RAIN: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking

Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini,
Taesoo Kim, Alessandro Orso, and Wenke Lee
Georgia Institute of Technology

ABSTRACT

As modern attacks become more stealthy and persistent, detecting or preventing them at their early stages becomes virtually impossible. Instead, an attack investigation or provenance system aims to continuously monitor and log interesting system events with minimal overhead. Later, if the system observes any anomalous behavior, it analyzes the log to identify who initiated the attack and which resources were affected by the attack and then assess and recover from any damage incurred. However, because of a fundamental tradeoff between log granularity and system performance, existing systems typically record system-call events without detailed program-level activities (e.g., memory operation) required for accurately reconstructing attack causality or demand that every monitored program be instrumented to provide program-level information.

To address this issue, we propose RAIN, a Refinable Attack Investigation system based on a record-replay technology that records system-call events during runtime and performs instruction-level dynamic information flow tracking (DIFT) during on-demand process replay. Instead of replaying every process with DIFT, RAIN conducts system-call-level reachability analysis to filter out unrelated processes and to minimize the number of processes to be replayed, making inter-process DIFT feasible. Evaluation results show that RAIN effectively prunes out unrelated processes and determines attack causality with negligible false positive rates. In addition, the runtime overhead of RAIN is similar to existing system-call level provenance systems and its analysis overhead is much smaller than full-system DIFT.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Information flow control**; **Intrusion detection systems**; • **Applied computing → System forensics**; **Surveillance mechanisms**; **Investigation techniques**;

KEYWORDS

attack provenance; record and replay; information flow analysis; forensic analysis

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Since modern, advanced attacks are sophisticated and stealthy, collecting and analyzing attack provenance data has become essential for intrusion detection and forensic investigation. For example, many attack investigation or provenance systems monitor and log interesting system events continuously to identify which process interacted with an unknown remote host and which process accessed or modified sensitive files. If the systems find such a suspicious process, they will analyze its previous behaviors to determine whether it was attacked and which resources were affected by it.

Attack investigation systems, however, entail a practical limitation because of their two most important but conflicting goals—collecting a detailed log and minimizing runtime overhead. To ensure an accurate attack investigation, an instruction-level log would ideally record the execution of all of the CPU instructions of all programs. Nevertheless, such systems [49, 50, 54] also incur tremendous runtime overhead (4×–20×), so they are impractical in real computing environments. Therefore, as many attacks eventually need to use system calls to access sensitive resources and devices, other practical systems [12, 24, 33, 34] mainly focus on system-call information, the collection of which incurs low runtime overhead (below 10%).

Although system-call-based investigation systems are practical, they suffer from dependency ambiguity and explosion [33] because it is difficult to reconstruct accurate attack causality with only system-call information. For example, when a process reads from a number of sensitive files and sends some (encrypted) data to a remote host, knowing which sensitive files the process sends (or it might not send any sensitive data) without instruction- or memory-level data-flow tracking that system-call-level log cannot provide becomes a challenge. To overcome this limitation, several systems [12, 23, 33, 34] instrument monitored programs to obtain interesting program-level information by modifying their source code or rewriting their binary code. Nevertheless, this approach is not scalable; that is, it must instrument each program again whenever it is updated. More importantly, it cannot cover dynamic code execution (e.g., code injection, self-modifying code, and return-oriented programming), which is frequently used by exploits.

This paper proposes RAIN, a practical **R**efinable **A**ttack **IN**vestigation system, that selectively provides an instruction-level detailed log while minimizing runtime overhead. RAIN satisfies these conflicting goals using a *system-call-level record-and-replay technology* and *on-demand dynamic information flow tracking (DIFT).* RAIN continuously monitors and logs system-call events and additional data for later replay while constructing a logical provenance graph. When it detects any anomalous event in the graph, it performs replay-based DIFT from the event to prune out any unwanted

dependency. Performing DIFT for every process in the graph, however, is infeasible because the overhead of DIFT is too high (usually around 10×–20× and at best, 2.7×, using decoupling techniques [27, 37]). Instead, RAIN performs system call-level *reachability analysis* to extract a subgraph tightly related to the anomaly and then conducts DIFT only for processes belonging to the subgraph.

