ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting

Shiqing Ma
Purdue University
shiqingma@purdue.edu

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Dongyan Xu
Purdue University
dxu@cs.purdue.edu

***Abstract*—Provenance tracing is a very important approach to *Advanced Persistent Threat* (APT) attack detection and investigation. Existing techniques either suffer from the dependence explosion problem or have non-trivial space and runtime overhead, which hinder their application in practice. We propose ProTracer, a lightweight provenance tracing system that alternates between system event logging and unit level taint propagation. The technique is built on an *on-the-fly* system event processing infrastructure that features a very lightweight kernel module and a sophisticated user space daemon that performs concurrent and out-of-order event processing. The evaluation with different realistic system workloads and a number of attack cases show that ProTracer only produces 13MB log data per day, and 0.84GB(Server)/2.32GB(Client) in 3 months without losing any important information. The space consumption is only $<$ 1.28% of the state-of-the-art, 7 times smaller than an off-line garbage collection technique. The run-time overhead averages $<$7% for servers and $<$5% for regular applications. The generated attack causal graphs are a few times smaller than those by existing techniques while they are equally informative.**

I. Introduction

There is an increasing need of detecting and investigating APT attacks in an enterprise environment. A very important approach to addressing this problem is provenance tracking. According to previous works [17], [32], provenance captures multiple aspects of information about an entity in a system: *what* the entity's origin is; *how* the entity is derived; and *when* it originated. In the context of APT defense, entities with trackable provenance information are of various granularity, such as processes, network connections, files, and data items within files. Correspondingly, the *what*-provenance of such an entity e is the set of external entities that have causally influenced e's value or state (e.g., if one file's content comes from a number of network connections, then its *what*-provenance contains the IDs of the corresponding sessions); whereas, the *how*-provenance of entity e consists of events and their causal ordering – which can be organized as a causal graph – that demonstrates how (and when) other entities influence e's value or state.

NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23350

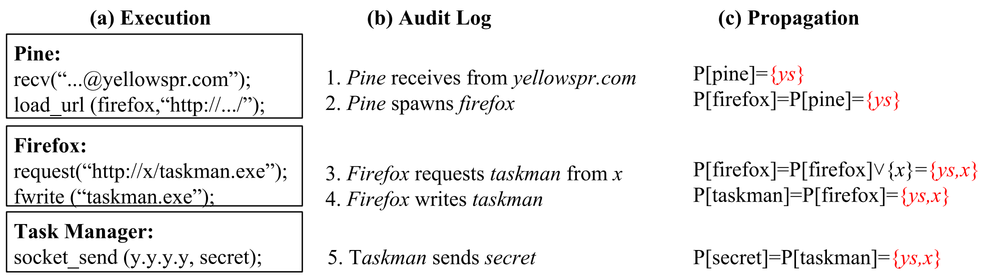
Existing Approaches. Existing techniques fall into two categories: *audit logging* and *provenance propagation (or tainting)*. Audit logging [16], [21], [25], [27], [29], [34]–[36], [39] records events during system execution and then causally connects events during attack investigation. They treat processes as subjects; files, sockets, and other passive entities as objects; and assume causality between subjects and objects involved in the same syscall event (e.g., a process reading a file). In general, audit logging incurs much lower overhead than per-instruction provenance propagation. Causal graphs can be constructed to denote both *what*- and *how*-provenance. Provenance propagation, or tainting [8], [11], [22], [23], [31], [37], [40], [51] works by first assigning IDs/tags to *provenance sources* (e.g., network sessions), and then propagating the IDs through program dependencies captured during execution. Provenance propagation usually entails set operations at the instruction level. Eventually, the set of provenance IDs that reaches a *sink* (e.g., a socket for send) denotes the sink's provenance. Provenance propagation usually only captures the *what*-provenance.

Consider the example in Figure 1. Figure 1 (a) denotes a simple attack. The user received a phishing email from attacker "Yellow Spring" and opened the URL in the email through `Firefox`. Upon visiting the website, a Trojan executable for task management was saved on the local disk. Later, the malware is executed and sends some secret to a remote host. Fig. 1 (b) shows the events captured by audit logging. Causality can be derived from events. Depending on the precision demanded and the scope of the analysis, events can be captured at different granularity (e.g., syscalls or memory accesses) and different scopes (e.g., host or whole enterprise).

Fig. 1 (c) shows the provenance propagation approach. IDs `ys` and `x` denote the different provenance sources. Observe that when `pine` spawns `Firefox`, the latter inherits the provenance of the former. The malware `taskman`'s provenance is the union of the provenance set of `Firefox` and the download URL `x`. At the end, we know the origins of the stolen secret, but we do not know its history. Such propagation can be exhibited within an application, across applications, and across hosts.

Both approaches have pros and cons, and neither meets the requirements for enterprise-wide APT detection/forensics. Logging has the following limitations:

(1) Dependence explosion is a major limitation of most audit logging. For a long-running process, an output event is assumed to be causally dependent on all preceding input events,



(a) Execution	(b) Audit Log	(c) Propagation

Pine:
recv("...@yellowspr.com");
load_url (firefox,"http://.../");
	1. *Pine* receives from *yellowspr.com*
	2. *Pine* spawns *firefox*
		P[pine]={*ys*}
		P[firefox]=P[pine]={*ys*}

Firefox:
request("http://x/taskman.exe");
fwrite ("taskman.exe");
	3. *Firefox* requests *taskman* from *x*
	4. *Firefox* writes *taskman*
		P[firefox]=P[firefox]∨{*x*}={*ys,x*}
		P[taskman]=P[firefox]={*ys,x*}

Task Manager: socket_send (y.y.y.y, secret);	5. T*askman* sends *secret*	P[secret]=P[taskman]={*ys,x*}

Fig. 1: Basic approaches to provenance tracing. (a) Actual executions in a top-down order; (b) Approach I: audit logging; (c) Approach II: provenance propagation.

and an input event is assumed to have causal influence on all subsequent output events. Such conservative assumptions create excessive false positive causal relations, making it difficult to reveal the true causality. In our previous work, we proposed to divide an execution to autonomous units [27] such that an output is only dependent on the preceding inputs *within the same unit*.

(2) High storage overhead. According to [28], audit logging easily generates gigabytes of log data per host every day. This is particularly problematic for APT defense, as APT malware tends to lurk in the victim host for a long time.

(3) Non-trivial run-time overhead. Although logging has relatively lower run-time overhead compared to provenance propagation because it does not require expensive per-instruction set operations, many existing logging systems [27], [28] are built on the default Linux audit logging infrastructure that can cause up to 40% slow-down to the whole system due to its poor design (Section V). This makes it undesirable in a production environment. Researchers have proposed advanced infrastructures [34]–[36] that can achieve much lower overhead. However, to achieve the low overhead, these systems usually do not perform any online event processing, but rather just record the events, leading to substantial space consumption and dependence explosion.

The propagation-based approach features much lower space overhead compared to logging as it does not generate log. It also has higher precision due to its fine-grained instrumentation. However it has many limitations that hinder its application in the real world:

(1) Substantial run-time overhead. Because propagation based techniques track individual instructions' execution and propagate (potentially) large provenance sets (Fig. 1 (c)), they usually incur substantial run-time overhead. State-of-the-art implementations without hardware support incur multiple factor of slow-down [23].

(2) Lack of implicit flow handling. Many propagation based techniques have difficulty handling *implicit flow*, which is information flow through control dependencies [30] (usually induced by program predicates).

(3) Complexity in implementation. Developers have to define provenance propagation logic for each instruction, a task which is tedious and error-prone. This problem is exacerbated when programs rely on third-party libraries; internal runtime engines (e.g., VMs); and various languages and their run-times, which all require specific instrumentation/tracking mechanisms.

In this paper, we develop ProTracer that leverages the advantages of both approaches and overcomes their respective limitations. It collects system events and processes them on the fly. The cost-effective online processing filters out events that are redundant or irrelevant for provenance analysis, substantially reducing the space consumption and the size of the generated causal graphs without affecting effectiveness.

System Goals. The goal of ProTracer is to provide efficient support for both the *what*-provenance and the *how*-provenance queries on any system objects such as processes and files. For example, given a corrupted file x, two *what*-provenance queries are: (1) *"What is the source/entry point of x?"* and (2) *"which other files in the enterprise were derived from (and corrupted by) x?"* A sample *how*-provenance query is: *"Construct a causal graph showing the events/entities that led to the corruption of x and those that have been further corrupted by x."* We aim to achieve *completeness*. In particular, the result of a what-provenance query on x must include all the external entities that directly/transitively affected x; the result of a how-provenance query must capture the set of internal and external entities that affected x and their causal relations with x.

The technique works as follows. It first leverages a selective instrumentation technique similar to BEEP [27] to partition an execution to units, by emitting special syscalls denoting the unit boundaries. Intuitively, an unit is an iteration of the event handling loop that processes an external request or a UI event. Different from [27], ProTracer does not simply log all the syscalls and the unit related events. Instead, it alternates between logging and provenance propagation. Logging is conducted when changes are made to the permanent storage or the external environment such as writing a file and sending a packet. For other events such as file reads and network receives, ProTracer performs *coarse-grained provenance propagation (tainting)*, which taints at the level of a unit and an system object (e.g. file) instead of an instruction and a memory byte. For example, if a unit receives packets from two network sessions x_1 and x_2, the unit is tainted with both sources. If later the same unit writes to a file on disk, a log entry is emitted containing the two sources. Then if the file is read by another unit, the unit is tainted with the two sources too. Note that avoiding logging as much as possible reduces the space overhead, and performing unit level and system object level taint propagation substantially reduces the run-time overhead compared to instruction level tainting. Unit level tainting does not lose any precision compared to a log-all-events strategy. Furthermore, ProTracer decouples its implementation from the expensive Linux audit logging system. It builds from scratch a highly optimized system. It has a lightweight kernel module that simply saves events to a ring buffer. The buffer is shared with a user space daemon that retrieves these events and processes them using a thread pool. ProTracer features out-of-order event processing, meaning that the event processing order does not need to be identical to the event order, maximizing concurrency.

Our contributions are summarized as follows.

