# LPROV: Practical Library-aware Provenance Tracing

Fei Wang
Purdue University
feiwang@purdue.edu

Yonghwi Kwon
University of Virginia
yongkwon@virginia.edu

Shiqing Ma
Purdue University
ma229@purdue.edu

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Dongyan Xu
Purdue University
dxu@cs.purdue.edu

## ABSTRACT

With the continuing evolution of sophisticated APT attacks, provenance tracking is becoming an important technique for efficient attack investigation in enterprise networks. Most of existing provenance techniques are operating on system event auditing that discloses dependence relationships by scrutinizing syscall traces. Unfortunately, such auditing-based provenance is not able to track the causality of another important dimension in provenance, the shared libraries. Different from other data-only system entities like files and sockets, dynamic libraries are linked at runtime and may get executed, which poses new challenges in provenance tracking. For example, library provenance cannot be tracked by syscalls and mapping; whether a library function is called and how it is called within an execution context is invisible at syscall level; linking a library does not promise their execution at runtime. Addressing these challenges is critical to tracking sophisticated attacks leveraging libraries. In this paper, to facilitate fine-grained investigation inside the execution of library binaries, we develop LPROV, a novel provenance tracking system which combines library tracing and syscall tracing. Upon a syscall, LPROV identifies the library calls together with the stack which induces it so that the library execution provenance can be accurately revealed. Our evaluation shows that LPROV can precisely identify attack provenance involving libraries, including malicious library attack and library vulnerability exploitation, while syscall-based provenance tools fail to identify. It only incurs 7.0% (in geometric mean) runtime overhead and consumes 3 times less storage space of a state-of-the-art provenance tool.

**ACM Reference Format:**
Fei Wang, Yonghwi Kwon, Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2018. LPROV: Practical Library-aware Provenance Tracing. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274694.3274751

## 1 INTRODUCTION

APT (advanced persistent threat) attacks are eternal enemies to cybersecurity communities and contemporary enterprise networks are suffering the most among all the network environments. Incented by tremendous economic interests in commercial espionage, attackers are taking persistent efforts in penetrating enterprise networks from diverse vectors, which motivates the increasing demands in cyber attack investigation. Detecting or intercepting an APT attack at its entry point is particularly challenging due to their advanced and stealthy attack techniques. For example, backdoor implantation scheme allows attackers to inject malicious code into a benign program in order to disguise the malicious behaviors as normal benign behaviors. Rather than downloading and executing obvious malicious programs, they leverage existing benign applications and services to conduct malicious behaviors such as downloading or opening attachments from well social-engineered emails, and clicking URL links in luring advertisements [44]. Hence, in recent years, in addition to the significant contribution manifested in attack detection, provenance tracking becomes an irreplaceable pillar in APT analysis and defense. Given a target system entity or object (e.g., compromised file, socket, or process), provenance tracking systems analyze it from multiple aspects, and figure out the entity's root (or origin) as well as deriving path [32, 44, 47, 48]. The root contains all the external entities (e.g., an IP address) affecting the status or value of the target entity; while the deriving path is an organized causal graph illustrating how the entity is eventually influenced from the root. Such tracking information can facilitate locating the attack and prevent repeated infection.

Existing provenance tracking techniques can be divided into three categories: non-unit provenance, tainting-based provenance, and unit-based provenance. Most provenance tools leverage event logging to trace system events (e.g., syscalls) and associate them for further offline attack tracking and investigation [28, 33, 34, 39, 46, 51, 53, 55]. While there are various system events such as network communication, memory operations and syscalls, most provenance tools focus on logging and analyzing syscall events as syscall logging (e.g., audit logging) is widely used (included in most Linux distributions by default) and practical (low overhead).

**Non-unit provenance techniques** Non-unit provenance tools have a conservative assumption: a process is causally related to all the system entities (e.g., files and sockets) it has accessed so far. In such conservative causality correlating models, any output object (e.g., files and sockets) of a process has causal relations to all the preceding input objects, resulting in many bogus causality relations and confusing the following attack investigation. We call such problem the dependence explosion problem [28, 38, 40, 44] and the problem becomes particularly severe in complex long-running programs such as *firefox* and *Apache*. For instance, consider a case


*ACSAC '18, December 3–7, 2018, San Juan, PR, USA*

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274751

where a user opens *firefox* and browses 10 websites. Then, the user carelessly downloads a malware binary on one of the website, namely *xxx.com*. Later, the malware is detected and an investigator wants to identify which website downloads the malware. When non-unit provenance techniques are used, they report all of the 10 websites as roots while only the website *xxx.com* is the true root cause and other 9 websites are not.

**Tainting-based provenance** Tainting-based provenance techniques [19, 22, 32, 35, 37, 49, 52, 54, 56, 67] assign tags to multiple tainting sources (e.g., receiving sockets and input files) and propagate them by monitoring executed instructions. When those tags reach sinks (e.g., sending sockets and output files), they detect information flow between sources and sinks, revealing the roots of the sink entities (e.g., IP addresses). However, since these techniques work on the instruction level, most of them cause significant runtime overhead and they are rarely used in production runs. Note that while the latest instruction-level tracking system may perform replay-based provenance analysis in low overhead (3.22%), its resource pruning and selective tainting in replay require pre-recorded instruction-level execution logs [32], hence it is not applicable in our context. Moreover, taint analysis suffers from over/under-approximation as it has difficulty handling implicit information flow through control dependencies (e.g., data compression and table lookup). Besides, taint set operations in instructions are error-prone since pointers, arrays, syscalls, third-party libraries and language-specific features should be carefully processed.

**Unit provenance techniques.** Unit-based provenance such as BEEP [44, 45] and Protracer [48] is the state-of-the-art in provenance tracking. They are based on an observation from a study [44]: in diverse open-source software including both of client-side and server-side, most programs are designed in input-triggered loops which dominate the event handling. Hence, unit-based provenance tracking techniques profile such loops from program binaries and partition programs' execution into units. Those units are semantically autonomous and they are usually responsible for independent input events. By partitioning a long-running execution into multiple small execution units, they significantly mitigate the dependence explosion problem. An output object is considered causally correlated to an input object only if they belong to the same unit. However, a single unit may not cover the whole execution subroutine for one input event, such as the asynchronous unit cooperation in message queue processing. Hence, memory dependency between units is also tracked to detect inter-unit dependencies. With such design, the unit-based provenance tracking techniques can accurately identify the root (the *xxx.com* website in the previous example) pruning out other bogus dependencies.

Unfortunately, while the unit-based provenance techniques mitigate the dependence explosion problem, they fail to consider another important aspect of provenance, the shared libraries. An integrated executable binary consists of a main module and several depending shared libraries. For example, the binary of *vim* links 14 shared libraries and *firefox* has more than 20.In most of provenance tracking systems, the main module and those libraries are usually handled as one process/program entity but they are not analyzed separately. Since the library loading phase is in the program initialization but outside any event handling loop, the output of any unit has no dependence on any input library in unit-based tracking techniques. As a result, the unit-based techniques are not able to track provenance in libraries. Note that although Protracer [48] correlates all the loaded libraries in generating the causal graph, it is still coarse-grained since they are the same input for all the units, which is considered no causality in the logic of dependence analysis. Unlike other input sources such as files and sockets which are value-based hence can be tracked by monitoring I/O syscalls, the provenance inside libraries is execution-based and the correlation cannot be simply tracked by causality deduction in I/O syscalls. Specifically, mapping or linking a library does not promise any execution instance, and whether a specific library is executed or how it is executed within a unit cannot be answered in the syscall granularity. Note that there have been proposed user-space tracing tools to perform provenance tracking [16, 58, 63], but they are too coarse-grained and the tracing requires repeated manual efforts in event/causality definition or program instrumentation because stealthy attacks could act as normal behaviors (See Section 2).

**Our solution** To this end, we develop a user-space library tracing technique and merge it into existing syscall-based provenance tracking systems in order to improve the visibility of library execution in provenance tracking. We propose a novel provenance tracking system LPROV which performs on the granularity of library functions other than those on syscalls. It aims at addressing the obstinate wart, the absence of user-space library provenance, in syscall-based auditing systems.

It works as follows. Upon the beginning of a program execution, LPROV is loaded into process memory by a customized loader. It records the entrance and exit of library calls by manipulating symbol tables and maintains library call stacks for each thread. To be integrated with the audit logging techniques, LPROV also deploys a kernel module to collect syscall events. Only when trapped into a syscall, a process's library call stack is retrieved for output. To ensure the efficient processing in kernel, a daemon process in user space is designed to take over log delivery and optimization (e.g., reduction of redundant or duplicate logs). During production runs, when a syscall is made, its deriving path from the library perspective is disclosed by the library call stack on causality correlations.

Our contributions are summarized as follows.