We evaluated RAIN using the red team exercises produced by the DARPA Transparent Computing program [2] and a recent real attack (StrongPity). These cases include normal background traffic with complex programs such as the Firefox web browser. Evaluation results show that RAIN is able to capture fine-grained causalities that accurately uncover the behaviors and effects of attacks, and in most cases the false positive rate is negligible. The runtime overhead is as low as 3.22% on SPEC CPU 2006, unlike previous instruction-level investigation approaches [49, 50, 54] whose runtime overhead is 4×–20×. Further, RAIN effectively reduces the number of processes to be replayed with DIFT and filters out, on average, more than 90% of processes. This is a considerable improvement in performance over the previous approach, which has to replay the entire system and conduct DIFT against *all* processes.

Motivating Example. To illustrate the constraint of existing provenance systems and the contribution of our work, we refer to a recent attack called *StrongPity* [13]. The attack infected over 1,000 systems in Italy and several other European countries in late 2016. The purpose of StrongPity was to steal and tamper with the victims' data by means of compromised data transfer or archiving tools. Take, for example, Alice, a finance manager who maintains and manages contracts and bidding files. Alice usually uses a popular `ftp` extension called `FireFTP` in her Firefox browser to transfer files to other hosts, such as a machine hosting the shared folder for her team. In the first step of the attack, the extension in Alice's Firefox is upgraded to one that contains a backdoor, resulting from the distribution site of `FireFTP`, which has been compromised. A malicious extension accesses Alice's file system, collects data from certain files, and sends the data to an attacker's controlled site. In addition, the extension modifies certain incoming files before they are saved which also pollutes files that rely on the modified ones.

As we mentioned earlier, conventional system call-level tracing and auditing cause false positives in damage assessments when the source of the program (`Firefox` and `FireFTP` extension) is compromised and source instrumentation (if any) becomes untrustworthy. For example, the system call traces in Figure 1 indicate data leakage by connecting any `read` system call from sensitive files to the `send` system call directed to the malicious site. Many of these flows may be spurious if the user-space browser does not actually propagate the data from the file to the remote host (i.e., not all of the files being read are actually leaked). Similarly, many of the processes and files indirectly affected by interactions with the tampered file may not actually be affected at all. One needs to track the user-space data flow to precisely identify these dependencies.

With RAIN, after discovering that a host is controlled by an attacker, an investigator performs an upstream analysis originating from the host. With data pruning and selective DIFT, RAIN returns a provenance subgraph that contains the exact data leakages to the host. Although the malicious extension reads a number of files, it leaks only a small portion of them. By providing accurate analysis,

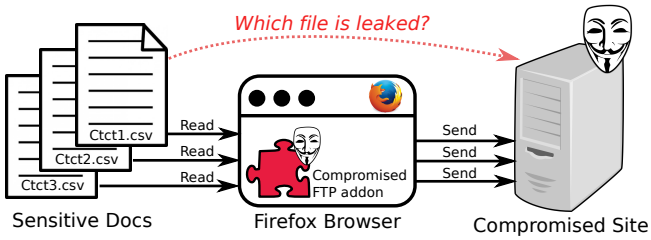


Figure 1: Example of causality inaccuracy (i.e., dependency explosion) in system-call-level provenance data.

RAIN saves the company from the fear of a large scale data leakage. We will revisit this example and elaborate on the details of the analysis in the following sections.

We summarize our contributions as follows:

- **A refinable attack investigation system.** We propose a new attack investigation system that efficiently records system-wide events in terms of system calls during runtime and refines the log with DIFT during replay to recover fine-grained causality. RAIN satisfies two conflicting yet important requirements: low runtime overhead and fine-grained causality information (at the CPU instruction level), both essential in the forensic analysis of attacks.
- **On-demand inter-process DIFT.** Instead of applying DIFT to whole-system events [49, 50, 54] which introduces tremendous overhead or which is likely infeasible, we introduce graph-based reachability analysis to filter out unrelated processes and selectively perform DIFT which makes inter-process DIFT feasible for attack investigation.
- **Accurate and comprehensive attack investigation.** We improve the accuracy of object-object, object-process, process-process causalities (§5.2) and significantly reduce the false positive rates generated by previous systems.

The rest of paper is organized as follows: §2 describes our threat model. §3 provides an overview and describes the architecture of RAIN. §4 describes system logging and record-replay techniques, and §5 explains the provenance graph. §6 presents the reachability analysis and the process of identifying triggers. §7 describes how RAIN performs selective DIFT, and §8 summarizes its implementation and presents the results of evaluation. §9 discusses limitations and future work, §10 summarizes related work, and §11 concludes.