- We propose the novel idea of combining both logging and unit level tainting to achieve cost-effective provenance tracing.
- We develop an efficient run-time that features on-the-fly event processing. It not only collects system events, but also filters out the redundant and irrelevant events on the fly. It achieves low run-time overhead by out-of-order event processing through a thread pool.
- We build a prototype and evaluate it on different systems with various users and workloads for over 3 months, and on a number of real-world attacks that we reproduce. Our results show that the space consumption of ProTracer is $<1.28\%$ of BEEP's on average, and about 7 times smaller than our previous offline log garbage collection technique LogGC [28]. The log generated per day is roughly 13MB without losing precision compared to BEEP. The run-time overhead averages $<7\%$ for servers and $<5\%$ for user systems, which is 4-10 times lower than the default Linux Audit Logging system, on which many techniques including BEEP were built, and comparable to lightweight logging systems such as Hi-Fi [34]–[36] that simply record events without processing them.

Like most existing audit logging systems [15], [27], [28], ProTracer trusts the kernel and any user space daemon associated with the provenance tracing system. More discussion about the assumptions, limitations and security analysis of ProTracer can be found in Section VI.

II. Motivation

Scenario: We will use a cyber attack scenario to motivate our technique. It is a *phishing attack*, in which an employee received an phishing email with a malicious link via *pine*, an email client. The email mentions that a free beta version of a costly program that the employee has been hoping to own is released on the Internet. The employee was excited and decided to try it out. He clicked the link; a new tab in `Firefox` was opened; he then downloaded the file to the local machine. However, the file is actually a back-door malware. Later when it is executed, a back-door process is started and sends some local file to a remote IP address.

State-of-The-Art: In BEEP [27], we observed that many programs share a common property: their execution is dominated by event handling loops. More importantly, individual iterations of these loops tend to handle relatively independent tasks such as serving a client request or handling a UI event. These observations were made by a study of more than 100 widely used open-source applications such as servers, browsers, and social networking applications. It was then proposed to partition an execution to autonomous units, each corresponding to an iteration of some event handling loop. In particular, program analysis was developed in [27] to recognize the unit-inducing loops, leveraging the following three observations: (1) such loops tend to be at the top level; (2) their loop bodies must make some I/O syscalls; and (3) their loop bodies dominate the execution time. Binary instrumentation is hence used to instrument the loop entry and exit points such that special syscalls are generated to indicate unit boundaries. *An output syscall is considered only dependent on the preceding input syscalls in the same unit,* whereas in other logging techniques [15], [16], [25], it is dependent on all the preceding input syscalls in the whole execution, leading to dependence explosion.

In some cases, a unit by itself may not fully cover the sub-execution that handles an independent input. Instead, a few inter-dependent units together constitute a semantically independent sub-execution. In practice, there are *memory dependencies* across unit boundaries. However, only some of them – called *workflow dependencies* – are helpful in connecting units that belong to the same sub-execution. Examples include the dependencies caused by the `enqueue` and `dequeue` operations of a task queue. In [27], inter-unit dependencies are identified via program analysis. A small number of memory operations that induce inter-unit dependencies are instrumented to emit special syscalls that help constructing the dependencies during off-line processing.

Fig. 2 shows the causal graph constructed by BEEP. The ovals on the left represent the units of `sendmail`, which checks the IP address through the Domain Name System (DNS) (a.a.a.a), and then interacts with the authentication server (b.b.b.b) and mail server (c.c.c.c) to fetch all emails. An email is further processed by a separate thread, whose unit is the one on the right of the dashed circle. The email is further filtered by `procmail` before it is opened by `pine`. Inside `pine`, the user clicks the phishing link, which triggers `Firefox`. `Firefox` uses multiple threads to process a request. The units in the dashed area correspond to units of the main thread and the tab thread, which uses an IPC channel `i.i.i.i` to communicate with a worker thread that downloads the backdoor file from `d.d.d.d`. The malware is later executed through `bash` and sends a file `f` to `e.e.e.e`.

Limitations of the State-of-the-Art. Although the causal graphs generated by BEEP (e.g. Fig. 2) are usually precise and concise, there are a few critical limitations that hinder the application of BEEP in practice.

(1) Substantial space overhead. BEEP generates a few GB log per-day for a system with a normal workload. This is because it logs all the provenance related syscalls including those generated by instrumentation. In [28], an offline garbage collection (GC) technique LogGC was proposed to prune redundant events from BEEP logs. However, it still requires storing all the events before pruning them. During pruning, it traverses the large log file back and forth in order to identify the redundant events. Due to the high cost of processing large files, one cannot afford running the GC technique frequently.

(2) Non-trivial run-time overhead. Although BEEP's instrumentation is lightweight, like many other audit logging systems

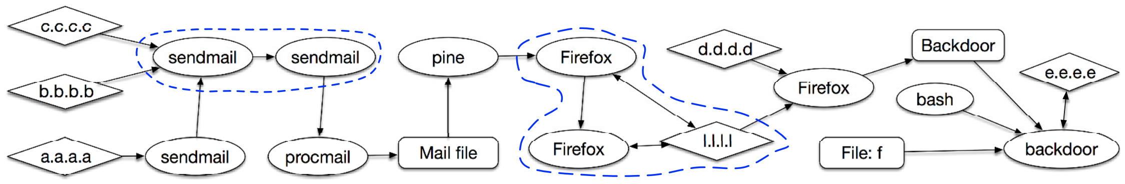


Fig. 2: The simplified causal graph of a phishing attack generated by BEEP [27]. The ovals represent execution units; the diamonds represent network sessions; the rectangles represent files. The nodes inside the dashed areas are those pruned away by ProTracer.

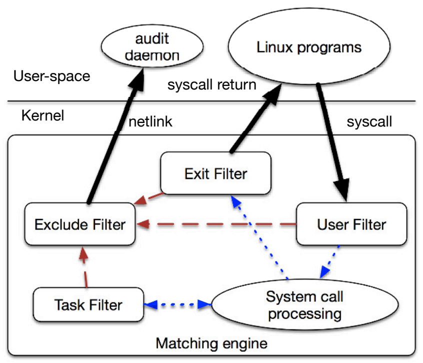


Fig. 3: Linux Audit system architecture.

(e.g. [15]) it is unfortunately built on the Linux Audit system that has non-trivial run-time overhead by itself. According to our experiment (Section V), the overhead can be as high as 43%.

A further inspection reveals that the Linux Audit logging system is unnecessarily heavy-weight. Fig. 3 illustrates the architecture of the Linux Audit logging system. It consists of two main parts: a kernel module for system call processing and a few audit applications that process/store auditing events, managed by an dispatcher daemon `audisp`. The kernel module receives syscalls from Linux programs. A syscall first goes through the *user* filter that decides if the syscall will be further sent to the other kernel modules for further processing. The *user* filter also forwards the syscall to the *exclude* filter to determine if the syscall should be prevented from being sent to the audit apps. After the syscall is processed (by other modules), the return state needs to go through the *exit* filter and then the *exclude* filter to determine if the state is interesting for auditing. The control is only given back to the Linux program after all these activities.

Note that most of the filtering work is done on the kernel side, which blocks the application execution for a long time. Second, all types of syscalls have to go through filters even if they are not interesting. It uses *netlink* to send data from the kernel to the user-space daemon, which is slow. Finally, the audit applications write to the log file, which also generates a lot of events that need to go through the costly procedure.

In Hi-Fi [36], researchers have developed a more advanced logging infrastructure with a substantially lower run-time overhead (3% in a representative workload). They leverage the *Linux Security Modules* (LSM) that allow adding light-weight hooks before accesses to kernel objects such as `inodes`, and use a high performance buffer to deliver kernel object access events to a user space logging application. Despite its low overhead, Hi-Fi does not perform event processing on the fly hence it records all events. Furthermore, kernel object access events are at a level lower than syscalls. As a result, some commonly used syscalls such as file read may lead to many kernel object accesses, which induce additional overhead. Finally, LSM hooks may have difficulty handling customized syscalls introduced by BEEP as those syscalls do not lead to any kernel object accesses. As such, the capability of solving dependence explosion cannot be easily ported to Hi-Fi.

The Basic Idea of ProTracer. We improve the practicality of provenance tracing by the following two aspects.

In the first aspect, We develop a lightweight kernel module. We will leverage a kernel facility called Tracepoints [5]. A tracepoint can be placed in both user and kernel code to provide a hook to call a kernel function (*probe*). In ProTracer, we will insert tracepoints to kernel functions that process provenance related syscalls (e.g., `sys_clone`). The tracepoint driver is extremely lightweight and simply stores the events to a ring buffer, which will be processed by the user space daemon through a pool of threads. More details can be found in Section III.

In the second aspect, we avoid logging as much as possible by alternating between tainting and logging. We only log when files are written to the disk or packets are sent through sockets for either IPC or real network communication. For other syscalls that only lead to intra-process information flow such as file reads and network receives, we perform unit level taint propagation. Tainting has the following benefits:

(1) Avoid logging redundant events. Consider the dashed area for `Firefox` in the middle of Fig. 2. At the entry point to the area on the left, ProTracer will introduce a new taint to represent the provenance of the hyper link, which is essentially the sub-graph to the left of the area. The taint is further propagated through the nodes inside the dashed area. Note that since no external accesses are performed inside the area, the taint remains the same until it gets out the area. As such, we avoids logging events in that duration without losing any provenance information. The same applies to the dashed box for `sendmail`. Similarly, consider an FTP server. Each unit of the server corresponds to processing a client request. Assume the client request is to upload a large file, which entails many network receive syscalls. In a pure logging system such as [27], all the syscalls need to be logged. In ProTracer, logging these events is avoided by taint propagation. In fact, all these syscalls have the same taint and do not add to the taint set.

(2) Avoid logging dead events. Tainting also allows ProTracer to handle the large number of syscalls that do not have any

permanent effects on the system. We call them the *dead events*. For example, syscalls related to temporary files represent a large portion of a raw audit log [28]. However, since these files are just used internally and never accessed by others, their taint propagation usually does not reach any other file writes or network sends and hence does not trigger any logging. Lets consider the FTP server example again. Assume during the processing of the file upload request, the connection is lost. The FTP server will eventually timeout and exit the execution unit without writing any data. In this case, the taint source representing the data session with the client IP is not propagated to any updates on the storage. Thus nothing needs to be logged.