- We propose an efficient provenance tracking system LPROV, combining library tracing in user space and syscall tracing in Kernel space. Whenever a provenance-related syscall is made from a thread, its full library-level execution path is also unveiled. Equipped with the library provenance, causality is revealed not only between explicit value-based input and output system entities but also inside the implicit fine-grained execution-based shared libraries.
- We devise a lightweight and efficient system-wide library tracing infrastructure. The tracing provides a friendly running environment for heavy-threaded programs and all the thread properties (e.g., concurrency) are well preserved.
- We evaluate our prototype and the results are promising. LPROV can precisely identify the provenance in malicious library, and it incurs only 7.0% runtime overhead (in geometric mean) and consumes 3 times less storage space (29.7% of the space for provenance data by BEEP [44]).

## 2 MOTIVATING EXAMPLE

The Linux Ebury attack in ssh service [10] motivates the importance of library aware provenance tracking. This attack leverages a stealthy backdoor to implant subsequent malicious binaries such as ssh clients or servers. The first version of Ebury attempts to replace ssh-related binaries such as *sshd*, *ssh* and *scp* by carefully crafted malicious binaries. However, the crafted programs are too obvious and attack ramification could be easily exposed to existing provenance tracking systems. Hence, Ebury evolves into exploiting well-camouflaged shared libraries rather than directly intruding the program bodies. In this paper, we reproduce a version of the library-base Ebury attack to show the effectiveness of Lprov.

**Library-based Ebury** To make the attack stealthy, library-based Ebury carefully chooses a particular library, *libkeyutils.so*, which is one of the libraries for Kerberos authentication. Specifically, Kerberos authentication is a widely used identity authentication protocol between ssh clients and servers and most Linux versions support it by default. In Linux, it is implemented by 4 libraries, *libkcrypto.so*, *libkrb.so*, *libkrbsupport.so* and *libgssapi_krb.so*. Among these libraries, the key management library *libkeyutils.so* is only called by `krb_get_notification_message` in *libkrb.so* which is, in fact, never called in the current Kerberos implementation. In other words, *libkeyutils.so* is a "dangling" library in ssh programs. Moreover, we observe that in most Linux versions, no other programs are using *libkeyutils.so* except the ssh service. Hence, the attacker chooses *libkeyutils.so* as it would only affect the ssh program without attracting attentions from users and security administrators.

**Attack Scenario** This is an exfiltration attack that aims at stealing users' private keys. An administrator is maintaining several servers and he generates public/private key pairs (one pair for one server or one pair for multiple servers) for remote login, which is protected by the ssh public key authentication. Considering the flexibility of server configuration, such as location changing, service switching and load balancing, those servers are managed by some dynamic domain name service (e.g., No-IP) where IP addresses are not fixed. Through an unverified package update, the *libkeyutils.so* library in administrator's laptop is replaced by a malicious one containing a backdoor in the library's *constructor*, transferring hosts' private keys to a remote attacker-controlled site `y.y.y.y`. Note that it is possible to make the program load the malicious library without physically replacing the library file. Specifically, the library can be placed into directories with higher search priority in the loading phase, without changing the original library file. After a few months, the administrator logs into a server and notices that the system has been compromised. He then realizes that his private key was leaked since the attacker successfully got through the login authentication.

**Provenance Analysis** The causal graphs tracking from the private key file `id_rsa.1` generated by BEEP [44] and Lprov are in Fig. 1. Note that the graph contains about thirty different remote server nodes and many file object nodes but we only analyze a small part of it (related to the provenance analysis). Since the ssh client is not a long-running program with an input handling loop, the unit partition of BEEP cannot take any effect here but all the objects are dependent on the whole process. Hence the network connection established for exfiltration in the library's *constructor* is causally correlated to *ssh* in the same way as other remote servers. When a user accesses any remote server through the ssh client, the corresponding public/private key is always read for identity authentication. Therefore, without extra evidence, it is impossible to differentiate the attacker's site `y.y.y.y` from other server addresses (e.g., `x.x.x.x`, `z.z.z.z`, and so on). Accomplishing the file transmission, the library's *constructor* outputs connection error messages and calls `exit` to terminate. Therefore, the ssh client would not connect to any server after the communication with `y.y.y.y`, preventing the attack being readily caught in the light of a suspicious ssh process associated with two different remote servers. From the administrator's perspective, the program termination is just regarded as a normal connection failure and no anomalies could be perceived. Moreover, the attack is conditionally triggered with a certain probability, hence most ssh connections launched from the client are benign ones. If the administrator is fortunate enough to obtain the whole and correct server-side logs such as login and DHCP, the suspicious connection to `y.y.y.y` could be identified. However, it is challenging to understand the root cause and internal profound details of the malicious library file.

In contrast to BEEP, Lprov's causal graph shows the execution-based provenance for the attack. In this clear context of execution paths, we can figure out that the private key was exfiltrated to the remote server `y.y.y.y` through the *constructor* function of *libkeyutils.so* in trivial efforts. Note that to simplify and clean the illustration, Figure 1 omits the provenance of the library file, but this will be detailed in Section 5.2.1.

## 3 SYSTEM OVERVIEW

Lprov leverages the same program unit instrumentation scheme from BEEP and ProTracer [44, 48], assuaging the concerns of dependence explosion. Fig. 2 illustrates the architecture of Lprov. Specifically, the customized loader mandatorily preloads the tracing library *lprov.so* into processes' memory at the program bootstrap and monitors the initialization procedure of libraries upon loading. The library *lprov.so* takes charge of tracing and storing library call events into a memory chunk shared with kernel. The kernel module and the user-space daemon are largely inherited from ProTracer [48] and we augment them to accommodate library-level events, but our contribution mainly lies in improving the library awareness of auditing-based provenance tracking by efficient library tracing. Our kernel module is responsible for (1) recording syscalls and (2) copying associated library call stacks into a circular buffer shared with the daemon process. The daemon pulls log entities from the shared buffer and outputs them to the log file for further provenance analysis. Note that to minimize the attack surface (i.e., to prevent hijacking from compromised libraries), external library codes are statically linked to the customized loader, *lprov.so* and the user-space daemon.

A shared library could be loaded through process initialization (by system loader) or calling `dlopen` on demand. In either way, when a library is loaded, *lprov.so* alters the offsets of exported function symbols in the library ELF header and redirects them to injected wrappers (i.e., entrance wrappers). In the entrance wrapper routine, the return address of a library call is then instrumented with another wrapper (i.e., exit wrapper) to catch function exit. During program execution, the two wrappers record the enter and exit sequences of

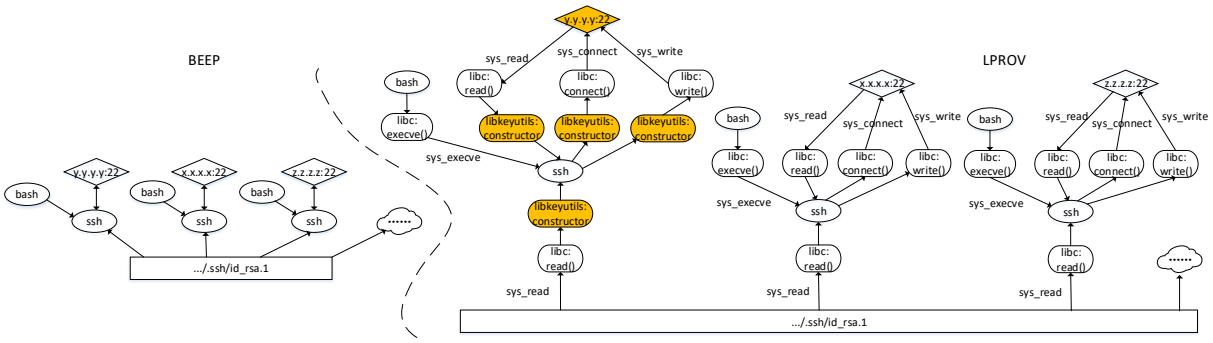


**Figure 1: Causal graphs generated in provenance tracking for Ebury exfiltration attack. BEEP can not distinguish the attacker address but LPROV gives a clear attack context.**

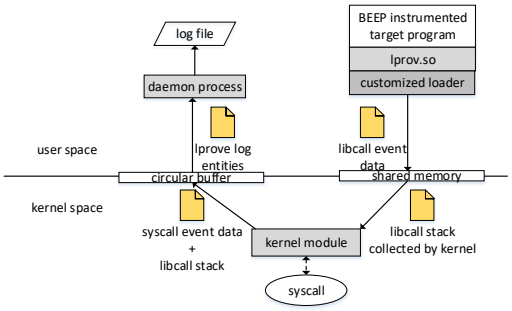


**Figure 2: The architecture of LPROV: dashed lines denote control flow and solid lines denote data flow.**

library calls to the shared buffer which is read by kernel later. Note that such a library tracing procedure hinges on dynamic symbol resolution and its limitations will be discussed in Section 6. Details of library call tracing are discussed in Section 4.1. Compared with syscalls involving low-level kernel object operations, user-space library functions are called much more frequently. For instance, more than 18.2 million library functions are executed when a user opens *firefox*, visits the homepage of *New York Times*, and clicks the headline. Meanwhile, only about 753 thousand syscalls are invoked. However, many of those library function calls are less significant (e.g. `tolower, toupper`) for provenance analysis. Hence, instead of recording all the call sites of library functions, LPROV focuses on a small fraction of important library calls (e.g., `read, write, constructor`) and builds a concise, yet comprehensive (regarding provenance tracking) library call graph. Only if a library call terminates at a syscall, its full call stack is considered for output. Other cases which do not interact with kernel are pruned . The design of kernel module is elaborated in Section 4.2.