2 THREAT MODEL AND ASSUMPTIONS

Our threat model is similar to those proposed in previous system provenance studies [12, 34, 40]—an OS and monitoring system are a trusted computing base (TCB). We take, for instance, an attacker who tries to attack the applications and resources of a system protected by RAIN and whose main goal is to exfiltrate sensitive data kept in the system or manipulate it to propagate misinformation. To achieve this goal the attacker may install malware on the system, exploit a running process, or inject a backdoor.

To realize a practical, refinable attack investigation system we assume the following: First, we assume that all of the attacks against the system begin after RAIN is deployed—that is, RAIN begins recording all of the attacks from their inception. Hardware trojans and OS

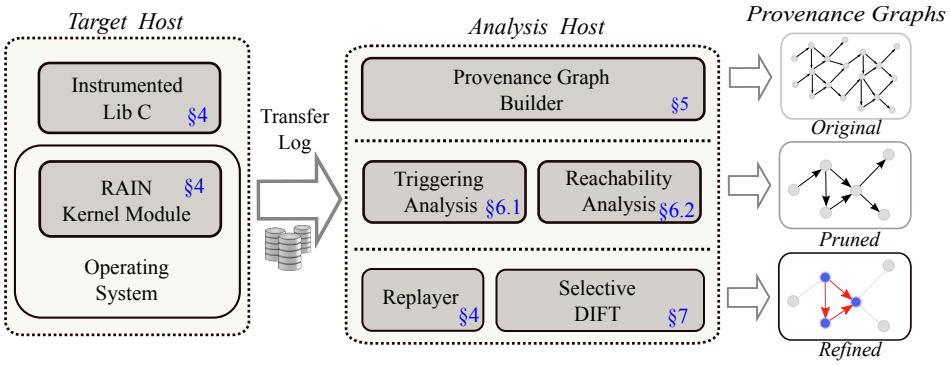


Figure 2: Overview of RAIN architecture.

backdoors are out of scope of this paper. Second, we assume that although an attacker could compromise the OS or RAIN itself, the attacker has no way of manipulating the previous provenance data containing attack attempts on the OS or RAIN. That is, although we can see the attacker attempting to compromise the OS or RAIN, any data recorded after a successful attack may not be reliable. In the future, RAIN could ensure data integrity by using previous secure provenance logging techniques [11, 55] and managing the provenance data in a remote analysis server. Also, by using state-of-the-art integrity-checking mechanisms [25, 32, 38, 43, 47], RAIN could determine when such an incident has occurred. Another assumption that we make is that an attacker uses only explicit attack channels, not side and covert channels which are beyond the scope of this paper. Although RAIN does not yet have a solution stopping these attacks, we believe a record-and-replay approach has the potential to detect attacks as shown in [14, 53].

Note that although some instruction-level attack investigation systems [50, 54] are capable of detecting attacks against an OS, they are too slow to be used in a real computing environment and are mainly applied for in-depth malware analysis, running a small number of samples in a controlled environment. Thus, we only assume integrity-checking mechanisms in this paper.

3 OVERVIEW

This section presents an overview of RAIN, a record-and-replay-based system that efficiently logs the whole-system events during runtime and conducts DIFT during replay to accurately determine fine-grained causal relationships between processes and objects (e.g., files and network endpoints) created during the execution of user-level processes.

Figure 2 represents the architecture of RAIN, which consists of two main components: the target host and the analysis host. In the target host, RAIN's kernel module logs all system calls that user-level processes have requested, including the return values and parameters that RAIN will use to generate a provenance graph. RAIN also records the execution of user-level processes by using kernel modules and an instrumented `libc` library to replay the processes later on. It collects all necessary information to reproduce the complete architectural state of user-level processes (i.e., all non-deterministic values including random numbers). The target host then sends the system call and record logs to the analysis host (§4).

In the analysis host, the provenance graph builder consumes the received system call log to construct a *coarse-grained* whole-system provenance graph that contains many security-insensitive causality events (§5). To refine the coarse-grained provenance graph, RAIN first detects *triggering points* representing suspicious events in the graph (e.g., accessing a sensitive file). Next, it initiates a *reachability analysis* (i.e., upstream, downstream, and point-to-point analyses) from the triggering points to create security-sensitive provenance subgraphs (SPS), which consist of basic units that replay with DIFT (§6). While selectively performing DIFT, the replay engine of RAIN replays each SPS to construct fine-grained causality subgraphs (§7). Lastly, with the fine-grained causality subgraphs, RAIN refines the original whole-system provenance graph to detect the true behavior and damages of any sophisticated attack that we were not able to observe in the original provenance graph.