Note that the aforementioned two kinds of reductions are different from the reduction in LogGC [28], which is an *offline* log garbage collection technique. LogGC is based on reachability so that all the events in the dashed area of Fig. 2 cannot be pruned as they are reachable from the backdoor process. Furthermore, it requires first acquiring the entire log file and then traversing the large file back and forth to identify unreachable items, incurring high cost.

(3) Allow concurrent event processing. Introducing a new taint to represent a provenance set allows out-of-order event processing. For example, by introducing a new taint when the dashed region of `Firefox` in Fig. 2 is entered, the processing of the `Firefox` events does not have to wait for the processing of the events in the sub-graph on the left of the shaded area. This maximizes the utilization of the thread pool. More details can be found in Section IV.

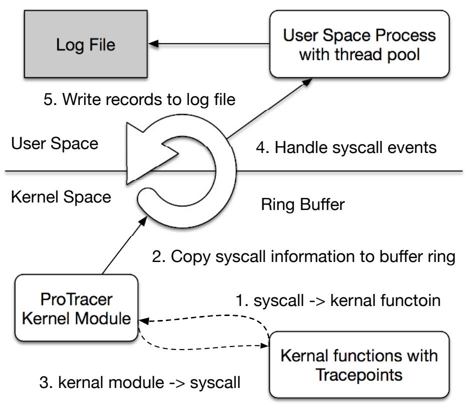

Log File
User Space Process with thread pool
5. Write records to log file
User Space
4. Handle syscall events
Kernel Space
Ring Buffer
2. Copy syscall information to buffer ring
ProTracer Kernel Module
1. syscall -> kernal functoin
Kernal functions with Tracepoints
3. kernal module -> syscall

Fig. 4: System architecture overview, dashed lines denote control flow, solid lines denote data flow, and numbers denote the order of the events.

III. System Architecture

The architecture of ProTracer is shown in Fig. 4. The system consists of two main parts: a kernel module and a user space daemon process. The kernel module is responsible for collecting syscall events and writing them to the ring buffer. The user space process fetches and handles these events, including deciding to log the events or perform taint propagation.

When a Linux application makes a syscall, the execution is trapped to the kernel space and the application is blocked until the kernel finishes processing the syscall. It is hence critical to ensure the kernel module is lightweight. ProTracer makes use of an existing lightweight Linux kernel trace facility, Tracepoints [5]. In particular, we identify the set of kernel functions that handle syscalls that can induce causality with system objects or other processes. They mainly fall into the following categories.

- All syscalls that operate on file descriptors (representing regular files, network sockets, device files, pipes and so on), including creation, read, write, and close.
- Special syscalls that help trace taints on certain types of objects. For example, *sys_bind* for sockets.
- IPC syscalls operating on pipes, semaphores, message queues, shared memory, and UNIX domain sockets.
- Process manipulation syscalls including process creation, termination, and privilege changes (escalation or degradation).
- Syscalls generated by program instrumentation to denote unit boundaries and inter-unit workflow.

The syscalls that are not monitored are mainly for time management (e.g. timer_create), fetching information from file system or kernel (e.g. getpid), and those not implemented (e.g. getpmsg). To our knowledge, the set is complete for provenance tracing with certain assumptions. Detailed discussion can be found in Section VI. Tracepoints are inserted at the entry and exit points of the kernel functions. They are lightweight hooks that can hand over the execution to our kernel module so that the syscall and its context can be copied to the ring buffer. The trace points at the entries are to collect the parameters while those at the exits are to collect the syscall results. We separate the two to allow better concurrency in event processing. Our kernel module is also responsible for managing the ring buffer to avoid any event loss.

ProTracer uses a user space daemon process to process the syscall events. To increase throughput, the daemon process uses a pool of worker threads, which is different from most existing works. All events are time-stamped so that we do not need to worry about the event order in the buffer and in the log file. A general worker thread assignment policy is that *syscalls from the same application cannot be processed by more than one worker thread*. In other words, event processing is in order for the same application. But it may be out-of-order for events from different applications. For each event, the daemon process needs to decide to log it or to perform taint propagation. More details are discussed in Section IV. All threads share a log buffer that stores the log records. The log records are written to disk only when the buffer is nearly full or the system is in a relatively idle state so that we can reduce the number of disk I/O operations.

To achieve good performance, ProTracer uses a ring buffer to share data between the kernel module and the user space daemon. The ring buffer is similar to the high performance buffer in Hi-Fi [36], which is also memory-mapped to the user space so that it can be accessed without any copy operations. However, we choose to use tracepoints for syscall interception instead of LSM hooks, to support customized syscalls and to trace at the syscall level instead of the lower kernel object access level.

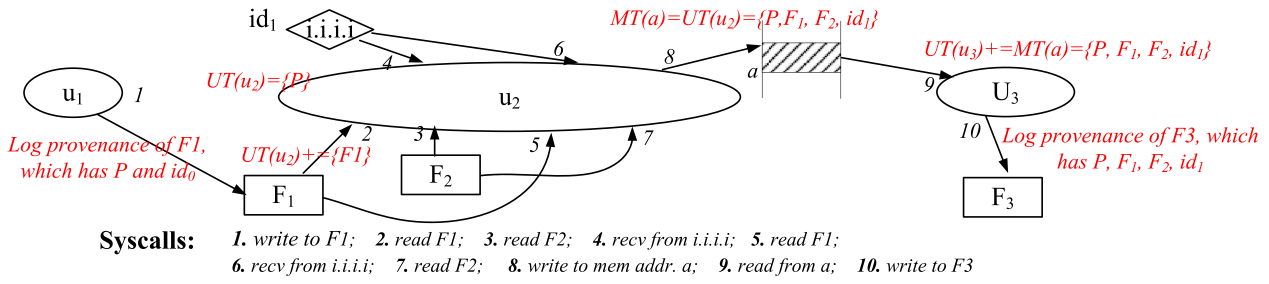

id$_1$
i.i.i.i
$MT(a)=UT(u_2)=\{P, F_1, F_2, id_1\}$
$UT(u_3)+=MT(a)=\{P, F_1, F_2, id_1\}$
6
8
4
a
9
$UT(u_2)=\{P\}$
u$_1$
1
u$_2$
U$_3$
10
Log provenance of F3, which has P, F_1, F_2, id_1
Log provenance of F1, which has P and id_0
2
3
5
7
$UT(u_2)+=\{F1\}$
F$_2$
F$_1$
F$_3$
Syscalls:
1. write to F1; 2. read F1; 3. read F2; 4. recv from i.i.i.i; 5. read F1;
6. recv from i.i.i.i; 7. read F2; 8. write to mem addr. a; 9. read from a; 10. write to F3

Fig. 5: An abstract diagram to illustrate the logging and tainting run-time. The numbers represent the order of the events. The logging/tainting behavior is highlighted in red on edges.$UT(u)$ and $MT(a)$ are simplified representations of the taint set of a unit u and address a, respectively. P denotes the current process and id_0 an ID denoting a network session.

IV. Tainting and Logging in the User Space Daemon

In this section, we explain the user space daemon that alternates between tainting and logging. The basic scheme is intuitively illustrated by Fig. 5. When receiving a syscall event, the daemon checks if it is a syscall that makes permanent changes to the external state (e.g., a file write or a socket send). If so, it logs the current taint set of the event to disk, which denotes the provenance of the associated object (e.g., the logging action in red on edge 1). When a new unit starts (i.e., the event handling loop starts to process a new and independent request), the taint set associated with the unit is reset to only containing the process itself (e.g., $UT[u_2] = \{P\}$ to the left of u_2), meaning the provenance of the unit only contains the current process.

Upon an input event, a new taint is created to denote the current provenance set of the input object (e.g., the new taint F_1 on edge 2 denoting the current provenance set of F_1 and ID id_1 on edge 3 denoting the network session). The taint is then added to the taint set of the unit, denoting that now the unit is causally related to the corresponding input source. Input syscalls only trigger taint propagation instead of logging. Upon a memory write representing workflow, the current unit taint set is propagated to the memory (e.g., the highlighted behavior on edge 8). Later, when another unit loads from the same memory location, the memory taint set is propagated to the unit (e.g., the behavior on edge 9). Eventually, when unit u_3 writes to F_3, the provenance of F_3 is the current unit taint set. Note that F_3's provenance set contains F_1, implying a causal edge between this event and the previously logged event about F_1. In our implementation, we associate timestamps with taints and events to facilitate recovery of such causality. It is worth noting that although there are 10 syscall events, only two entries are logged, which are sufficient to disclose both the what- and how-provenance of F_1 and F_3. In particular, the how-provenance is represented by the causal graph.

Next, we discuss the details of our design using an abstraction of the system.

A. Definitions

The definitions related to our discussion are presented in Fig. 6. To support tainting, three data structures are introduced to store taints for objects, units, and memory, respectively. In particular, we use an *ObjectTaintStore* structure to associate a singleton taint to an object of two possible kinds: *Inter-Process Communication objects* (IPCs) that are essentially a special kind of sockets, and memory-mapped files. We use a *UnitTaintStore* structure to associate a process with a set of taints, denoting the taints of the current execution unit, which is usually an iteration of the event handling loop. *MemTaintStore* associates a set of taints with a memory address, which is to support intra-process taint propagation. ProTracer selectively instruments a very small number of critical memory reads and writes that denote the inter-unit workflow (i.e., high level data flow [27]) of the application, e.g. the reads and writes of a task queue that is used to pass user requests across execution units. A taint can be a time-stamped IPC, file, or an ID that represents a taint source, which can be either a network session or an email received. In other words, we use IDs to denote external sources. The mapping is maintained by a *TaintSource* structure. In contrast, for objects internal to the system we may use a taint consisting of the object and a timestamp ts to denote the provenance of that object at ts, which may represent a set of IDs (e.g., in event 2 in Fig. 5 F_1 denotes the current provenance of file F_1, which contains P and id_0).