The daemon fetches log data from the circular buffer as a consumer. A log entity is a syscall event annotated by a library call stack. Among candidate entities, the daemon only outputs the ones which are not duplicating any already-output log record. This part is dissected in Section 4.3.

**Assumption** In this paper, like other existing provenance systems built on audit logging, we trust the kernel and the user-space processing daemon. Apart from that, LPROV also assumes the integrity of the library tracing component in user space. The limitation of such assumptions will be discussed in Section 4.1.3 and Section 6.

# 4 DESIGN AND IMPLEMENTATION

## 4.1 Library Call Tracing

In this section, we discuss the detailed design of LPROV and compare it with *ltrace* (the most widely used library call tracing tool in Linux).

*4.1.1 Design.* The tracing functionality of LPROV is encapsulated as a library *lprov.so* loaded into the runtime memory of the tracee. It realizes logging library calls with three cooperative components: dynamic symbol redirector, entrance/exit wrapper and customized loader.

**Dynamic Symbol Redirector** This component serves as the pillar of *lprov.so*. It is composed of an overrider of `dlopen` and a constructor function (of *lprov.so*).

When importing a module, the loader typically runs in a lazy binding mode, where the resolution of external function symbols is not executed until the program reaches their first reference. The system resolver relies on the symbol table of library's exported functions to parse entrance addresses of external functions.

Hence, the overrider manipulates specific fields of the table to direct the resolution results to our injected wrapper routines (introduced in the next two paragraphs). Specifically, the overrider first calls the original `dlopen` and obtains the library handle. Then, it locates the addresses of dynamic function symbols (i.e., `DYNSYM` section which contains the list of exported functions). Finally, the attributes of `st_value` and `st_info` inside `Elf_Sym` structure are modified for each symbol. The `st_value` attribute of a dynamic function symbol implies the offset of the function's entrance to the library handle. We change the values of `st_value` in order to interpose the functions.

However, if a library function is compiled as `IFUNC` whose implementation is bound during runtime, changing the value of `st_value` will cause the program to crash when the function is called since `st_value` of an indirect function indicates the address of the runtime resolver instead of the function entrance. Since these functions are specified as `IFUNC` and the types are stored in `st_info`, we also alter this value to make the system resolver handle indirect functions as non-indirect ones. In addition, the original address of an indirect function could be pre-fetched by `dlsym`. Hence, the constructor function of *lprov.so* interposes `dlopen` on all the libraries (including dependent libraries) in the program's ELF header, while the libraries loaded by `dlopen` during runtime are automatically instrumented by the overrider.

**Entrance/Exit Wrapper** The manipulated symbol table dispatches library calls to the corresponding function entrance wrappers. A library function's entrance routine (1) stores functions' original return addresses (stored at `(%esp)`) and stores programs' states, (2) logs library call entrance events, (3) redirects return addresses to exit wrappers, and (4) restores programs' states and resumes library function execution.

An exit wrapper (1) fetches functions' original address and storing programs' states and (2) pops out the last library call entrance events and resumes library function returns.

**Customized Loader** To guarantee that the library call tracing functionality is deployed before the target program executes any library function, the library *lprov.so* must be loaded before all others. In Linux, generally, this could be done by enabling the `LD_PRELOAD` environment variable. However, many anti-debugging and self-protecting techniques probe `LD_PRELOAD` and then refuse to execute when the variable is used. Moreover, according to Linux security policy, `LD_PRELOAD` referring to any untrusted path is ignored in loading phase when binaries' `setuid` or `setgid` bit is set. To this end, rather than using the `LD_PRELOAD`, we modify the loader to enforce the pre-loading of *lprov.so* anyway through `do_preload`.

Apart from all the exported functions, a library's constructor plays a critical role in the library loading mechanism. It is called by the loader upon the library importing procedure, before the library handle is actually returned. Therefore, we customize the loader further to monitor the execution of libraries' constructors. Specifically, in the loader's initializer `_dl_init`, at the execution points where `call_init` is called and returns, the constructor entrance event is pushed and popped as regular library calls.

*4.1.2 Design Choices.* In this section, we elaborate our design choices comparing with the design choices of *ltrace*, and highlight advantages of Lprov. In particular, *ltrace* leverages breakpoints and debugging mode in order to interpose library calls, which results in high overhead and missing library call events. In contrast, Lprov uses an dynamic instrumentation approach which outperforms *ltrace* in various aspects. In the following, we elaborate the advantages from five different perspectives.

**Interposing Nested Library Calls** To interpose library calls, *ltrace* leverages `ptrace` which imposes high overhead. In addition, *ltrace* inserts software breakpoints (`INT 3`) at the PLT (Procedure Linkage Table) trampoline entries of library functions. Note that software breakpoints incur significant overhead as the entire process is stopped when a breakpoint is reached. More importantly, the resolutions of external function symbols in different libraries (modules) are independent and they all hold their local PLTs. The breakpoint insertion of *ltrace* is confined within the PLT trampoline entries of the program (the main module) but all the libraries' PLT segments are not instrumented. Hence, if library calls are nested, *ltrace* is unable to trace the inner ones.

| Unlike *ltrace*, Lprov focuses on the symbol resolution phase inside the loader. Hence, it can trace nested library calls with significantly lower overhead compared with *ltrace*. |
|---|

**Interposing Non-PLT Library Calls** Functions in a library can be called through indirect calls using function pointers. Such calls do not go through PLT trampolines hence *ltrace* is not able to interpose such library calls[1].

| Since Lprov operates through dynamic symbol tables, all library calls by function pointers derived during symbol resolution can be traced in Lprov. |
|---|

**Constructor Function** While constructor function initializes variables and states, it can also execute any code upon library loading. Hence, without the awareness of constructor function, distinguishing library calls inside it from the ones outside is particularly challenging. Unfortunately, *ltrace* is not capable of tracing the constructor functions as it focuses on PLT trampolines.

| The customized loader of Lprov makes sure that Lprov traces all the constructor functions as well as library function calls within the constructor functions. |
|---|

**Thread Support** *ltrace* leverages software breakpoints (i.e., `INT 3`) to interpose library calls. Unfortunately, when a program halts on `INT 3`, it enters a SIGTRAP status where all the threads are suspended. In addition, to be informed of function returns, *ltrace* inserts breakpoints at the return addresses, which would trap the execution of other threads into unexpected breakpoint handling routine when text segments are shared in a thread group. As a result, *ltrace* hurts thread concurrency, eventually incurring additional overhead in multi-threaded applications.

| As Lprov does not rely on software breakpoints, it does not have any of the aforementioned issues. It offers a thread-friendly tracing environment, where thread concurrency properties are preserved well. |
|---|

**Runtime Overhead** Compared to Lprov, the breakpoint scheme of *ltrace* incurs much higher overhead due to context switch and the limited thread support. Table 1 shows the runtime overhead of the two tracing tools in the same workloads of four server-side programs (*httpd, simplehttpd, proftpd, sshd*), eight client-side programs (*firefox, filezilla, lynx, links, w3m, wget, ssh, pine*) and three editors/readers (*vim, emacs, xpdf*). The Apache Benchmark [1] tool is used to measure the two web service programs, *Apache httpd* and *simplehttpd*, and *ftpbench* [7] is used to measure *proftpd*. For *firefox*, we use the standard browser benchmarking tool *SunSpider* [12], and we use corresponding program scripts for all the other programs. In the columns of *ltrace*, `N/A` in *httpd* and *firefox* indicates that programs do not terminate in a reasonable time limit (e.g., meaning that it incurs more than 1000% overhead) or just crash. This is because *ltrace* incurs particularly high overhead in multi-threaded programs. Note that to measure the performance of *ltrace*, we develop a simple library hooking module that leverages the same techniques used in Lprov without any additional provenance related components. Specifically, it only records entrances and exits of library calls.

| The primary results show that Lprov outperforms *ltrace*. The overhead incurred by *ltrace* is 10-20 times more than that of Lprov. Moreover, Lprov can trace more library calls as *ltrace* is not capable of tracing nested and Non-PLT library calls. Observe there exists a gap between the amounts of recorded library calls. |
|---|

[1] https://linux.die.net/man/1/ltrace, the latest document claims that *ltrace* can handle the `dlopen` case but the latest software version 0.7.3 still does not have this feature.