4 REPLAY-ABLE SYSTEM LOGGING

4.1 System Logging

The system logging component resides inside the kernel of the operating system as a kernel module. We hook the system call table to intercept the arguments and return values of causality-related system calls. The component logs the semantics of system calls between kernel objects and events such as `open`, `read`, `write` file operations and `connect`, `recv`, `send` network operations. We also include essential semantics such as the file path and the file descriptor in the `open` syscall.

To uniquely identify the object, we log the related kernel semantics of processes and files, which include `inode`, `major`, `minor`, `gen` for files, and `pid`, `tgid` for processes. We also refer to kernel data structures if necessary (e.g., to get the string of a file's path from the file system structure, `dentry`) which enables us to reduce the log size for constructing the provenance graph to focus on unique processes and objects. We use the `relayfs` ring buffer to efficiently transfer the system call logs from the kernel to the user space. The logs are packed, compressed, and transmitted off of the target host to a security-assured analysis host.

4.2 Enabling Replay-Ability

Compared to previous system logging schemes [4, 12, 31, 34], RAIN not only logs semantics for building the coarse-grained provenance graph, but also the non-determinism that enables faithful replay.

For this purpose, we reuse Arnold, the open-source framework of a process-level record replay technique [19]. As an advantage, Arnold supports the independent replay of processes so we do not have to replay the entire system's execution (e.g., [20, 50]) for analysis. Rain extends Arnold so that it supports *system-wide* recording, which accounts for every process execution in user-space. As a result, Rain can replay *any* part of the system's execution on-demand for selective DIFT without losing completeness.

To replay the execution faithfully, Arnold records the return value of system calls, IPC communications such as signal and system V queues, and caches the data for every file or network I/O system call. For multi-threading applications, the `pthread` library in `libc` is hooked to record and enforce the order of the thread switch. To handle shared memory, Arnold replays involved processes cooperatively to regenerate shared data, and to improve the replay reliability. It also records and replays random numbers as well as `RDTSC` by using `prctl(PR_SET_TSC,PR_TSC_SIGSEGV)` from [5].

Rain aims to detect and analyze attacks that may have previously gone undetected, so instead of being confined to a predefined list of known programs, Arnold's process level record-replay technique has been extended from a single process to *system-wide* executions. Although Rain does not replay all of the recorded executions for refining attack provenance owing to the reachability analysis (§6) and selective DIFT (§7), this *system-wide* feature is critical as it enables us to refine *any* demanded part of execution. We achieve this by hooking the `execve` syscall inside of the kernel so that when a program is loaded via `execve`, we force it to become a recording process by creating the recording contexts. Throughout the analysis Rain can replay each demanded execution independently to resolve the associated fine-grained causality.

4.3 Storage Footprint

The storage use of Rain comes from system logging and recording. We serialize the logging data using the Apache Avro [1] binary format, which incurs around 500MB–2GB per day for desktop use in our experiments. On the side of recording, the file, network I/O cache constitutes most of the storage cost. To optimize storage use, Arnold [19] applies compression, caches the data using "Copy-on-RAW", and manages the data pieces in a B-tree. In our experiment, the record logs of system-wide executions (excluding the OS) on a desktop produce around 2GB of storage per day. Therefore, the storage cost is 2.5–4GB per day (or less than 1.5TB per year). With the market price for a 2TB hard drive or cloud storage being around 50 US dollars, we believe that our storage cost is both reasonable and affordable. Note that instead of selectively storing data [34], we choose to store all of the raw data first and then generate the provenance graph selectively, following a set of pruning algorithms (§6). Our storage footprint reflects the size of raw data.

5 PROVENANCE GRAPH

We construct a graph structure called *Provenance Graph* which contains the whole-system execution during the entire period of monitoring. Rain uses this graph as the basic model to represent system objects, events, and their causal relationships. We begin by processing the syscall logs. When first constructed from system logging, the graph is coarse-grained. It is then pruned and refined incrementally according to analysis requests. We use nodes to represent system objects and edges to represent causality between system objects.