As mentioned in Section I, we cannot capture all important provenance by taint propagation alone, which does not record the history of an object or a process. As such, in addition to taint propagation, we also log important events. More specifically, we log all the permanent changes to the system, such as file writes, file deletes, outgoing network traffic, and process creation, together with their taints. *LogBuffer* is a memory buffer to store these changes. We use a memory buffer to avoid frequent disk accesses. More importantly, the memory buffer allows us to easily avoid logging events related to temporary files, which are often in a large number. More discussion can be found later.

As mentioned in Section III, ProTracer intercepts all syscalls related to provenance, including those related to units. We abstract these syscalls to a few representatives as shown in Fig. 6. In particular, they denote file, IPC, network session, process spawn, memory reads/writes denoting inter-unit workflow, unit enter/exit, and taint source related operations. The runtime behavior corresponding to these events will be discussed next. Note that although our implementation intercepts both the entry and the exit of a kernel function that handles a syscall, our abstraction combines the two events into one abstract event for discussion simplicity.

$$
\begin{array}{lll}
OT \in \mathit{ObjectTaintStore} & ::= & (\mathit{IPC} \mid \mathit{File}) \rightarrow \mathit{Taint} \\
UT \in \mathit{UnitTaintStore} & ::= & \mathit{Process} \rightarrow \mathcal{P}(\mathit{Taint}) \\
MT \in \mathit{MemTaintStore} & ::= & \mathit{Address} \rightarrow \mathcal{P}(\mathit{Taint}) \\
t \in \mathit{Taint} & ::= & (\mathit{IPC} \mid \mathit{File} \mid \mathit{ID}) \times \mathit{TimeStamp} \\
\mathit{TSrc} \in \mathit{TaintSource} & ::= & \mathit{ID} \rightarrow (\mathit{Session} \mid \mathit{Email}) \\
LB \in \mathit{LogBuffer} & ::= & \overline{(\text{WRITE} \times (\mathit{File} \mid \mathit{Session} \mid \mathit{IPC}) \times \mathcal{P}(\mathit{Taint}) \times \mathit{TimeStamp} \mid} \\
 & & \overline{\text{DEL} \times \mathit{File} \times \mathcal{P}(\mathit{Taint}) \times \mathit{TimeStamp} \mid \text{FORK} \times \mathit{Process} \times \mathcal{P}(\mathit{Taint}) \times \mathit{TimeStamp})} \\
e \in \mathit{Event} & ::= & \mathit{FileOpen}(\mathit{Process}, \mathit{File}) \mid \mathit{FileRead}(\mathit{Process}, \mathit{File}) \mid \mathit{FileWrite}(\mathit{Process}, \mathit{File}) \mid \\
 & & \mathit{FileDel}(\mathit{Process}, \mathit{File}) \mid \\
 & & \mathit{IPCRead}(\mathit{Process}, \mathit{IPC}) \mid \mathit{IPCWrite}(\mathit{Process}, \mathit{IPC}) \mid \\
 & & \mathit{SessionCreate}(\mathit{Process}, \mathit{Session}) \mid \mathit{SessionRead}(\mathit{Process}, \mathit{Session}) \mid \\
 & & \mathit{SessionWrite}(\mathit{Process}, \mathit{Session}) \mid \\
 & & \mathit{Fork}(\mathit{Process}, \mathit{Process}) \mid \\
 & & \mathit{MemWrite}(\mathit{Process}, \mathit{Address}) \mid \mathit{MemRead}(\mathit{Process}, \mathit{Address}) \mid \\
 & & \mathit{UnitEnter}(\mathit{Process}) \mid \mathit{UnitExit}(\mathit{Process}) \mid \\
 & & \mathit{EmailRecv}(\mathit{Process}, \mathit{Email})
\end{array}
$$

$$f \in \mathit{File} \quad c \in \mathit{IPC} \quad a \in \mathit{Address} \quad p \in \mathit{Process} \quad x \in \mathit{Session} \quad m \in \mathit{Email} \quad ts \in \mathit{TimeStamp}$$

Fig. 6: Definitions for Logging and Tainting.

TABLE I: Logging and Tainting Rules.

Rule #	Event	Action
1	$\mathit{FileOpen}(p, f)$	$OT[f] = \langle f, \mathbf{getTime}() \rangle;$
2	$\mathit{FileRead}(p, f)$	$UT[p] \cup= \{OT[f]\}$
3	$\mathit{FileWrite}(p, f)$	$LB = LB + \langle \text{WRITE}, f, UT[p] \cup \{OT[f]\}, \mathbf{getTime}() \rangle$
4	$\mathit{FileDel}(p, f)$	**if** (f is owned by p) $LB = LB - \langle *, f, *, * \rangle$; **else** $LB = LB + \langle \text{DEL}, f, UT[p], \mathbf{getTime}() \rangle$; $OT[f] = \mathit{nil}$;
5	$\mathit{IPCRead}(p, c)$	$OT[c] = \langle c, \mathbf{getTime}() \rangle;\ UT[p] = UT[p] \cup \{OT[c]\}$
6	$\mathit{IPCWrite}(p, c)$	$LB = LB + \langle \text{WRITE}, c, UT[p], \mathbf{getTime}() \rangle$
7	$\mathit{SessionCreate}(p, x)$	$t = \mathbf{newSource}();\quad OT[x] = \langle t, \mathbf{gettime}() \rangle$
8	$\mathit{SessionRead}(p, x)$	$UT[p] = UT[p] \cup \{OT[x]\}$
9	$\mathit{SessionWrite}(p, x)$	$LB = LB + \langle \text{WRITE}, x, UT[p], \mathbf{getTime}() \rangle$
10	$\mathit{Fork}(p_1, p_2)$	$LB = LB + \langle \text{FORK}, p_2, UT[p_1], \mathbf{getTime}() \rangle$
11	$\mathit{MemWrite}(p, a)$	$MT[a] = UT[p];$
12	$\mathit{MemRead}(p, a)$	$UT[p] \cup= MT[a]$
13	$\mathit{UnitEnter}(p)$	$UT[p] = \{\langle p, - \rangle\}$
14	$\mathit{EmailRecv}(p, m)$	$t = \mathbf{newSource}();\quad UT[p] = UT[p] + \{\langle t, \mathbf{getTime}() \rangle\}$

B. Run-time Operation Rules

Table I describes the taint propagation and logging operations conducted by the threads in the user space daemon. A worker thread (in the daemon) receives an event from the ring buffer and processes it based on the rules in the table.

File Operations. Rules 1-4 are for file related event processing. For a file open event with process p opening a file f, ProTracer creates a new taint that consists of the file object and the current timestamp. The taint denotes the provenance set of the file at this moment, which may include multiple external sources. The principle is that ProTracer uses a singleton taint to represent a provenance set for a system object that can propagate information across processes, including file and IPC. This is a critical design decision which will be further explained. The taint is then associated with f through the *ObjectTaintStore* OT. Upon a file read, the taint set of the current execution unit of p is enhanced with the taint of f, meaning the current execution of p is also affected by the provenance of f. Upon a file write, a log entry is inserted to the log buffer denoting the write operation and the associated taint set, which is the union of the current unit taint set and the file taint (Rule 3). Intuitively, after the write, the file inherits the taints of all the preceding input syscalls in the same unit. The design choice of using a singleton taint to denote the provenance (taint) set of an object on external storage has a few critical advantages over the design of directly propagating provenance sets.

- An object may be transitively dependent on a large set of taint sources. It is expensive to propagate taint sets, which entails allocating space and performing set unions. Hence, ProTracer uses a singleton taint consisting of the object and a timestamp to denote the current taint set of the object and propagates the taint.
- The design allows out-of-order processing of events in the ring buffer. As mentioned earlier, the kernel inserts events with timestamps to the ring buffer and the user space daemon retrieves and handles these events from the same buffer. Events from different processes may be dispatched to different threads that execute concurrently. As such, event processing across

processes may be out-of-order. For example, assume two applications A and B. A writes to a file f and closes it before B reads it. The file read event (in B) may be processed before the file write (in A) is processed. If we directly propagate the taint set of f from A to B, we have to wait for the file write to be processed before processing the file read, substantially limiting concurrency. With the current design, the file read will use a fresh taint, without waiting for the computation of the set. The timestamps of the taint set (recorded to the log buffer at the write event) and the fresh taint (introduced at the file read) would allow ProTracer to infer the proper mapping between the set and the new taint during the offline causal analysis.

- The design allows us to record not only the *what* provenance, but also the *how* provenance. Traditional techniques based on standard tainting [21] can only record the set of taint sources associated with an object, missing the history about how the object was created and updated. With the current design, each time an object is updated (i.e. written to the permanent storage), a log entry representing the set of taints of the object is recorded.

Upon deleting a file (Rule 4), ProTracer not only resets the taint of f, but also removes all the log entries in the buffer related to f if the process p is the exclusive owner of f, meaning f is a temporary file that does not escape the lifespan of its owner. We say the p is the owner of f if p creates f and f is never read by another process. If p is not the owner, the log entries related to f cannot be removed as the history of f may still be of interest. For example, an APT attack may remove a malicious library generated in an earlier phase of the attack (by another process) to cover its trail. The history of the malicious library is still valuable although it is deleted. In addition, the deletion event itself needs to be logged as it is part of the malicious behavior. The log buffer is flushed to the disk when it is close to full. It often takes a long time for the log buffer to reach its capacity so that most temporary file deletes happen before the buffer is flushed, allowing the pruning of dead log events (Section II) such as temporary file reads and writes.

IPC Operations. Processes may use IPC (e.g. pipes) to communicate with each other. Upon an IPC write (Rule 6), a log entry is added to denote the write and the provenance of the write, which is essentially the current unit taint set. Following the design policy of using singleton taints to allow out-of-order processing, upon an IPC read a new taint consisting of the IPC object and the current timestamp is created and added to the unit taint set of the receiver process (Rule 5).

Network Operations and Process Spawn. Network operations are handled similar to file operations. We consider a network session as a unique taint source. As such, each time a session is created, a new taint ID is created and associated with the session. When a process p receives packets from a session, the taint of the session is added to the unit taint set of p (Rule 8). When p sends a packet through a session, the provenance of the network send is denoted by the unit taint set of p. A log entry containing the taint set is recorded. Such entries allow ProTracer to construct causality across hosts. When a process p_1 spawns another process p_2 (Rule 10), the provenance of the child is the unit taint set of its parent. A log entry is added to record the fork and the corresponding taint set.