**Table 1: LPROV has much lower runtime overhead than *ltrace***

| Program | ltrace | | LPROV | | #Gap |
|---|---|---|---|---|---|
| | # of calls | overhead | # of calls | overhead | |
| httpd | N/A | N/A | 8764.7K | 53% | N/A |
| simplehttpd | 946.4K | 563% | 965.5K | 28% | 19.1K |
| proftpd | 1407.8K | 622% | 1494.3K | 36% | 86.5K |
| sshd | 4987.0K | 769% | 5019.8K | 44% | 32.8K |
| ssh | 3668.6K | 813% | 3741.6K | 30% | 73.0K |
| firefox | N/A | N/A | 394461.6K | 126% | N/A |
| filezilla | 7311.5K | 988% | 7383.2K | 47% | 71.7K |
| lynx | 912.6K | 672% | 933.0K | 34% | 20.4K |
| links | 659.7K | 565% | 697.8K | 29% | 38.1K |
| w3m | 757.2K | 622% | 785.3K | 41% | 28.1K |
| wget | 448.8K | 390% | 453.3K | 30% | 4.5K |
| pine | 1092.8K | 522% | 1103.6K | 34% | 10.8K |
| vim | 8276.0K | 734% | 8336.9K | 37% | 60.9K |
| emacs | 3053.2K | 658% | 3104.4K | 31% | 51.2K |
| xpdf | 781.5K | 742% | 803.0K | 40% | 21.5K |

*4.1.3 Data Integrity.* As malicious libraries and recorded library call stacks reside in the same memory space, the tracing data might be compromised during runtime. To mitigate the issues, we can leverage complementary address space (re)randomization techniques [18, 29, 61] such as ASLR (Address Space Layout Randomization). Specifically, randomization techniques make sure that the library of LPROV is loaded into a random address. Moreover, we can leverage these techniques to randomize addresses of our data structures. Also, hardware features can be leveraged to ensure data integrity [41] as well.

## 4.2 LPROV Kernel Module

**Tracing Target** We leverage ProTracer's [48] kernel event tracing framework built on Tracepoints [13] to monitor syscall instances. The details of tool selection for kernel event tracing are discussed in Appendix A.1. For provenance purpose, only the syscalls related to explicit causality deduction are considered in kernel tracing. The set of tracing targets is the same as Protracer [48], including syscalls on basic file read/write operations, file redirection syscalls, IPC syscalls, process management syscalls and customized syscalls to mark the unit in/outs. In particular, LPROV enlarges the set by another category of syscalls, memory permission manipulating syscalls. Change of memory access right is a critical clue for provenance inference, especially in user-space library tracing. For example, a function in library *A* alters the permission of pages allocated to library *B* and modifies the image of *B* dynamically. BEEP and Protracer are oblivious on such dependence because the inter-unit memory read event defined in them is an explicit value-based read that cannot model an implicit *execution-based read*. Hence, LPROV additionally collects syscall events of `mprotect`[2] and `mmap`. When library *A* invokes `mprotect` to set the `PROT_WRITE` or `PROT_EXEC` flag to a memory chunk mapped to library *B*, then the corresponding memory section of *B* has runtime dependence on *A*. In addition, syscall `mmap` can be utilized to obtain a memory chunk with specific permissions and this is the default way library binaries are loaded into process memory by the program loader. Library tracing is accomplished by *lprov.so*, and hence it is unnecessary to additionally handle the general library mapping here. Specially, if files (including libraries) are non-anonymously mapped, they will be considered either input or output object of the process according to the permissions and opening modes of file descriptors. For all the other anonymously (i.e. `MAP_ANONYMOUS`) mapped executable pages, we regard them as a part of the main module and then the main module is considered dynamically dependent on the `mmap` invoker.

**Event Collecting** For a syscall instance, in addition to the standard syscall event which is recorded by existing systems built on audit logging, LPROV also correlates the thread's library execution path to this event. The kernel module retrieves library call stacks from the buffer shared with user-space applications, packs syscall events with the stacks into log entries and delegates the log outputting task to the user-space daemon through the circular buffer. By doing so, kernel does not need to wait for the previous events to be completely processed. The size of the buffer can be also configured so that kernel would not wait when the buffer is full. It can be also configured to drop events when the buffer is full. To accelerate the library call processing inside kernel, the buffer maintaining programs' library call stack is a per-thread buffer indexed by thread id so that kernel can access the memory efficiently. To prevent expensive dynamic memory management from weighing in, we choose to pre-map the per-thread buffer with a fixed size when the kernel module is loaded.

## 4.3 LPROV Daemon Process and Log Analysis

**Daemon Process** The daemon keeps reading log events from kernel through the producer-consumer buffer. Since there are only one producer (producer does not overwrite data) and one consumer, the circular buffer is implemented in lock-free mode. Plumbed from Protracer [48], the daemon marshals a thread array to perform log processing and logs from one process should be tackled by only one thread to maintain processes' consistent and complete execution context. The on-the-fly log processing phase aims at reducing duplicate events on the same system object within one unit. For example, downloading of a large file inside browsers is fulfilled by thousands of socket-reading and file-writing operations, but only one of them needs to be recorded for provenance purpose. In addition, the processing also connects redirected files (dup function group) to prevent causality loss. Note that we do not apply the log reduction scheme playing with taint propagation in Protracer [48] since the taint cannot clarify the implicit execution path from the source to target hence not applicable to our purpose. Moreover, if we combine LPROV with the tainting technique, a taint must be spawned for each library call stack per syscall event, which does not take any advantage over the direct logging. Eventually, the filtered log entities are delegated to the log outputting thread to generate the log file on disk.

**Log Analysis** LPROV provenance tracking is expected to answer how a system object is affected and how it affects the system in the perspective of syscall and library events. Hence, we provide both of backward and forward tracking, disclosing the deriving path and the aftermath of objects in a directed provenance graph. The log analysis algorithm is similar to that in BEEP [44] and Protracer [48] but LPROV augments the provenance graph by handling

[2]The more efficient syscall `pkey_mprotect` is not included here since it is not supported until Linux-4.9 kernel and it also requires specific hardware assistance.

additional events on memory permission manipulation and capturing library-level execution causalities. The algorithm (Algorithm 1) is elaborated in Appendix A.2.

## 5 EVALUATION

We set up four machines (machine A, B, C, and D) with similar hardware configuration (16GB RAM and Intel i7 CPU) in our evaluation experiments. In order to compare the overall performance, those machines are all deployed LPROV and BEEP.

We select 11 programs from Table 1 for measurement as some of them share the same functionalities. Not all of them are instrumented by BEEP as programs like *ssh* do not have an event handling loop and they are not designed to run for a long time.

We assign one machine (i.e., machine A) as the server (running both server-side and client-side programs) and the other three (e.g., machine B, C, and D) as pure client machines (running client-side programs only). We refer the anonymous users of these machines with the same alphabet letters (User A, B, C, and D).

User A configures, manages and maintains the server service. The server operates a tiny web server for a group project, an FTP server for file sharing and an SSH server for remote accessing. The other three users (Users B, C, and D) are required to actively communicate with the designated server and use the selected programs during the experiment. Besides the necessary communication with the server, the three users also have their own behavior profiles. User B mainly uses the machine to watch TV or movies online, visit social network sites, browse news and chat with friends. User C undertakes most of his project coding work on the assigned machine. He usually downloads and reads programming manuals or documents. User D is preparing some coming interviews. She mainly accesses her email account for personal communication, watches presentation videos and visits Q&A websites to collect interview-related materials.

### 5.1 Performance Overhead

**Storage Overhead** The performance experiment lasted two weeks and the results are shown in Table 2. From the results, we can observe that in spite of logging the library call stack, the storage consumption of LPROV is only 29.7% of BEEP. Since LPROV applies a on-the-fly reduction phase, the generated log has much less redundancy compared to the original audit event log. Note that because LPROV logs additional library-level events and the tainting scheme in Protracer [48] is not applicable in our context, it appears to be unavoidable that LPROV has greater storage overhead than Protracer (See additional performance evaluation in Appendix A.3).

**Runtime Overhead** As shown in Figure 3, while LPROV logs more user-space library information, its runtime performance is still competitive and acceptable. Specifically, the geometric mean and arithmetic mean of the programs' overhead in LPROV are 7.0% and 8.6%, compared to 8.4% and 9.8% in BEEP, 4.5% and 5.3% in ProTracer. We notice that the programs' performance in Figure 3 achieves significant improvement from Table 1. This is because the daemon process decouples the onerous log outputting task from the library tracing component. Note that in runtime overhead benchmarking, instead of imposing intensive and bursting tasks, we apply day-long program workloads from general use cases which we

**Table 2: Comparison of storage overhead between LPROV and BEEP in a two-week performance experiment**

| User | BEEP | | LPROV | | LPROV/BEEP | |
|---|---|---|---|---|---|---|
| | #item(M) | size(GB) | #item(M) | size(GB) | item | size |
| A | 139 | 116 | 59 | 45 | 42.4% | 38.8% |
| B | 185 | 146 | 62 | 41 | 33.5% | 28.1% |
| C | 119 | 91 | 42 | 26 | 35.3% | 28.6% |
| D | 109 | 90 | 33 | 20 | 30.3% | 22.2% |
| Avg. | 138 | 111 | 49 | 33 | 35.5% | 29.7% |
| Prog. | BEEP | | LPROV | | LPROV/BEEP | |
| | #item(M) | size(GB) | #item(M) | size(GB) | item | size |
| httpd | 40.9 | 32.3 | 13.9 | 8.3 | 33.9% | 25.7% |
| proftpd | 29.2 | 23.6 | 8.3 | 5.4 | 28.4% | 22.9% |
| sshd | 33.0 | 26.8 | 15.4 | 10.8 | 46.7 % | 40.3% |
| ssh | 10.5 | 8.1 | 4.1 | 2.8 | 39.0% | 34.6% |
| firefox | 207.7 | 168.9 | 68.0 | 49.6 | 32.7% | 29.4% |
| filezilla | 13.7 | 10.6 | 2.7 | 1.6 | 19.7% | 15.1% |
| w3m | 3.9 | 3.1 | 1.2 | 0.9 | 30.8% | 29.0% |
| wget | 2.6 | 2.0 | 0.8 | 0.5 | 30.8% | 25.0% |
| pine | 1.9 | 1.6 | 1.0 | 0.8 | 52.6% | 50.0% |
| vim | 26.4 | 21.0 | 15.2 | 11.0 | 57.6% | 52.4% |
| xpdf | 15.0 | 11.4 | 6.0 | 3.9 | 40.0% | 34.2% |

profiled from the four deployed machines/users. Such measurement reflects real-world scenarios where APT attacks happen (i.e., institute/organization/enterprise environment). It also reasonably amortizes the expensive GUI constructing/destroying computation at program opening/closing. There is an outlier. LPROV incurs high overhead (28.9%) on *firefox*. This is caused by the fact that *firefox* makes massive library calls (around 3 million per page loading in average). We believe that the problem could be attenuated by reducing the library logging scope as discussed in Section 6.