Causality	Granuality
Process-Object	Coarse level
Object-Process	Coarse + Fine level
Process-Process	Coarse + Fine level
Object-Object	Fine level

Table 1: Granularity of analysis.

5.1 Nodes

Nodes in the provenance graph can be classified into two categories: processes and objects. Processes represent user-level processes. Objects represent files and network endpoints. All nodes contain a timestamp that represents the time that the entity (represented by the node) was generated by the operating system. Each process node contains `pid`, `tid`, and process name. Each File node contains the full path name of the file and `inode`, `major`, `minor`, and `gen` which uniquely identify a file. Nodes representing network endpoints contain the IP address and port of the network entity. Note that the tracking scope of Rain does not extend beyond the target host. In other words, we treat the remote host as a black box and do not track its internal logic or state. This limitation can be addressed if we apply Rain on the remote host and causally relate the two hosts, but we will explore this issue in future work.

5.2 Causality Edges

Edges represent causal relationships between nodes in the provenance graph. We define four types of edges: process-object causality, object-object causality, object-process causality, and process-process causality. Among these causalities, we observe that only the process-object causality can be tracked reliably by syscall-level logging. The remaining causalities require either full or partial fine-grained user-space tracking. We summarize these granularity requirements in Table 1.

Process-Object Causality. Process-object causality, which denotes the causal relationship between a process and an object, is established when a user-level program accesses a file or network object. As the operating system provides these I/O services to user-space, this causality can be captured by syscall logging (i.e., file or network I/O) without false negatives. For example, the data can be loaded from a file to the memory of process via a `read` syscall or written to a file from the process via a `write` syscall. In the case of a `read` syscall, the process has a directional edge to a file; and the case of a `write` syscall, a file node has an edge to a process. Similar causalities exist between processes and network endpoints. In the case of a `mmap` syscall, the direction of causality is determined by the syscall arguments such as `PROT` and `FLAG`.

Object-Process Causality. Object-process causality is established when the object affects the execution of a process or its control flow. Usually the process has an edge from an executable file if that file is loaded and executed by the process. Rain captures this causality by monitoring the `execve` syscall. The use of libraries is

typically tracked by `open`, `mmap`, or `dlopen` syscalls. However, this causality may not be true by just analyzing syscalls. For example, the developer may include libraries but not use them. To affirm causality between processes and libraries, we need to further track the control flow to see if its value (address) exists within a library's address space such as that described in [41]. Particularly for sophisticated attacks, an accurate object-process causality is crucial for detecting control-flow hijacking and identifying the sources of exploit payloads that could be used to access library functions. We accurately determine this type of causality during the DIFT phase.

Object-Object Causality. Object-object causality occurs in the case of data flow between two objects. The data flow inside of a process starts from an *inbound* object (e.g., via `read` syscall) and ends at an *outbound* object (e.g., via `write` syscall). One can infer this type of causality by simply pairing inbound and outbound objects. However, simply monitoring file or network I/O syscalls (e.g., our motivating example in §1) or statically analyzing on the program is inaccurate because we need to track the data propagation in the user-level execution. As the dynamic taint analysis is prohibitively expensive, RAIN tracks the Object-object causalities during replay (§7).

Process-Process Causality. Process-process causality is based on the relationships between two processes. Processes can be causally related if one is cloned by the other via the `clone` syscall, or they could be related because of inter-process communications (IPC). Some IPCs (e.g., pipe, message queues, and semaphores) can be observed from syscalls. However, causality cannot be accurately determined in the case of shared memory. Even though a `mmap` or `shmget` syscall indicates the establishment of an IPC channel, it does not necessarily mean data was actually exchanged between processes. To track such causality, RAIN relies on DIFT to monitor data propagation among memory operations.

5.3 Graph Construction

We construct the provenance graph by linking nodes and edges according to the above causality definitions. Our construction is based on uniquely identifiable objects. Since `pid` and `inode` are recycled if a process is terminated or file is deleted respectively, we use `path` as a unique identifier of processes and files (as nodes) since the collision of these objects with the same name is low in practice. After being processed from system call logs, each entry of a node or an edge is compressed and stored in a binary format. When requested for analysis, those within in the time frame are converted and imported into a graph database. In particular, we used Neo4j [6].