Execution Unit Related Operations. These events are generated by selective program instrumentation [27]. Application executables are instrumented in a very small number of places to emit special syscalls to indicate the beginning and the end of an execution unit, and memory operations that denote the high level workflow between units. ProTracer needs to propagate taints through the memory object involved. Upon a write to a memory object, the unit taint set is propagated to the object (Rule 11). Later, when the same memory object is read in another unit, its taint set is inserted to the taint set of the new unit (Rule 12). As mentioned in Section II, execution units are considered autonomous and their correlations are only through the workflow related memory objects explicitly monitored by ProTracer. Therefore, when the execution leaves a unit and enters a new unit, the unit taint set is reset to only containing the process itself (Rule 13).

Taint Source Operations. Upon events such as receiving an email, a new ID representing the source is created and inserted to the unit taint set (Rule 14). Note that these events may be at a higher level than syscalls. In our implementation, the corresponding protocol libraries are instrumented to generate these high level events.

Example. Consider the example in Fig. 7. We have two programs running in the system: a browser and a PDF reader. Parts of the code snippets of the two applications are shown. Although the code snippets simulate the workflow in a real-world browser and a real-world PDF reader, they are substantially simplified and abstracted to be consistent with our definitions in Fig. 6. Specifically, the browser has two threads: the UI thread that handles UI events and the worker thread that performs background operations such as downloading a file. The event handling loop dominates the execution of the UI thread. The beginning of the loop is instrumented by a function `UnitEnter()` that will produce an event denoting the start of a unit. In lines 8-11, if the UI event is the click of a hyper link, the URL is added to the work queue. Since the queue operations denote the workflow across units, the enqueue operation is instrumented to generate a memory write event (line 9). The worker thread execution is dominated by the loop in lines 22-36, which acquires a request from the work queue and processes it. Lines 23-24 denote the unit instrumentation and the memory read instrumentation. If the request is to access a URL, a temporary file "`tmp`" is created to store the downloaded content. A session is created and used to download the resource (lines 28-29). The downloaded content is written to the file (line 30). An IPC object is created to communicate with the PDF reader to display the PDF file (lines 32-33). The temporary file is deleted at the end (line 34).

The PDF reader is also event driven. If it receives an IPC request to render a PDF file, it acquires the file through IPC and saves it to `buf` (lines 56-57) before rendering it. If it receives a UI request to save the PDF file, it creates a file and writes `buf` to the file (lines 62-64). ProTracer detects that `buf` carries workflow across units (i.e. the loop iterations corresponding to

(a) Browser code

```
   UI Thread
1  …
2  /*UI event dispatching loop*/
3  while (true) {
4    UnitEnter ();
5    ui_event = poll (…)
6    if (ui_event.type==…)
7      …
8    if (ui_event.type==HYPER_LNK) {
9      MemWrite(q.tail( ));
10     q.enqueue(ui_event.url);
11   }
12   …
13   UnitExit ();
14 }
```

```
   Worker Thread
20 …
21 /*Event processing loop*/
22 while (!q.empty( )) {
23   UnitEnter( );
24   MemRead(q.head( ));
25   u = q.dequeue( );
26   if (u==URL_REQ) {
27     f= FileCreate("tmp");
28     x=SessionCreate (u, …);
29     buf=SessionRead(x);
30     FileWrite (f, buf…);
31
32     c=IPCCreate(reader, ...);
33     IPCWrite(c, f);
34     FileDel(f);
35     ...
36 }
```

(b) PDF reader code

```
50 …
51 /*Event processing loop*/
52 while (true) {
53   UnitEnter( );
54   e = poll(…);
55   if (e==OPEN_BY_IPC) {
56     c=IPCCreate(browser,…);
57     buf=IPCRead(c);
58     MemWrite(buf);
59     pdf_render(buf);
60   }
61   if (e==SAVE_AS) {
62     o= FileCreate("a.pdf");
63     MemRead(buf);
64     FileWrite (o, buf…);
65     ...
66 }
67
```

(c) A sample system execution

Program	TimeStamp	Event	Rule	OT[]/MT[]	UT[]	LB
Browser	1	UnitEnter(b)	13		UT[b]={OT[b]}={<b,->}	
	2	MemWrite(b,q[0])	11	MT[q[0]]=UT[b]={<b,->}		
	3	UnitEnter(b)	13		UT[b]={OT[b]}={<b,->}	
	4	MemRead(b,q[0])	12		UT[b]=UT[b] U MT[q[0]]={<b,->}	
	5	SessionCreate(b,x)	7	OT[x]=<t_1,5>		
	6	SessionRead(b,x)	8		UT[b]=UT[b] U {OT[x]}={<b,->, <t_1,5>}	
	7	FileWrite(b,f)	3			LB=<W,f,{<b,->,<t_1,5>},7>
	8	IPCWrite(b,c)	6			LB=<W,f,{<b,->,<t_1,5>},7>; <W,c,{<b,->,<t_1,5>},8>
	9	FileDel(b,f)	4	OT[f]=nil		LB= <W,c,{<b,->,<t1,5>},8>
Reader	10	UnitEnter(r)	13		UT[r]={OT[r]}={<r,->}	
	11	IPCRead(r,c)	7	OT[c]=<c,11>	UT[r]=UT[r] U OT[c] ={<c,11>,<r,->}	
	12	MemWrite(r,buf)	11	MT[buf]=UT[r]={<c,11>,<r,->}		
	13	UnitEnter(r)	13		UT[r]={<r,->}	
	14	MemRead(r,buf)	12		UT[r]=UT[r] U MT[buf]={<c,11>,<r,->}	
	15	FileWrite(r,o)	3			LB=<W,c,{<b,->,<t_2,5>},8>; <W,o,{<c,11>,<r,->},15>

Fig. 7: Example for the logging and tainting run-time. The shaded statements correspond to syscalls. The statements in red are those instrumented by ProTracer to generate special events.

the IPC and the save-as-a-file operations), the read and write of `buf` are instrumented (lines 58 and 63).

Fig. 7 (c) shows a sample execution of the system, in which the user clicks a hyper link denoting a remote PDF file, the file is then downloaded and rendered by the reader, and finally the user further saves the file. The table shows the events generated by ProTracer and how the run-time processes these events. The second column shows the timestamps; the third column shows the events with process b and r denoting the browser and the reader, respectively. The fourth column shows the rules applied and the last three columns show the state of the various data structures.

Observe that in the first unit corresponding to the click of the hyper link, the `UnitEnter` event causes the unit taint of b to be reset to $\{\langle b, -\rangle\}$. Upon the `MemWrite` at 2, the taint of the queue is updated to contain the taint of the current unit. The execution then proceeds to the unit from the worker thread that downloads the file. At 3, the unit taint set is reset. At 4, the taint set of the queue is unioned with the unit taint set. At 5, since a network session is considered an external source, an ID t_1 is generated to denote the source. The taint of the session is inserted to the unit taint set at 6 due to the `SessionRead` event. At 7, the downloaded content is saved to the temporary file f, and thus a log entry is inserted to the log buffer LB to denote the write and the provenance. The file is further passed to the reader through an IPC c. The `IPCWrite` event leads to another log entry at 8. At 9, the deletion of f leads to the removal of the first log entry as f is a temporary file.

Timestamps 10-15 correspond to the execution of the reader, which consists of two units. The first one renders the file and the second one saves the file. At 11, a new taint is created to denote the provenance set of the IPC object c at timestamp 11, which essentially denotes the set $\{\langle b, -\rangle,\ \langle t_1, 5\rangle\}$. The taint is inserted to the unit taint set of r. The unit taint set is propagated to `buf` at 13. In the second unit (timestamps 13-15), the taint set of `buf` is retrieved and inserted to the unit taint set. When the file is written, another log entry is added to denote the write.

There are three important things that we need to point out. (1) Although there are 15 events, ProTracer only needs to log two of them, which are the two in LB at the end. In other words, on-the-fly taint propagation avoids storing a lot of events. (2) ProTracer introduced a new taint $\langle c, 11\rangle$ to denote the provenance set of c at 11 such that the processing of the reader events and the processing of the browser events can be performed concurrently by different threads. And from the timestamp 11 and the log, we know that $\langle c, 11\rangle$ must represent the taint set in the first log entry. (3) The log entries reflect the history of the file, whereas existing techniques only track the sources of the file.

C. Handling Global File Accesses

A long running execution can often be divided to three parts. The first one is the start phase, which is responsible for loading configurations, allocating resources like file descriptors for application log files. The second one is the event handling loop, which handles a large number of external requests. The third one is the closing phase, where all resources are deallocated before the process terminates. In the previous sections, we mainly focus on the event handling loops, which dominate and generate units. However, handling the other two phases, especially file operations in those phases, is equally important. Files opened in the start phase are often used throughout the whole execution. For example, the `Apache httpd` server opens its application log files (e.g. access log and error log) in the start phase. The access log will be written within any unit that handles an external request. This log file is a shared object across most of the units, which would cause unnecessary dependence between units. To address this problem, we apply a special policy to objects opened in the start phase. In particular, these objects are stated as global in the log. During execution, they are not considered as shared objects and operations on these files are not logged.

Unlike log files, which are opened in the start phase and not closed until the end phase, some objects used in the start phase have a very short life time. They are usually opened, read and then closed. Typical examples include configuration files and libraries used by an application. Our policy is to log these events, because the data read from these files can be possibly utilized for a malicious purpose. An example is that a malicious library downloaded from a remote site is loaded by a normal application in the start phase.