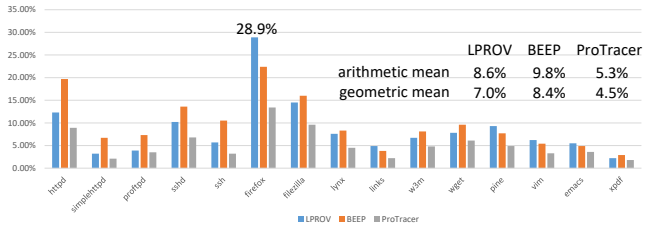


**Figure 3: Comparison of runtime overhead of BEEP, ProTracer and LPROV.**

### 5.2 Case Study

*5.2.1 Ebury Variant Attack.* We use another variant of Linux Ebury attack [10] to show the effectiveness of LPROV in provenance tracking. The attack leverages a malicious library to alter memory images of benign programs. The attacker steals a student's campus credential and leverages the credential to spread the malice. Note that universities usually assign each student one credential for all the campus resources and it would lead to serious privacy issues if the credential was stolen. Given a compromised credential on an end-host, LPROV can then assist identifying the attack source and preventing future attempts.

**Attack Scenario** Bob is a college student and he usually uses ssh service to reach lab computers on campus. He installed LPROV and BEEP on his laptop and enabled them in his daily application usage. One day, he was notified by the campus network administrator

that his account was sending spam and phishing emails. After necessary investigation, Bob proved to be innocent but his account was temporarily deactivated and he was required to use the two-factor authentication in the future. Later, he wants to know how the attack happened as he did not ever tell anyone else his credential information, type in his credential on others' machines or open any phishing links.

Note that as ssh client stores all the public keys of known hosts to verify their identities, the program should have displayed a warning message (e.g., the public key mismatch) if the attack happened on the connection to a known server. Unfortunately, Bob confirmed that he deleted the the `known_hosts` file as many websites suggest to do so (assuming that the server changed its key pair) [14]. Moreover, the latest variants of Linux Ebury can instrument the authentication part to suppress the mismatch warning [10].

**Attack Investigation** Bob did not store his student credential in any file. Hence, instead of concentrating on sensitive files (as we did in Section 2), he focused on analyzing logs from all the processes he recently used. Eventually, he obtained Fig. 4 (the graph is simplified) when dissecting an *ssh* process and the graph of Lprov contains the complete attack chain. As demonstrated in Fig. 4 (the red dash line is auxiliary for presentation), the PLT image permission of ssh process was manipulated by the constructor of *libkeyutils.so.1* and then the process connected to a remote address `b.b.b.b:22` which is a machine outside of the campus. Based on this result, we can infer that the constructor altered the process's PLT entries (probably the function path of `connect`) and redirected the connection to the attacker's server. Further, we apply backward tracking on the object of the library file. It reveals that the malicious library is downloaded by *firefox* from `a.a.a.a:43`. Then, it was implanted into a target directory through unverified package installation. Note that this attack cannot be uncovered in BEEP due to its lack of user-space library semantics. Specifically, in BEEP, even if backward tracking is performed on all the loaded libraries, the root cause of the attack would remain unexposed unless `a.a.a.a` is pre-known as malicious.

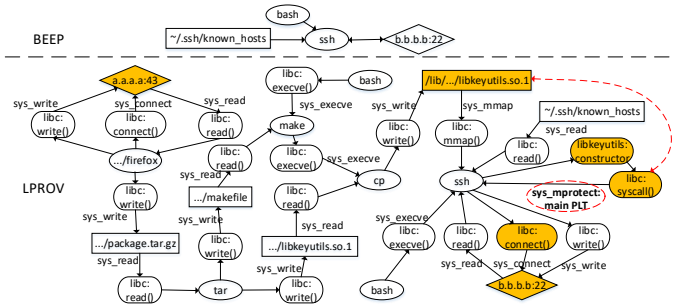


**Figure 4: Provenance graphs generated by BEEP and Lprov for the student credential stealing attack.**

*5.2.2 Library Vulnerability Exploitation.* We use an infiltration attack reproduced from CVE-2015-7547 [2–4] to show the effectiveness of Lprov in the context of remote vulnerability exploitation. There is a buffer-overflow vulnerability in *libresolv.so* that can be triggered by maliciously crafted DNS responses. Specifically, when a client calls `getaddrinfo` with `AF_UNSPEC` to resolve a domain name, a pair of IPv4 and IPv6 requests would be sent by `send_dg` and `send_vc` in *libresolv.so*. If the response is larger than the pre-allocated 2048 bytes buffer, the two functions allocate additional heap buffers and the mismanagement of the buffers leads to the exploitation. Further details can be found on [5].

**Attack Scenario** The attacker acts as a local DNS proxy and responds all the DNS requests from the victim client which conducts DNS resolution on `getaddrinfo`. When receiving requests from the victim client, the attacker sends crafted responses to implant a malicious payload [3, 4]. The payload launches a shell executing *wget* to download a file `secret.txt` that replaces the current secret file under the victim's home directory and then exits. Several days later, the file was found compromised.

**Attack Investigation** In Fig. 5, BEEP fails to obtain the origin of the attack rooting from the compromised library, while Lprov successfully captures the execution path of the exploitation when tracking from `secret.txt` (`name4_r`, `nsearch` and `sndmmsg` are shorts for `_nss_dns_gethostbyname4_r`, `__libc_res_nsearch` and `__sendmmsg`). The *bash* process is spawned inside *client* when executing `__libc_res_nsearch`. Further, by checking the library-level provenance on the exit event of `client`, we conclude that the attack is caused by a vulnerability exploitation because `__libc_res_nsearch` does not return when `client` terminates. Note that Lprov effectively identifies the attack provenance even though the library call stack does not include `send_dg` and `send_vc` which are static functions (Related discussions can be found in Section 6).

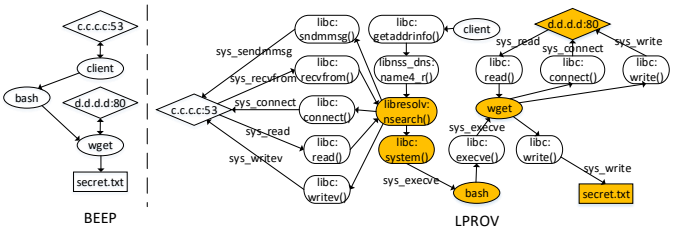


**Figure 5: Provenance graphs generated by BEEP and Lprov for library vulnerability exploitation.**

*5.2.3 Library Loading Analysis.* DARPA Transparent Computing program [6] investigates library loading processes in an effort to thwart advanced attacks that leverage malicious libraries. To demonstrate the effectiveness of Lprov, we test a sample library program provided by the red team of DARPA TC program. This case focuses on understanding internal details of library loading.

**Scenario** (1) Process *libloader A* is launched by *bash*. (2) *libloader A* spawns a thread *B* that prints out some text. (3) *libloader A* maps */dev/shm/testlib* into memory and reads *testlib.so* into the mapped memory. (4) *libloader A* dynamically loads */dev/shm/testlib* by `dlopen` and the library's constructor spawns another thread *C* that prints out some testing text. (5) *libloader A* calls 8 library functions `f1`-`f8` by `dlsym` and each function prints out some text. (6) *libloader A* prints out some text and exits.

**Provenance Analysis** Fig. 6 presents the provenance graphs tracking from the standard output device `/dev/stdout` and the two threads (*B* and *C*), where `testlib`, `libptd` and `ptd_create` are shorts for `/dev/shm/testlib`, `pthread_create` and `libpthread`. Lprov accurately correlates *testlib.so* and */dev/shm/testlib*, reveals the creation of thread *C* during */dev/shm/testload* loading and clearly

distinguishes output behaviors from the 8 different library functions, while BEEP misses all of those details due to the mishandling on memory mapping and the oblivion on library events. Note that the graph has two *libloader A* nodes spawned by *bash* because thread *B* has no causality to the two files */dev/shm/testlib* and *testlib.so* and this would be clarified when tracking from thread *B* (Fig. 6 is a combination of the provenance graphs on the three tracking objects).

