensic importance – are in-distinguishable from the important ones. In comparison, the introduction of units allows us to define workflow dependences. According to our results, detecting workflow dependences requires very little instrumentation and provides high accuracy.

Magpie [6] monitors kernel and application level events to extract control flow and system resource usage of an input request in a server. Then Magpie uses application specific schema to identify correlated events from both applications and kernel. Magpie needs predefined event schema for each application, whereas BEEP only requires a few training runs. The instrumentation in BEEP is also more lightweight than Magpie.

In our previous work [22], we proposed an execution reduction technique that aims to faithfully replay a failure with a reduced log. Similar to BEEP, our previous work focused on reducing loop iterations (also called units.) However, the previous technique requires source code and manual annotations to define the loops to reduce; whereas BEEP only requires binary programs for automatic unit loop identification. To achieve faithful replay, our previous technique has to consider all memory dependences without distinguishing low level and workflow dependences. The entailed instrumentation is much more heavy-weight than BEEP.

Fredrikson et. al. [10] proposed a technique allowing users to specify the granularity of logging and analysis, and define the notion of event causality. Despite its flexibility, it needs a lot of human user involvement.

**Information Flow.** Digital forensics systems [24], [27], [30] monitor and log flow of requests from network to detect and analyze malware. Panorama [30] tracks information flow at the memory operation level. It needs special hardware or the overhead would be too high. VPath [27] uses a virtual machine monitor to watch thread and network activities to detect causalities. It can provide precise information flow with low overhead if the pre-defined activity patterns are accurate. PassV2 [24] is a provenance aware storage system allowing propagation of meta data to the attached system to track information flow. Dynamic data flow tracking (DFT)/taint analysis is a class of techniques to track fine-grained propagation of relevant data [16], [14], [25] on the fly, to prevent information leak or zero-day attacks. BEEP is more light weight compared to these techniques.

Finally, BEEP can be integrated with low-overhead and trusted auditing systems [13], [28] to achieve better performance and attack resilience.

## IX. Discussion

In this section, we discuss the limitations of BEEP. First, the current implementation of BEEP has the same level of trustworthiness as the Linux audit system. Both have to trust the operating system, which makes them vulnerable to kernel level attacks. If an intruder compromises the kernel, he/she can disable the audit daemon or delete audit logs to erase evidence of the attack. However, this is not a fundamental limitation of BEEP, as BEEP can be ported to the hypervisor level and hence kept out of the reach of kernel level attacks.

Second, a remote attacker may intrude the system via some non-kernel level attacks and acquire the privileges to tamper with the binaries instrumented by BEEP. In this case, BEEP will be able to at least precisely record the initial intrusion. All events after the initial intrusion should be marked as untrusted.

Third, a legal user of the system with BEEP installed may try to attack BEEP. However, unless the user has the privileges to tamper with the instrumented executables, all he/she can do is to generate bogus events such as fake KILL signals using his/her own programs. Such behavior can be detected as each event is tagged with the process id and hence BEEP knows these signals are from an executable that has not been instrumented.

Fourth, while BEEP has preprocessed a set of commonly used applications, a user can choose to install new applications. In this scenario, since the applications have not been trained and instrumented by BEEP, unit-level logging cannot be achieved and BEEP has to fallback to process level logging provided by the audit system. Consequently, we may coalesce multiple units into one and lose percision. However, the precision of the already instrumented applications is not affected. We anticipate BEEP will constantly remind the user that he/she should train the newly installed applications as soon as possible.

Fifth, BEEP still requires user involvement. BEEP asks the user to provide regular use cases as training inputs. The user is also supposed to apply BEEP to each long running application in his/her system. In the future, we anticipate the whole process can be fully automated by logging and analyzing regular system execution automatically.

Finally, BEEP is not capable of processing obfuscated binaries due to the difficulty of binary instrumentation. For such programs, BEEP will treat the whole process as a unit and degrade to a regular audit system.

## X. Conclusion

We develop BEEP, a highly accurate attack provenance tracing technique enabled by a novel selective fine-grained logging method. It partitions a long running process to multiple autonomous units that handle independent input data. Such units are delimited by iterations of high level input event dispatch loops. BEEP automatically reverse-engineers such loops and the instructions that can cause workflows between units. It then instruments the binary to log unit boundaries and workflow dependences. Together with the regular audit log, BEEP-generated log enables identifying precise causality between a root cause (i.e. an attack) and its symptoms, avoiding the dependence explosion problem with regular audit logs. We demonstrate that BEEP achieves this with negligible runtime overhead and low space overhead.

## XI. Acknowledgment

We would like to thank our shepherd, Zhichun Li, and the anonymous reviewers for their insightful comments. This research has been supported by DARPA under Contract 12011593. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of DARPA.

## References

[1] http://www.backtrack-linux.org/.

[2] http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?name=backdoor/linux/blackhole.

[3] http://www.webkit.org/perf/sunspider/sunspider.html.

[4] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering.*

[5] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In ISSTA'11.

[6] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In OSDI'04.

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI'08.

[8] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In SOSP'11.

[9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In SSYM'04.

[10] M. Fredrikson, M. Christodorescu, J. Giffin, and S. Jhas. A declarative framework for intrusion analysis. In Advances in Information Security. Springer US, 2010.

[11] A. Goel, W.-c. Feng, W.-c. Feng, and D. Maier. Automatic high-performance reconstruction and recovery. *Computer Network, Volume 7, 2007*

[12] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In SOSP'05.

[13] M. Jakobsson and A. Juels. Server-side detection of malware infection. In NSPW'09.

[14] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for effciently accelerating software-based dynamicdata flow tracking on commodity hardware. In NSDI'12.

[15] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M. Wang. Provenance-aware tracing ofworm break-in and contaminations: A process coloring approach. In ICDCS'06.

[16] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In VEE'12.

[17] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In OSDI'10.

[18] S. T. King and P. M. Chen. Backtracking intrusions. In SOSP'03.

[19] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In NDSS'05.

[20] S. Krishnan, K. Z. Snow, and F. Monrose. Trail of bytes: efficient support for forensic analysis. In CCS'10.

[21] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In ISPASS'10.

[22] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward generating reducible replay logs. In PLDI'11.

[23] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In PLDI'05.

[24] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. In USENIX Annual technical conference, 2009.

[25] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In NDSS'05.

[26] S. Sitaraman and S. Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In IWIA'05.

[27] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. vpath: precise discovery of request processing paths from black-box observations of thread and network activities. In USENIX Annual technical conference, 2009.

[28] A. Vasudevan, N. Qu, and A. Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In HICSS'11.

[29] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In CCS'09.

[30] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In CCS'07.

[31] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. In DSN'03.