Discussion: Completeness of ProTracer. As introduced in Section I, we aim to capture all the external and internal entities that affect a system object and their casual relations. With the assumption that all the provenance related syscalls are intercepted by ProTracer, we want to show that the alternation between tainting and logging and the pruning of events (e.g., through file deletions) do not affect completeness. According to the rules in Table I, within a unit, all input events are captured and propagated to the taint set of the unit. With the assumption that all the inter-unit workflows through memory accesses are captured[1], the taint set is properly propagated across units. Upon an outgoing syscall, the set is logged. During offline analysis, causal edges are introduced by connecting a log entry containing a taint (of an input file), such as taint F_1 on edge 2 in Fig. 5, and a preceding log entry containing the provenance set corresponding to the taint (e.g., the log entry generated by the action on edge 1 in Fig. 5). This ensures not only that the set of external sources is complete, but also the history of the object is captured (by the causal graph). The way that ProTracer prunes log entries related to temporary files is safe because only the log entries related to a file owned by the process are removed. When the file is owned, its information cannot reach other processes. In the case that a temporary file is copied to another permanent file, its provenance is completely inherited by the permanent file so that the temporary file log entries are no longer useful. ProTracer also precludes syscalls caused by application logging as they introduce bogus causality across units. Note that application log usually records a subset of what ProTracer is already recording and is hence redundant.

[1]We will discuss situations where our assumptions may not hold in Section VI.

V. Evaluation

In this section, we will show the evaluation results of ProTracer. The experiments were conducted on five identical machines with four cores and 4GB RAM. Most of the binaries in our experiment machines have been instrumented by BEEP [27].

A. Effectiveness

Daily Usage: In the first experiment, we emulated the daily (24 hours including break time) usage of five computer users, and collected logs. To compare the space consumption, we run both ProTracer and BEEP [27] at the same time during the experiment. After the logs were generated, we also applied LogGC [28] to garbage collect the BEEP logs to acquire the reduced logs. To create workload diversity, we emulated users exhibiting different usage patterns: User 1 uses the system to run a web server for a group project. An FTP server is also running in the same system; User 2 is new to Linux. He just tries out various applications in the system and accesses his personal emails and the Internet; User 3 is preparing for an exam. She is mainly reading documents and watching video lectures; User 4 uses *vim* a lot to finish his course project report besides accessing the Internet; User 5 mainly uses the system for watching movies and communicating with friends. In addition, we performed an extended 3-month emulation of user 1, whose machine is used as a server; and user 4, who actively uses client programs.

We compare the number of log entries and the log file sizes for BEEP, LogGC, and ProTracer. The results are presented in Table. II. Columns 8 and 9 show the ratios between the ProTracer logs and the BEEP logs, whereas the last two columns show the same ratios between the LogGC logs and the BEEP logs. Observe that ProTracer can significantly reduce the number of events that need to be logged. On average, ProTracer only needs to log 1.85%(Daily)/1.45%(3 months) of the entries in BEEP. Since ProTracer records taint IDs, our log file format is slightly more efficient than BEEP and LogGC. On average, ProTracer's disk space consumption is only 1.28%(Daily)/1.02%(3 months) of BEEP's. The results vary for different users because of the different workloads and use patterns. Even in the worst case (user 1 that hosted servers), ProTracer generated less than 4% of the log entries, and consumed less than 2% of the disk space.

Compared to LogGC, ProTracer has better space efficiency in most cases. This is reasonable because LogGC garbage-collects events based on their reachability from live system objects. In other words, events that do not contribute to any

	Logs	BEEP		ProTracer		LogGC		ProTracer/BEEP		LogGC/BEEP	
		# Items	Size(KB)	# Items	Size(KB)	# Items	Size(KB)	# Item	Size	# Item	Size
3 Months	Server	274,242,874	157,839,158.66	3,371,453	880,584.40	45,429,888	21,311,715.08	1.23%	0.56%	16.57%	13.50%
	Client	500,928,294	168,269,687.89	7,844,783	2,437,009.51	30,449,339	10,037,472.39	1.57%	1.45%	6.08%	5.97%
	Average	387,585,584	163,054,423.27	5,608,118	1,658,796.95	37,939,614	15,674,593.74	1.45%	1.02%	9.79%	9.61%
Daily	1	3,610,501	1,673,830.40	133,159	30,830.20	637,596	280,576.00	3.69%	1.84%	17.66%	16.76%
	2	1,730,339	581,389.08	37,205	8,873.10	293,957	98,768.73	2.15%	1.53%	16.99%	16.99%
	3	2,221,662	743,986.67	9,558	2,245.02	3,382	1,351.68	0.43%	0.33%	0.15%	0.18%
	4	3,768,783	1,265,993.45	52,009	16,078.02	183,947	60,139.52	1.38%	1.27%	4.88%	4.75%
	5	2,471,859	829,968.11	23,998	7,226.04	22,248	8,970.24	0.97%	0.87%	0.90%	1.08%
	Average	2,760,629	1019033.54	51,186	13,086	228226	89961.23	1.85%	1.28%	8.27%	8.83%
Applications	Apache	3,262,452	4,570,766.23	47,305	31,380.62	288,482	404,797.44	1.45%	0.69%	8.84%	8.86%
	Vim	1,089,732	370,508.82	11,215	3,053.91	99,452	34,017.28	1.03%	0.82%	9.13%	9.18%
	Firefox	3,672,740	4,655,331.54	302,873	288,344.52	352,587	447,406.08	7.70%	6.62%	9.60%	9.61%
	W3M	368,752	259,001.72	25,793	18,068.22	82,959	57,671.68	6.99%	6.98%	22.50%	22.27%
	ProFTPD	48,374	12,183.53	689	124.80	4,539	1,228.80	1.42%	1.02%	9.38%	10.09%
	Wget	873,205	189,782.34	7,938	897.81	4212,847	88,811.52	0.91%	0.47%	47.28%	46.80%
	Mplayer	858,236	188,811.93	240	16.00	0	0.00	0.03%	0.01%	0.00%	0.00%
	Pine	59,125	21,285.72	648	286.72	642	327.68	1.10%	1.35%	1.09%	1.54%
	Xpdf	56,083	9,534.20	64	16.00	0	0.00	0.11%	0.17%	0.00%	0.00%
	MC	7,823	9,026.81	26	8.00	0	0.00	0.33%	0.09%	0.15%	0.00%

TABLE II: Comparison of effectiveness of various systems with different duration (3 months and 24 hours); users, and applications.

live system objects are removed. ProTracer not only avoids logging such dead events (Section IV), but also precludes redundant events that affect live system objects (e.g. repetitive socket reads from the same session and execution units that do not access any taint sources but rather serve as part of the information flow path). In some cases, LogGC is more space efficient (e.g. user 3). This is mainly due to temporary files. LogGC is an offline log reduction method, which has sufficient information to precisely decide if a file is temporary. ProTracer uses the log buffer to delay writing log entries to the disk, hoping that the temporary file related entries will be removed by the time the buffer is flushed. However, some temporary files related log entries will be flushed to disk if the files are not deleted by the time the buffer is flushed. Besides, ProTracer also logs events that belong to the start and end phases of an execution, whereas BEEP/LogGC ignores those events. For user 3, `MPlayer` and `Xpdf` were frequently used and they produced a large number of temporary files. LogGC was able to remove all the records that belong to these two programs, whereas ProTracer keeps some of them. Also observe that on average, ProTracer only generates 13MB log per day, which is very affordable.

Application Perspective: We also compare the space consumption of the different systems on various applications. The logs specific to applications are extracted from the whole system logs. The results are presented in the lower half of Table II. Observe that the ProTracer logs are significantly smaller compared to BEEP Logs. The number of records is reduced to less than 8%, and the log size shrinks to less than 7% for all programs. The results vary across different applications due to the different behavioral patterns of the applications. For browsers like `Firefox`, accessing a single web page can introduce many taints as it may access the web server, advertisement server, image storage server and so on. Since each resource request will cause a log entry, the log buffer is filled up much faster and more frequently, compared to other applications. As a result, ProTracer records more temporary files related events. Programs like `Xpdf` interact with files and the screen. There is no outgoing information via sockets or other files. ProTracer only needs to record a small number of events in the start and the end phases. For most programs, ProTracer occupies less space than LogGC. But for some of them (e.g. `Xpdf`), LogGC performs better. This is because LogGC ignores the events in the start and the end phases of a process. However, the results also show that the overhead is minor for these applications.

B. *Logging Overhead and Scalability*

We also perform experiments to study the run-time overhead and scalability of ProTracer. Fig. 8 shows the accumulated log size over time for user 1. The solid line shows the growth of the BEEP log size over time, and the dashed line shows ProTracer's. In general, the growth is similar although the scales of the sizes are different. The sharpest growth occurs in the 15th-20th hour, indicating the user was intensively using the system. Even in this period, the growth of the ProTracer log is about 13MB, suggesting very good scalability. There are some shape differences between the two lines near the 20th hour. This is mainly because ProTracer has better log reduction for the applications used during that time period, compared to other applications.

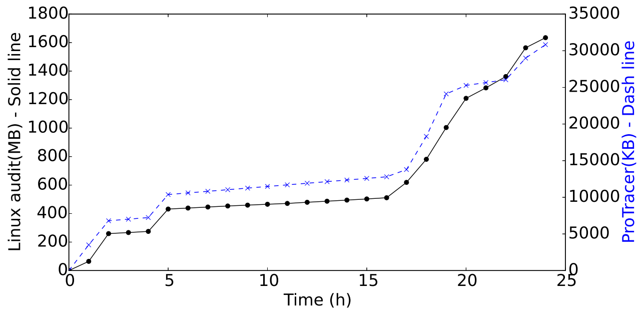


Fig. 8: Accumulated log size from the one-day execution of BEEP and ProTracer. Note the size of BEEP log is measured by *megabytes*, whereas the ProTracer log is measured by *kilobytes*.

Fig. 9 and Fig. 10 show the run-time overhead comparison between ProTracer and the default Linux Audit system, with the same set of syscalls monitored. Note BEEP is built on the Linux Audit system and hence more expensive. We perform two sets of experiments. The first one is for server programs. We use the `Apache Benchmark` [1] to test two web servers `Apache` and `MiniHttp`, and `ftpbench` to test `ProFTPD`. We also test different concurrency configurations, with the number of requests sent at the same time being 1, 2, 4, and 8. The results are shown in Fig. 9. The benchmarks tend to give the system a lot of pressure, which would cause higher overhead than regular usage. The baseline we use is the native Linux system without running the Linux Audit system. Observe that the overhead of ProTracer is less than 7%, whereas the Linux Audit system has a much more significant overhead (more than 5 times larger).