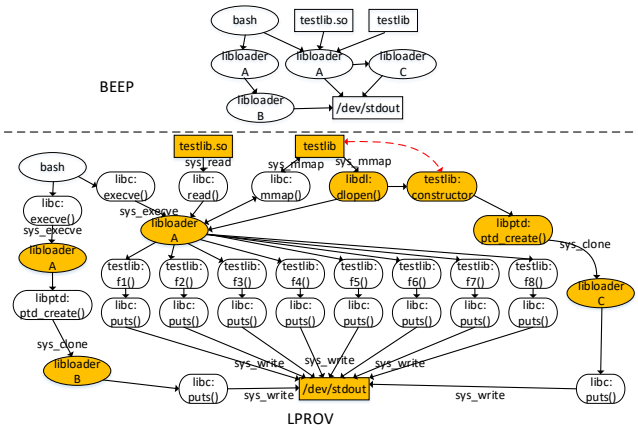


**Figure 6: Provenance graphs generated by BEEP and Lprov for DARPA library loading case.**

## 6 DISCUSSION

**Library Attestation** Binary attestation is a widely used technique to ensure program integrity. For instance, operating-system-level virtualization techniques such as application container and enclave on trusted hardware can leverage software attestation to provide trusted computing platforms. While it can be used to detect and prevent possible compromise on libraries, local attestation can be thwarted by replacing the attestation measurement and remote attestation can be compromised if the attestation interface is undermined in advance. Moreover, Lprov can provide detailed contextual information of attacks (e.g., how the system is compromised and which files/processes are affected) while library attestation would simply detect and thwart attacks. To this end, we argue that monitoring internal library behaviors is a crucial primitive to enhance the security of libraries.

**Tracing Protection** Like other approaches, Lprov assumes the kernel and user-space tracing facilities are benign and uncompromised. Note that it is a general assumption shared among various tracing systems. This assumption can be relaxed if the implementation of Lprov submerges into hypervisor level [23]. However, the hypervisor tracing has limited portability and it suffers from the high overhead. Guarding user-space tracing tools is a knotty problem as they can be subverted by other user-space programs. However, the *initial tampering* with the tracing infrastructure can be accurately recorded and users can revoke the trust on the logs from that time point. The same strategy is also applicable in tracee's runtime, where the library tracing is trusted as long as the memory image of *lprov.so* or the customized loader is not altered via `mprotect`. Note that if malicious libraries residing in the tracee can locate and overwrite the shared memory containing tracee's library events (e.g., via memory disclosure vulnerabilities), the initial tampering would not be captured since no syscalls are triggered. Nevertheless, we argue that hardening memory operations is an orthogonal research effort to Lprov.

**Tracing Coverage** Lprov traces library calls through the standard symbol resolution in `dynsym` table. Hence, Lprov might be less effective if malicious libraries opt for customized loading, static linking, or dynamic binary generation. For instance, executable binaries can be encoded into a library, then decoded and mapped at runtime. The following function execution (through hardcoded offsets) inside the mapped memory images is invisible to Lprov. Also, exported functions inside the same library can have execution dependence and it can be handled in either static linking or dynamic symbol resolution. For example, string-related functions like `strcpy` and `strlen` are frequently used inside other *glibc* functions, such as `getenv` and `setenv`. However, instead of inserting PLT trampolines at the library's ELF header to resolve them during runtime, *glibc* opts to statically link those functions as local ones with the `hidden_def` attribute. Unfortunately, the current implementation of Lprov is not able to identify such nested library calls (e.g., `strcpy` in `getenv`) as the inner ones are not resolved through the symbol resolver. However, note that we can still capture the outer functions (e.g., `getenv`) which are sufficient in practice. We argue that even with these limitations, it is still effective in the perspective of provenance investigation.

**System Overhead** As shown in evaluation, Lprov still has non-trivial overhead on *firefox* and it is actually due to the program's extremely heavy execution dependence on library functions. To mitigate this, Lprov can opt to whitelist several libraries and only trace recently changed ones, for example, libraries updated in the past four weeks. However, memory images of libraries cannot be trusted due to the manipulation of memory access permission from `mprotect`. Lprov hence needs extra communication channel between kernel and user space to dynamically deploy tracing on libraries whose memory image is contaminated during runtime.

## 7 RELATED WORK

**Non-unit audit logging** In recent years, significant efforts have been taken on the system-level audit logging [20, 21, 27, 28, 34, 38, 40, 46, 51, 53, 68]. They track system objects to infer mutual dependence for provenance investigation. Equipped with the generated system logs, root causes of attacks are revealed by backward analysis [15, 34, 39] and the aftermath of attacks is identified by forward tracking [21, 28, 38, 40, 52, 68]. Nevertheless, most of them regard the whole lifecycle of a process as a single system entity, which incurs dependence explosion problem. To assuage such concerns, researchers search for other system resources to complement syscall events. File offset is additionally leveraged in [55] to handle file-related syscalls to provide fine-grained tracing. However, it is only a specific optimization on file objects but not generic for other entities. In [42], memory operations between pages is utilized to establish low-level object dependence. But due to its lack of program semantics, the system is not sufficiently effective.

**Unit-based audit logging** Unit-based event tracing is the state-of-art in system-level logging and it follows the line of work that has

been done in audit logging. Improved from those coarse-grained techniques, unit-based schemes provide fine-grained provenance inference by program partitioning. In BEEP [44], relying on the observation that long-running programs usually respond user inputs by event handling loops, programs are trained by dynamic program analysis to extract those loops. Then the loops are instrumented by customized syscall events at both of enter and exit. Based on this technique, the execution of programs is partitioned into multiple loop instances and each instance is named a program unit. Therefore, dependence between input and output objects is efficiently disambiguated by confining causality correlation within the same unit. ProTracer [48] improves BEEP by applying dynamic object tainting to unit processing and decoupling high-overhead audit logging from event tracing. It performs between system tainting and audit logging to lower runtime and storage overhead. Each accessed object is assigned a unique taint and the taints are propagated upon object read. An event is considered for logging only if it is an output event. Furthermore, in the offline processing, the units sharing the same taint set are collapsed to make the generated graph concise. In MPI [47], the unit is refined to data structure instances but it requires efforts in developer annotation. Note that Lprov leverages existing unit-based provenance techniques.

**Hybrid approaches** In [60], to infer the causality relationship between syscalls, dependence analysis on low-level instructions is entailed. Such technique requires instruction instrumentation and incurs non-trivial runtime overhead. Since it works on deep and fine-grained program analysis, it involves much low-level memory dependence such as dynamic memory management which has no importance in causality inference. Barham et al. proposed the system Magpie [16] which monitors user-space events, kernel events, middleware and system resource usage for each application input by system-level instrumentation. It is a tool chain designed for system workload extraction under real-world conditions in operating systems. Although the system achieves high performance with relatively low overhead, it needs an application-specific schema to parse and correlate those system events, which restricts the system scalability. In [26], authors devised a logging and analysis system targeting intrusion detection, allowing users to specify the operating granularities, however, it requires users to provide the causality definition beforehand. Therefore, although the system offers implementation flexibility, it involves lots of human efforts and limits the system practicability. In RAIN [32], dynamic information flow tracking is applied in a replay-based provenance system to infer fine-grained causality. It leverages syscall events to minimize the analysis scope and performs refinable attack investigation by selective tainting, and it only incurs 3.22% overhead. Nevertheless, the tainting optimization in replay still requires pre-recorded instruction-level execution logs, and hence it is not applicable in Lprov's scenario.

**Information flow and object tainting** The inference of information flow between system objects is adopted by lots of existing forensic systems to enhance malware detection and analysis [50, 57, 64]. Yin et al. proposed the memory-level tracking system Panorama [64] to disclose the information flow between malware and sensitive data. The system can accurately capture malicious data access and processing launched by malware. But due to its high runtime overhead in low-level dependence analysis, it requires special support in hardware-level tracking. In [57], researchers invented a event tracing system VPath that can work either in OS kernel or virtual machine manager. It monitors thread behaviors and network activities to deduce system-wide causalities. Vpath can reveal precise information flow at low overhead, however, it needs carefully pre-defined patterns for system activities. Dynamic taint analysis in system entities is a widely used technique to perform provenance tracking. It can capture sensitive information leakage and system input causality through fine-grained data propagation [24, 25, 31, 37, 50, 52, 57, 62, 64–66]. This area has been well studied in multiple perspectives including file, socket and low-level kernel objects on various operating systems [24, 43, 65]. In spite of the high accuracy in provenance analysis, dynamic tainting system has limited usability due to its high overhead caused by heavy instrumentation. Besides, tainting can only tell whether there exists causality between two end objects but cannot offer a clear deriving path from the source to the sink.