Semantic-Preserving Aggregation. Naturally, edges from I/O events such as `read` and `write` constitute a large portion of the graph. However, many are called successively indicating a single "large" `read` or `write` syscall execution. Therefore, we aggregate these successive calls for conciseness as inspired by [52]. For example, we merge two `read` syscalls as long as no other file system call occurs between them (e.g., a "write" to the same file). Note that we collapse these successive syscalls in an "indexing" style so that we do not remove the unique semantics of each individual edge. Thus, the selective DIFT still has the flexibility to perform taint tracking between desired I/O syscalls. The aggregation alleviates the traversing and storage costs of edges by 10%–50%.

6 COARSE-LEVEL PRUNING

After constructing the coarse-grained provenance graph, we prune it to generate a *security-sensitive provenance subgraph* (i.e., SPS) in two steps: a triggering analysis and a reachability analysis that uses the results of the triggering analysis. The SPS will be used as the target for selective DIFT, in which fine-grained causalities will be resolved.

6.1 Triggering Pinpoint

In the initial provenance analysis, we apply a set of methods to scan the logs and identify suspicious (i.e., *security-sensitive*) processes, events, and objects. The triggering analysis relies on three approaches: external signals, security policies, and customized comparisons. We perform the analysis offline by examining the provenance graph. This process can be done earlier when system call logs are available as with conventional intrusion detection systems [7–9].

External Signals. External signals are notifications from partners or third parties (e.g., an anti-virus company). For example, an analyst may receive advice from an anti-virus vendor to specifically check for the existence of certain executable files. All events performed by these executables can be labeled as triggering points. In our motivating example, the victim receives a notice that the distribution site of the FTP extension was compromised. This triggers an analysis of all behaviors of the browser starting from the malicious extension update.

Security Policy. Security policy checking also serves as a triggering pinpoint method. Based on administrative security policies, we create a set of policies that define *concerning* events used as *triggering points* for analysis. These policies include processes interacting with sensitive files or a sequence of events that deviates from the typical pattern of system calls. For example, it is a violation that a process reads from certain sensitive files and then sends read data to an unknown remote host. Recent research has shown that the detection of attacks based on system-call sequence analysis can be improved with machine-learning techniques [35].

Customized Comparisons. We sometimes need to compare the states of objects at different times or locations to identify suspicious points. Take the data-tampering case in our motivating example. In order to identify files that have been tampered with, we compare the files (e.g., by comparing their hash digests) that have been downloaded via the browser extension to the original version of these files. If they differ, the *tampered* files are used as triggering points. This type of comparison typically requires application-specific semantics which are useful but not the focus of this work. For example, Gyrus [26] compares user interface (UI) inputs and network outbound traffic to determine user intention discrepancies.

6.2 Reachability Analysis

Starting from the identified triggers, we perform a reachability analysis to become aware of the *potential* original source(s) and impacts. This analysis prunes out unrelated executions and enables

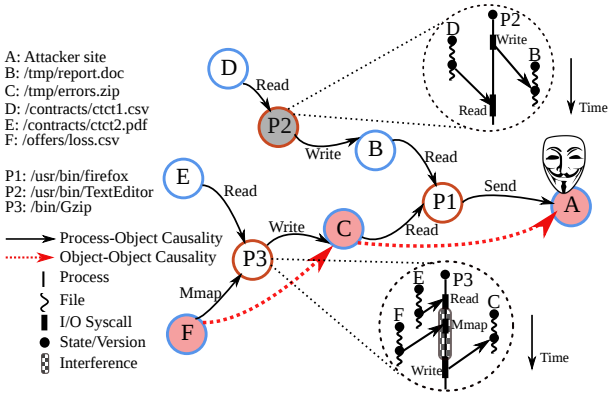


Figure 3: Coarse-level pruning and fine-level refinement on the motivating example. The gray shaded node indicates that process P2 is pruned from the SPS because of the negative interference between files B and D. The red-shaded nodes represent files on the causality path from origin F to attacker site A via file C.

our DIFT to focus on resolving fine-grained causality in the attack-related executions. Although our DIFT is performed offline, the high cost of DIFT is not eliminated but *migrated* which is also pointed out by [27]. Hence we argue that it is still impractical to perform full taint tracking, even if it is performed offline. With reachability analysis, we set boundaries on the DIFT, avoiding tainting "dead" branches or regions of the graph.