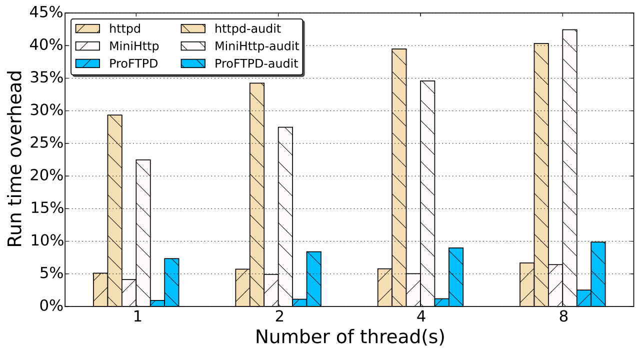


Fig. 9: Run-time overhead with different concurrent thread(s) for server programs.

We also perform experiments for client programs. We use standard benchmarks if they are available such as `SunSpider` for `Firefox`. Otherwise, we use the batch mode for programs like `vim` or `W3M`. We perform the experiments with ProTracer and with the Linux Audit system. The baseline we use here is the native Linux system without any logging system running. The results are shown in Fig. 10. Observe that ProTracer has less than 3.5% run-time overhead for all these programs, whereas the overhead of the Linux Audit system is 7-8 times larger.

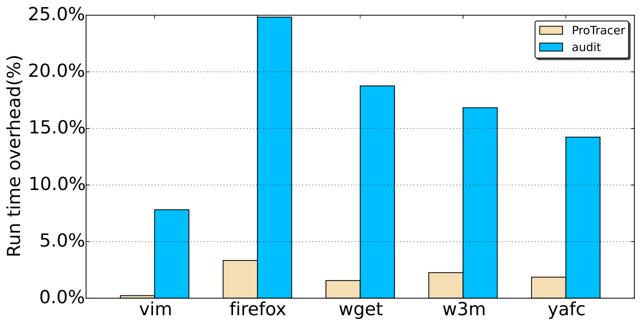


Fig. 10: Run-time overhead for client programs.

C. Attack Investigation Cases

In this section, we use a number of attack cases to show that the causal graphs generated by ProTracer during attack analysis are smaller than those by BEEP, but equally informative, and the time taken to generate the graphs is much less. We reproduce a few realistic attack scenarios for our experiment. With each scenario, we perform two *what*-provenance queries to understand the sources and the ramifications of the attacks, and also the *how*-provenance query to understand the attack path (Section I). We compare the query results by BEEP and ProTracer, and also cross-check with our prior knowledge. To emulate real-world attack scenarios, each experiment lasted for a few hours with the attack performed in the middle.

The first case is a *backdoor attack*. The attacker detected that the running FTP server was ProFTPD-1.3.3c, which had a backdoor command [3]. He compromised the server, and was able to get a *bash* shell. He then downloaded a backdoor program using *wget*, and started this backdoor to get permanent access. A few days later, the administrator got a warning that the FTP server had a backdoor, and decided to check if the backdoor had been exploited. If so, what damages have been inflicted.

The second scenario is *information theft* [27]. An employee had a under-the-table deal with one competitor of his own company: he copied some information from the company, and leaked it to the competitor by pasting it to a public page via vim. When the company found that the information was leaked, they should be able to pair the file that contained the information with the web page that leaked the information, among thousands of files. ProTracer shall also allow them to prove that the attacker leaked it among all the other employees that have the access to the file.

The third scenario is *illegal storage* [16]. One of the server administrators wanted to store some illegal files on a server. However, he did not want the files to be in his own directory. Instead he created a directory under another user's home directory, and downloaded the illegal files to the directory. He replaced the `ls` program to hide the existence of this directory. When the files were eventually found, the victim user was considered a suspect because of the presence of those files in his/her directory. The investigator should be able to identify that the administrator was the one that committed the crime. Note here we assume that the administrator cannot tamper with the log file generated by ProTracer.

The forth scenario is *cheating student* [4]. An instructor's password was stolen by a student. The student downloaded a file containing midterm scores from `Apache`, and uploaded a modified version. The instructor noticed that the average score became higher, and started to suspect someone had modified the file. Luckily, students used static IP addresses on campus and off-campus IPs were forbidden to connect to the server. So finding the IP from which the current file was uploaded would help identify the student. Moreover, the administrator should be able to find other suspicious activities of the student. In our case, the student also downloaded a few files containing answers to future quizzes.

The fifth is *phishing email* [2] that was discussed in Section II.

Parts of the results are shown in Table III. The second column shows the experiment duration. The next three columns show the size of the logs by different systems. Then we show the time it takes to perform the queries. The last two columns show if BEEP and ProTracer produce matched results for the two *what*-provenance queries (i.e., the *backward* query for attack sources and the *forward* query for attack ramifications). For the first scenario, the backward query is not applicable.

Scenario	Duration	Log file size(KB)			Run time(s)		Investigation	
		BEEP	LogGC	ProTracer	BEEP	ProTracer	Backward	Forward
Backdoor attack	3h54min	832,753	174,693	79,834	74	11	-	Match
Information theft	4h22min	587,494	94,759	13,938	39	5	Match	Match
Illegal storage	2h58min	369,585	63,375	10,864	32	5	Match	Match
Cheating student	1h17min	179,748	29,485	9,385	17	3	Match	Match
Phishing email	4h36min	975,753	183,795	82,343	64	8	Match	Match

TABLE III: Attack scenarios. *Backward* means backward *what*-provenance query; and *forward* means forward query; *match* means ProTracer is able to precisely and concisely uncover the attack path.

Observe that ProTracer produces much smaller logs and the query processing time is much shorter. The query results show no qualitative differences and precisely disclose the provenance.

Scenario	#source	#process	#file	#nodes
Backdoor	33/33	23/23	37/66	580/128
Infor theft	1/1	4/4	21/36	148/82
Illegal storage	24/24	6/6	56/72	388/208
Student hacker	2/2	2/2	67/85	432/226
Phishing email	5/5	8/8	12/12	864/305

TABLE IV: Causal graph comparison (BEEP/ProTracer).

To further compare the quality of the query results. We compare the causal graphs generated by BEEP and ProTracer in answering the forward queries. The results are shown in Table IV, which shows the number of taint sources, the number of processes (i.e. the internal nodes along the attack path), the number of files affected by the attacks, and the number of nodes in the graphs. Note that LogGC would produce the same graphs as BEEP. Observe that the two systems produce the same set of taint sources and processes. The differences in the files are due to the fact that BEEP does not log file accesses in the start and the end phases (e.g. loaded libraries). In this sense, we argue that the ProTracer-induced graphs are more complete. Finally, the ProTracer-induced graphs are much smaller than graphs generated by BEEP, reducing the human inspection efforts.

To acquire an intuitive understanding of the differences between ProTracer graphs and BEEP graphs, we present parts of the graphs by BEEP and ProTracer for the forward query in the backdoor attack case. The query aims to find all the reachable items from the external IPs connected to the FTP server. There are three connections. The one from `a.a.a.a` downloaded and uploaded a file, and exploited the backdoor. The one from `b.b.b.b` simply downloaded and uploaded a file. The one from `c.c.c.c` lost its connection. Observe that the ProTracer graph is a lot more concise and clear. This is because using tainting, ProTracer avoids logging many events and generating nodes for those events. For example, the box on the bottom of the BEEP graph shows a zoom-in view of part of the graph. It is reduced to a single node (`FTP-a0`) in our graph. And our graph is still equally informative. Moreover, the events related to `c.c.c.c` are precluded from our log in the first place as the taints did not propagate to any permanent changes.

VI. Discussion

In this section, we will discuss the limitations of ProTracer.

(1) While the execution of many programs can be divided autonomous units which are only connected through workflow memory dependencies, this may not hold for all programs. For programs that do not have unit structure (i.e., does not have event handling loops), ProTracer treats the entire execution as a unit, which may cause dependence explosion.

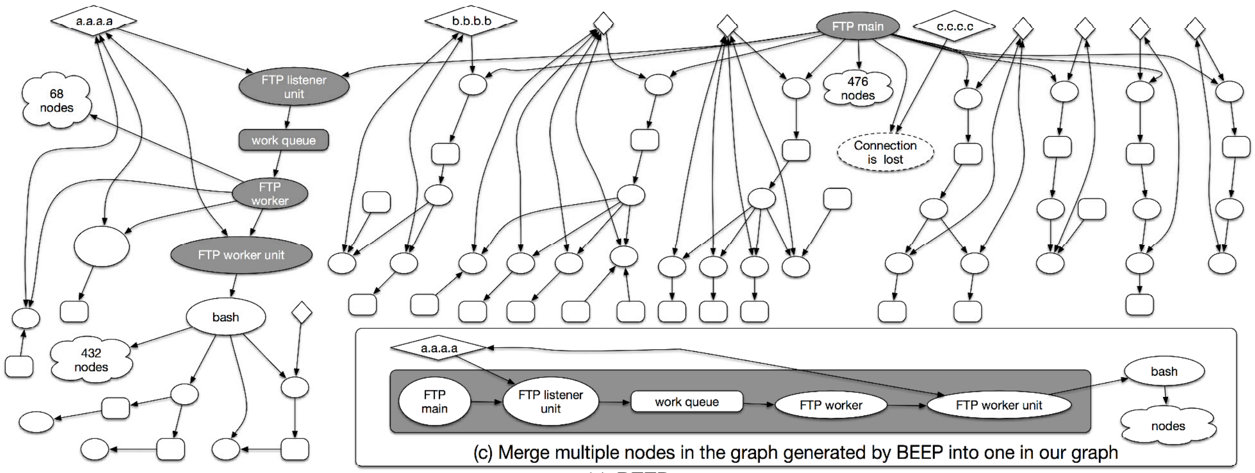
(2) Similar to BEEP [27], ProTracer relies on training runs to identify unit loops and workflow dependencies. However, the training may not be complete. If unit loops cannot be properly identified, ProTracer treats the entire execution as a unit. Once a unit loop is identified, the corresponding workflow dependencies can be identified as they rarely change [27]. In theory, however, ProTracer may miss such dependencies hence the corresponding memory accesses are not instrumented, leading to broken causal paths.