**Log protection** Protecting log integrity is important. Jacobsson et al. proposed a lightweight logging-based malware detection system which operates in a real-time client/server audit mode [30]. The end-host under auditing delivers fresh runtime log data (encrypted and signed) to a remote server for further verification of system infection. However, it relies on the network communication and the connection between client and server becomes the weakest link. In [59], the authors proposed XRec, a primitive system offering integrity of execution trace on instruction level by branch trace messages. It can verify whether a specific code segment has ever been executed. However, it incurs 200%-400% performance overhead as it works in a system debug mode. In [17], researchers devised LPM (Linux provenance module), the first generic framework to build secure provenance-aware systems. It leverages LSM to create a trusted provenance-aware execution platform, collecting whole-system provenance with low overhead. Sundararaman et al. presented a secure disk system, SVSDS, that performs transparent versioning of data in the disk-level [17]. By enforcing specific data constraints, the system can protect executables and system log files. The latest secure logging systems are implemented on the trusted hardware execution of Intel SGX. In [36], the syslog is ported onto the boundary of trusted enclaves and application logs are generated through enclave *ocalls*. To secure the log storage, all the log data are encrypted on the disk using the SGX secrecy sealing/unsealing functionality. Note that they are complementary to Lprov.

## 8 CONCLUSION

Attack provenance is a persistent effort in enterprise networks. We develop and present a novel provenance-oriented library tracing system Lprov which enforces library tracing on top of existing syscall logging based provenance tracking approaches. Lprov is lightweight and efficient, offering much better support for heavy-threaded programs than existing tools such as *ltrace*. With the dynamic library call stack, the provenance of implicit library function execution is revealed and correlated to system events. The fine-grained provenance on library functions facilitates the locating and defense of malicious libraries (e.g. Linux Ebury). In experiments, our system prototype can precisely identifies the provenance inside malicious libraries with highly competitive overhead.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1748764 and 1409668, ONR under contracts N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

## REFERENCES

[1] 2018. Apache Benchmark. https://httpd.apache.org/docs/2.2/programs/ab.html.
[2] 2018. CVE-2015-7547. https://goo.gl/MTpo3V.
[3] 2018. CVE-2015-7547 Exploit-DB. https://www.exploit-db.com/exploits/39454/.
[4] 2018. CVE-2015-7547 Google Security Blog. https://goo.gl/rHw1C5.
[5] 2018. CVE-2015-7547 Patch. https://goo.gl/6ZhooX.
[6] 2018. Darpa Transparent Computing. https://goo.gl/EA77zv.
[7] 2018. ftpbench. https://github.com/selectel/ftpbench.
[8] 2018. An Introduction to KProbes. https://lwn.net/Articles/132196/.
[9] 2018. KProbes. https://goo.gl/SH2s4r.
[10] 2018. Linux Ebury. https://goo.gl/T677bv.
[11] 2018. Linux Security Module. https://goo.gl/gW2ykd.
[12] 2018. sunspider. https://webkit.org/perf/sunspider/sunspider.html.
[13] 2018. Tracepoints. https://goo.gl/TB6cas.
[14] 2018. Updating Host Keys. https://goo.gl/ztY8QT.
[15] Paul Ammann, Sushil Jajodia, and Peng Liu. 2002. Recovery from Malicious Transactions. *IEEE Trans. on Knowl. and Data Eng.* 14, 5 (2002).
[16] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *OSDI'04*.
[17] Adam Bates, Dave (Jing) Tian, Kevin R.B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *USENIX Security'15)*.
[18] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/2810103.2813691
[19] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The World's Fastest Taint Tracker. In *RAID'11*.
[20] Uri Braun, Simson Garfinkel, David A. Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I. Seltzer. 2006. Issues in Automatic Provenance Collection. In *IPAW'06*.
[21] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding Data Lifetime via Whole System Simulation. In *SSYM'04*.
[22] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *ISSTA'07*.
[23] Zhui Deng, Dongyan Xu, Xiangyu Zhang, and Xuxiang Jiang. 2012. IntroLib: Efficient and transparent library call introspection for malware forensics. In *DFRWS'12*.
[24] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and Event Processes in the Asbestos Operating System. In *SOSP'05*.
[25] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI'10*.
[26] Matt Fredrikson, Mihai Christodorescu, Jonathon T. Giffin, and Somesh Jha. 2010. A Declarative Framework for Intrusion Analysis. In *Cyber Situational Awareness - Issues and Research*. 179–200.
[27] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Middleware'12*.
[28] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. 2005. The Taser Intrusion Recovery System. In *SOSP'05*.
[29] William Hawkins, Anh Nguyen-Tuong, Jason D. Hiser, Michele Co, and Jack W. Davidson. 2017. Mixr: Flexible Runtime Rerandomization for Binaries. In *Proceedings of the 2017 Workshop on Moving Target Defense (MTD '17)*. ACM, New York, NY, USA, 27–37. https://doi.org/10.1145/3140549.3140551
[30] Markus Jakobsson and Ari Juels. 2009. Server-side Detection of Malware Infection. In *NSPW'09*.
[31] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. 2012. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *NDSS'12*.
[32] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. RAIN: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking. In *CCS'17*.
[33] Yang Ji, Sangho Lee, and Wenke Lee. 2016. RecProv: Towards Provenance-Aware User Space Record and Replay. In *IPAW'16*.
[34] Xuxian Jiang, Aaron Walters, Dongyan Xu, Eugene H. Spafford, Florian P. Buchholz, and Yi-Min Wang. 2006. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. In *ICDCS'06*.
[35] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS'11*.
[36] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing System Logs With SGX. In *ASIA CCS'17*.
[37] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *VEE'12*.
[38] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution. In *OSDI'10*.
[39] Samuel T. King and Peter M. Chen. 2003. Backtracking Intrusions. In *SOSP'03*.
[40] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality. In *NDSS'05*.
[41] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 437–452. https://doi.org/10.1145/3064176.3064217
[42] Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. 2010. Trail of Bytes: Efficient Support for Forensic Analysis. In *CCS'10*.
[43] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *ASPLOS '16*.
[44] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *NDSS'13*.
[45] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *CCS'13*.
[46] 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 *ACSAC'15*.
[47] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *USENIX Security'17*.
[48] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *NDSS'16*.
[49] Stephen McCamant and Michael D. Ernst. 2008. Quantitative Information Flow As Network Flow Capacity. In *PLDI'08*.
[50] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. 2009. Layering in Provenance Systems. In *USENIX ATC'09*.
[51] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-aware Storage Systems. In *USENIX ATC'06*.
[52] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *NDSS'05*.
[53] Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: Collecting High-fidelity Whole-system Provenance. In *ACSAC'12*.
[54] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO'06*.
[55] S. Sitaraman and S. Venkatesan. 2005. Forensic analysis of file system intrusions using improved backtracking. In *IWIA'05*.
[56] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS'08*.
[57] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. 2009. vPath: precise discovery of request processing paths from black-box observations of thread and network activities. In *USENIX ATC'09*.
[58] Dawood Tariq, Maisem Ali, and Ashish Gehani. 2012. Towards Automated Collection of Application-level Data Provenance. In *TaPP'12*.
[59] A. Vasudevan, N. Qu, and A. Perrig. 2011. XTRec: Secure Real-Time Execution Trace Recording on Commodity Platforms. In *HICSS'11*.
[60] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior Based Software Theft Detection. In *CCS'09*.
[61] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Rerandomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 367–382. http://dl.acm.org/citation.cfm?id=3026877.3026906
[62] K. Xu, H. Xiong, C. Wu, D. Stefan, and D. Yao. 2012. Data-Provenance Verification For Secure Hosts. *IEEE Transactions on Dependable and Secure Computing* 9, 2 (2012).

[63] Chao Yang, Guangliang Yang, Ashish Gehani, Vinod Yegneswaran, Dawood Tariq, and Guofei Gu. 2015. Using Provenance Patterns to Vet Sensitive Behaviors in Android Apps. In *Security and Privacy in Communication Networks'15*, Bhavani Thuraisingham, XiaoFeng Wang, and Vinod Yegneswaran (Eds.).
[64] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *CCS'07*.
[65] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *OSDI'06*.
[66] Hao Zhang, Danfeng Daphne Yao, and Naren Ramakrishnan. 2014. Detection of Stealthy Malware Activities with Traffic Causality and Scalable Triggering Relation Discovery. In *ASIA CCS'14*.
[67] Mingwu Zhang, Xiangyu Zhang, Xiang Zhang, and Sunil Prabhakar. 2007. Tracing Lineage Beyond Relational Operators. In *VLDB'07*.
[68] Ningning Zhu and Tzi-Cker Chiueh. 2013. Design, implementation, and evaluation of repairable file service. In *DSN'13*.

# A APPENDIX

## A.1 Kernel Event Tracing

A syscall interrupt (INT 80) traps program execution into kernel mode and the kernel module steps in at this moment. Linux offers several convenient and stable tracing facilities in kernel, LSM [11] (Linux security module), Tracepoints [13] and KProbes [8, 9] to register customized kernel event handlers. Specifically, LSM is often applied to implement MAC (mandatory access control) around kernel objects such as *inode*. It operates at a finer granularity than syscalls, which can incur additional overhead when a syscall accesses an object more than once, and miss customized syscall events introduced by unit partition in BEEP and Protracer [44, 48]. Tracepoint is a lightweight kernel tracing infrastructure which is adopted by many high-performance tracing tools such as Linux perf and audit logging. Unlike LSM, tracepoint allows tracing kernel work flows at different granularities (either object or function) through pre-defined tracing events. It statically embeds global event-tracing placeholders into kernel source code and users can then register effective probe functions on those tracepoint instances. Complementary to tracepoint, kprobe is a dynamic tracing infrastructure for kernel debugging. Among the three techniques, kprobe has the finest kernel tracing granularity since it provides the interface which can insert callbacks at any kernel-space instruction address, however, as a result of its purely dynamic design operating on software breakpoints, it has higher overhead than tracepoint. To this end, we select tracepoint as the underlying event handling mechanism to provide the workaround for syscall tracing inside the kernel module.

## A.2 Log Analysis Algorithm

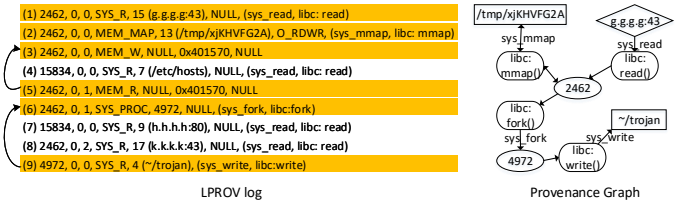


**Figure 7: Log analysis example. Log entries are in Lprov log format** $event = \{pid_e, uid_e, uinst_e, type_e, obj_e, para_e, lstack_e\}$

**Algorithm 1:** Lprov Log Analysis Algorithm

**Input** : $Log_{rev}$ - Lprov log in reverse event order
: $obj_t$ - designated tracking object
**Output**: $Graph$ - Lprov provenance graph
**Def.** : $Object$ - set of objects causally related to $obj_t$
: $event$ = $\{pid_e, uid_e, uinst_e, type_e, obj_e, para_e, lstack_e\}$ - a Lprov event in $Log_{rev}$
: $pid$ - process id
: $uid, uinst$ - unit id and unit instance
: $type_e \in$ {SYS_R/SYS_W, SYS_PROC, MEM_R/MEM_W, MEM_PERM, MEM_MAP} - event type
: $obj_e$ - system object in an event
: $para_e$ - event parameter
: $lstack_e$ - libcall stack including the syscall
: $Memory[pid]$ - set of memory use instances in $pid$
: $rel[u]$ - whether unit $u$ is causally related to $obj_t$
: $edge = (obj_a, obj_b, lstack)$ - a directed causality edge from $obj_a$ to $obj_b$ interconnected by $lstack$
**Init.** : $Object \leftarrow \{obj_t\}$
: $Graph \leftarrow \Phi$
: $Memory[pid] \leftarrow \Phi$
: $rel[u] \leftarrow False$

1 **foreach** $event \in Log_{rev}$ **do**
2  $unit \leftarrow (pid_e, uid_e, uinst_e)$
3  **if** $type_e \in$ {SYS_W, SYS_PROC, MEM_PERM} $\wedge$ $obj_e \in Object$ **then**
4  $Graph \leftarrow Graph \cup \{edge(pid, obj_e, lstack_e)\}$
5  $Object \leftarrow Object \cup \{pid_e\}$
6  $rel[unit] \leftarrow True$
7  **end**
8  **if** $type_e ==$ SYS_R $\wedge$ $rel[unit] == True$ **then**
9  $Graph \leftarrow Graph \cup \{edge(obj_e, pid, lstack_e)\}$
10  $Object \leftarrow Object \cup \{obj_e\}$
11  **end**
12  **if** $type_e ==$ MEM_R $\wedge$ $rel[unit] == True$ **then**
13  $Memory[pid] \leftarrow Memory[pid] \cup \{para_e\}$
14  **end**
15  **if** $type_e ==$ MEM_W $\wedge$ $para_e \in Memory[pid]$ **then**
16  $Memory[pid] \leftarrow Memory[pid] - \{para_e\}$
17  $rel[unit] \leftarrow True$
18  **end**
19  **if** $type_e ==$ MEM_MAP **then**
20  **if** $para_e \in$ {RDONLY, RDWR} $\wedge$ $rel[unit] == True$ **then**
21  $Graph \leftarrow Graph \cup \{edge(obj_e, pid, lstack_e)\}$
22  $Object \leftarrow Object \cup \{obj_e\}$
23  **end**
24  **if** $para_e \in$ {WRONLY, RDWR} $\wedge$ $obj_e \in Object$ **then**
25  $Graph \leftarrow Graph \cup \{edge(pid, obj_e, lstack_e)\}$
26  $Object \leftarrow Object \cup \{pid_e\}$
27  $rel[unit] \leftarrow True$
28  **end**
29  **end**
30 **end**

Algorithm 1 describes the process of backward tracking in Lprov. It takes as input the reverse-ordered log and a designated system object, and then generates a causal graph by correlating system entities in Lprov events. In Algorithm 1, a tuple of *pid*, *uid* and *uinst* defines a unique runtime process unit (line 2). Lprov has seven types of *event*, where SYS_R/SYS_W is the system read/write event such as socket or file read/write, SYS_PROC is the process creation event, MEM_R/MEM_W is the memory read/write event used to infer unit dependence, MEM_PERM is the memory permission event that makes executable memory pages writable (mprotect and anonymous mmap) and MEM_MAP is the non-anonymous memory mapping event. In a Lprov *event*, $obj_e$ is file descriptor for SYS_R/SYS_W and

**Table 3: Comparison of storage overhead between Lprov and ProTracer in a two-week performance experiment**

| User | ProTracer | | Lprov | | ProTracer/Lprov | |
|---|---|---|---|---|---|---|
| | #item(M) | size(GB) | #item(M) | size(GB) | item | size |
| A | 17 | 11 | 59 | 45 | 28.8% | 24.4% |
| B | 28 | 16 | 62 | 41 | 45.2% | 39.0% |
| C | 14 | 7 | 42 | 26 | 33.3% | 26.9% |
| D | 12 | 6 | 33 | 20 | 36.4% | 30.0% |
| Avg. | 18 | 10 | 49 | 33 | 36.7% | 30.3% |
| Prog. | ProTracer | | Lprov | | ProTracer/Lprov | |
| | #item(M) | size(GB) | #item(M) | size(GB) | item | size |
| httpd | 5.3 | 3.0 | 13.9 | 8.3 | 38.1% | 36.1% |
| proftpd | 3.8 | 2.2 | 8.3 | 5.4 | 45.8% | 40.7% |
| sshd | 4.3 | 2.5 | 15.4 | 10.8 | 27.9% | 23.1% |
| ssh | 1.4 | 0.9 | 4.1 | 2.8 | 34.1% | 32.1% |
| firefox | 27.1 | 15.1 | 68.0 | 49.6 | 39.9% | 30.4% |
| filezilla | 1.8 | 1.0 | 2.7 | 1.6 | 66.7% | 62.5% |
| w3m | 0.5 | 0.3 | 1.2 | 0.9 | 41.7% | 33.3% |
| wget | 0.4 | 0.2 | 0.8 | 0.5 | 50.0% | 40.0% |
| pine | 0.3 | 0.2 | 1.0 | 0.8 | 30.0% | 25.0% |
| vim | 3.4 | 1.9 | 15.2 | 11.0 | 22.4% | 17.3% |
| xpdf | 2.0 | 1.1 | 6.0 | 3.9 | 33.3% | 28.2% |

`MEM_MAP`, *pid* for `SYS_PROC`, memory section of a module/library for `MEM_PERM` and invalid for other event types; $para_e$ is the memory address for `MEM_R/MEM_W`, file and memory permission (`RDONLY`, `WRONLY` or `RDWR`) for `MEM_MAP`, and invalid for other event types; *lstack* is the library call stack for each causality edge and it is invalid for `MEM_R/MEM_W` events. Specifically, the algorithm can be inducted into three causality rules: a process unit has causality to the tracking object if (1) an event within the unit writes/creates any system entity that has causality to the tracking object (line 3-7, line 24-28) or (2) the unit writes a memory chunk which is read by another unit that has causality to the tracking object (line 12-18); while a system entity has causality to the tracking object if (3) the entity is read by an event within a unit that has causality to the target object (line 8-11, line 20-23). Fig. 7 is a log analysis example, where the target object `trojan` is casually correlated to the unit ($pid = 2462, uid = 0, uinst = 0$). The forward tracking algorithm is just the reverse version of Algorithm 1, and hence it is omitted in discussion.

## A.3 Additional Performance Evaluation

We apply the object tainting technique proposed in ProTracer to BEEP's logs, performing the comparison of storage overhead between Lprov and ProTracer. As Table 3 demonstrates, the storage consumption of ProTracer is 30.3% of Lprov. This is expected due to the inapplicability of ProTracer's tainting-based optimizations in our execution-based library provenance tracking, as discussed in Section 4.3. We argue that this is a tradeoff between cost and benefits.