(3) Just like most audit logging systems, ProTracer requires that the kernel and the user space daemon are not compromised. This limitation can be mitigated by porting ProTracer to a hypervisor. Furthermore, if a system is clean to begin with and an attacker successfully subverts the system at a later time, the initial subversion will be accurately captured by ProTracer. But the log entries after the system's subversion cannot be trusted.

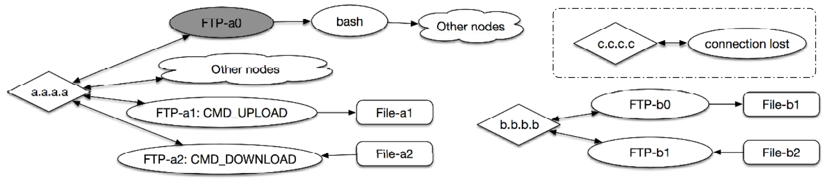
(4) Similar to most logging systems, ProTracer excels at capturing provenance through benign and commonly used applications, such as browsers and editors, as many attacks leverage these applications. In contrast, malware usually makes use of various methods to protect themselves such as obfuscation and self-modification, which may create trouble for ProTracer's analysis. As a result, ProTracer usually treats malware execution as a single unit. We argue that this is reasonable because all malware actions are – by definition – of interest (instead of noise) to attack investigation.

VII. Related Work

System logging: Lots of works [9], [10], [15], [16], [21], [24], [26], [29], [34], [35], [35], [36], [52] have been done in tracking provenance using system-level audit logs. However, many of them suffer from *dependence explosion* and have high overhead. BEEP [27] and LogGC [28] are the most closely related work. ProTracer uses similar unit partitioning to avoid dependence explosion. Compared to BEEP and LogGC, ProTracer has much lower space and run-time overhead, due to the new infrastructure and the integration of tainting and logging. The graphs by ProTracer are also much more concise. Some of these techniques [34]–[36] provide high performance. However, they do not perform any on-the-fly reduction but rather simply store the whole traces. They may also be susceptible to dependence explosion.



(a) BEEP



(b) ProTracer

Fig. 11: Part of the graphs generated by BEEP and ProTracer for the backdoor case.

Dynamic information flow tracking and tainting: Tainting and dynamic information flow tracking [12]–[14], [20], [23], [33], [42], [47]–[50] have been studied from different aspects (e.g., file system, kernel object level, network flow) on different platforms (e.g., Linux, Android) including some new operating system prototypes like Asbestos [12] or HiStar [49] to precisely trace provenance. They can trace provenance with high precision. But their run-time overhead tends to be on the high end, due to the heavy-weight instrumentation. ProTracer borrows the basic concept of tainting, optimizes it at the unit level to avoid the heavy-weight instrumentation used in existing approaches. Moreover, tainting alone cannot answer how-queries.

Log storage and presentation: Provenance data can be represented as graphs, researchers have done a lot of work [44]–[46] on reducing the size of these graphs by borrowing ideas from Web graph compression and dictionary based encoding. In [6], researchers leverage Mandatory Access Control (MAC) policies to reduce the storage cost of provenance based on the Hi-Fi [36] system. We envision such policies can also be adopted in ProTracer to achieve more sophisticated reduction. G^2 [18] stores logs in databases, and provides execution graphs that can be analyzed using LINQ queries or user-defined programs. However, it depends on printed messages by the applications. Some techniques [9], [28] try to reduce events offline. Since ProTracer performs online reduction, the whole trace is not visible to ProTracer. Some reduction that is easy for offline analysis cannot be applied online. In other words, these offline reduction techniques are complementary to ProTracer.

Log integrity: In [19], researchers proposed a real-time server/client audit model. The client sends integrity-assured log to the server side for post-mortem detection of infections. ProTracer can provide pre-analyzed logs with small size, which help [19] gain more accurate results and better performance with lower network traffic. In [43], researchers proposed a primitive that provides the integrity of execution trace. It works on instruction-level execution traces. The same idea can be applied in ProTracer to guarantee the integrity of the log and provide better attack resilience. Recently in [7], researchers propose a novel generic framework for the development of provenance-aware systems based on LSM to secure such systems. Other researchers also try to enhance the storage system to provide the integrity of the provenance data. For example, [41] suggests using an isolated versioning system – working at the disk level – to store provenance; [38] develops a storage system that allows repetitive reads but only a single write to guarantee data integrity.

VIII. Conclusion

We develop ProTracer, a cost-effective provenance tracing system that features the capabilities of alternating between logging and unit-level taint propagation, and event processing through a lightweight kernel module and a sophisticated concurrent user space daemon. Our evaluation results show that ProTracer substantially improves the state-of-the-art. In our experiments, it only generates 13MB audit log per day, and 0.84GB(Server)/2.32GB(Client) in 3 months with less than 7% overhead, while the generated logs do not lose any attack-related information.

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 award 1409668, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. 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] Apache benchmark. https://httpd.apache.org/docs/2.2/programs/ab.html.

[2] Irs phishing email. https://www.spamstopshere.com/blog/spam-news/alert-irs-scam-email-links-malicious-code.

[3] Proftp backdoor. http://www.osvdb.org/69562.

[4] Student hacker. http://www.huffingtonpost.com/2014/03/05/student-hacking_n_4907344.html.

[5] Tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.

[6] Bates, A., Butler, K. R., and Moyer, T. Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs. TaPP'15.

[7] Bates, A., Tian, D. J., Butler, K. R., and Moyer, T. Trustworthy whole-system provenance for the linux kernel. Usenix Security'15.

[8] Bosman, E., Slowinska, A., and Bos, H. Minemu: The world's fastest taint tracker. RAID'11.

[9] Braun, U., Garfinkel, S., Holland, D. A., Muniswamy-Reddy, K.-K., and Seltzer, M. I. Issues in automatic provenance collection. In *Provenance and annotation of data*.

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

[11] Clause, J., Li, W., and Orso, A. Dytan: A generic dynamic taint analysis framework. ISSTA '07.

[12] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazieres, D., Kaashoek, F., and Morris, R. Labels and event processes in the asbestos operating system. SOSP '05.

[13] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. OSDI'10.

[14] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS) 32* (2014).

[15] Gehani, A., and Tariq, D. Spade: Support for provenance auditing in distributed environments. Middleware'12.

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

[17] Groth, P., Miles, S., Fang, W., Wong, S. C., Zauner, K. P., and Moreau, L. HPDC'05.

[18] Guo, Z., Zhou, D., Lin, H., Yang, M., Long, F., Deng, C., Liu, C., and Zhou, L. G^2: A graph processing system for diagnosing distributed systems. USENIX ATC'11.

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

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

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

[22] Kang, M. G., McCamant, S., Poosankam, P., and Song, D. Dta++: Dynamic taint analysis with targeted control-flow propagation. NDSS'11.

[23] Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D. Libdft: Practical dynamic data flow tracking for commodity systems. VEE '12.

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

[25] King, S. T., and Chen, P. M. Backtracking intrusions. SOSP '03.

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

[27] Lee, K. H., Zhang, X., and Xu, D. High accuracy attack provenance via binary-based execution partition. NDSS'13.

[28] Lee, K. H., Zhang, X., and Xu, D. Loggc: garbage collecting audit log. CCS '13.

[29] Ma, S., Lee, K. H., Kim, C. H., Rhee, J., Zhang, X., and Xu, D. Accurate, low cost and instrumentation-free security audit logging for windows. ACSAC '15.

[30] Masri, W., Podgurski, A., and Leon, D. Detecting and debugging insecure information flows. ISSRE '04.

[31] McCamant, S., and Ernst, M. D. Quantitative information flow as network flow capacity. PLDI '08.

[32] Miles, S., Groth, P., Branco, M., and Moreau, L. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*.

[33] Muniswamy-Reddy, K.-K., Braun, U., Holland, D. A., Macko, P., Maclean, D., Margo, D., Seltzer, M., and Smogor, R. Layering in provenance systems. USENIX ATC'09.

[34] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. I. Provenance-aware storage systems. Usenix ATC '06.

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

[36] Pohly, D. J., McLaughlin, S., McDaniel, P., and Butler, K. Hi-fi: Collecting high-fidelity whole-system provenance. ACSAC '12.

[37] Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. MICRO 39.

[38] Sion, R. Strong worm. ICDCS'08.

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

[40] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. Bitblaze: A new approach to computer security via binary analysis. ICISS '08.

[41] Sundararaman, S., Sivathanu, G., and Zadok, E. Selective versioning in a secure disk system. Usenix Security'08.

[42] Tak, B. C., Tang, C., Zhang, C., Govindan, S., Urgaonkar, B., and Chang, R. N. vpath: precise discovery of request processing paths from black-box observations of thread and network activities. USENIX ATC'09.

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

[44] Xie, Y., Feng, D., Tan, Z., Chen, L., Muniswamy-Reddy, K.-K., Li, Y., and Long, D. D. A hybrid approach for efficient provenance storage. CIKM '12.

[45] Xie, Y., Muniswamy-Reddy, K.-K., Feng, D., Li, Y., and Long, D. D. Evaluation of a hybrid approach for efficient provenance storage. *ACM Transactions on Storage (TOS) 9*, 4 (2013), 14.

[46] Xie, Y., Muniswamy-Reddy, K.-K., Long, D. D., Amer, A., Feng, D., and Tan, Z. Compressing provenance graphs. TaPP'11.

[47] Xu, K., Xiong, H., Wu, C., Stefan, D., and Yao, D. Data-provenance verification for secure hosts. *Dependable and Secure Computing, IEEE Transactions on 9*, 2 (2012).

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

[49] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in histar. OSDI '06.

[50] Zhang, H., Yao, D. D., and Ramakrishnan, N. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. ACSAC'14.

[51] Zhang, M., Zhang, X., Zhang, X., and Prabhakar, S. Tracing lineage beyond relational operators. VLDB '07.

[52] Zhu, N., and Chiueh, T.-c. Design, implementation, and evaluation of repairable file service. DSN'13.