The reachability analysis extends the triggering points to uncover possible upstream origins, downstream impacts, and causality paths between two points. Even though at this stage the graph includes only partial causalities (§5.2), computing the SPS on top of it and pinpointing the part that desires further DIFT is sufficient for capturing the the remaining causalities. With the SPS we can efficiently perform DIFT with a clear scope rather than the whole graph. We present the analysis interface in Algorithm 1 and the graph traversing algorithms in Algorithm 2.

6.2.1 Upstream and Downstream Pruning. Upstream pruning *reversely* scans the provenance graph from the triggering point and prunes out unrelated nodes and edges. The analysis follows the information flow and time sequence to extend the subgraph such that $Subject \rightarrow (write/send) \rightarrow Object \rightarrow (read/recv) \rightarrow Subject$. For example, in Figure 3, from the attacker's site (node "A"), we scan the `send` or `write` events to A; after finding Firefox (node "P1"), we further scan `read` or `recv` events that P1 performs earlier. For the shared memory case in process-process causality, we also scan for the syscall events that establish the IPC (e.g., `shmget` and `mmap`) and extend the SPS to that process. For downstream pruning we check forward events resulting from the suspicious processes and other files that are affected afterwards. Pruning also follows the information flow and extracts the downstream SPS. Pruning is either naturally bounded (meaning no more related causality is found) or bounded by the network interface.

6.2.2 Point-to-Point Pruning. Point-to-point analysis indicates *whether* and *how* two points are causally related in the graph. This analysis works on top of upstream and downstream pruning. Given two points, we first perform downstream pruning from the earlier point until the later point timestamp. Second, from the later point we perform upstream pruning until the timestamp of the early point. Then we inspect the two resulting subgraphs to identify the intersection set, called *meeting* nodes (FindMnodes() in Algorithm 2). Along with pruning, each point maintains the tags of its parent and ancestor nodes. Finally, we use the tags of meeting nodes to construct the full paths (i.e., SPS) (RecoverPaths() in Algorithm 2). The remaining causalities along the paths will be captured by the selective DIFT.

Algorithm 1 Coarse Level Pruning Interface

```
function UpStrmPru(TPoint)
    GTraverse(TPoint,UP)
function DownStrmPru(TPoint)
    GTraverse(TPoint,DOWN)
function PtP(TPoint_1,TPoint_2)
    GTraverse(TPoint_1,DOWN)
    GTraverse(TPoint_2,UP)
    Mnodes ← FindMnodes(up_nodes,down_nodes)
    RecoverPaths(Mnodes)
```

Algorithm 2 Information Flow Based Graph Pruning

```
Require: up_nodes, down_nodes
function regist_interference(ne,na)
    if ne ∈ {read, recv} then
        if ne.timestamp ≤ na.timestamp then
            add_interference(ne,nd)
    else if ne ∈ {write, send} then
        if ne.timestamp ≥ na.timestamp then
            add_interference(ne,nd)
function GTraverse(TPoint,Direction)
    ngb_nodes ← read_graph(TPoint.uid)
    for node ∈ ngb_nodes do
        if Direction = UP then
            if TPoint.type = Process then
                if node ∉ {read, recv} then
                    continue
            else if TPoint.type ∈ {File, Host} then
                if node ∉ {write, send} then
                    continue
            up_nodes(node) ← TPoint
            regist_interference(node,TPoint)
            GTraverse(node,UP)
        else if Direction = DOWN then
            if TPoint.type = Process then
                if node ∉ {write, send} then
                    continue
            else if TPoint.type ∈ {File, Host} then
                if node ∉ {read, recv} then
                    continue
            down_nodes(node) ← TPoint
            regist_interference(TPoint,node)
            GTraverse(node,DOWN)
```

6.2.3 Data Interference in Memory Space. We further look into the system call sequence of each process execution. We observe that for object-to-object causality, the inbound object can be causally related to the outbound object only if their existence ranges overlap in the process memory space. We call these overlaps "interferences," which fortunately can be identified in the system call sequence. By examining interference situations, we skip performing DIFT in the case of non-interference. For example, in Figure 3, the outbound "report.doc" file (node B) has no interference with the inbound "contract1.csv" file (node D) because the `read` takes place after the `write`. Therefore, the interference analysis rules out the necessity of doing DIFT for the "TextEditor" process (node P2). Meanwhile, the "Gzip" program (node P3) is an positive interference example, as both of the two upstream files ("contract2.pdf" (node E) and "loss.pdf" (node F)) once shared memory space with the outbound "errors.zip" file (node C).

It is this interference situation in the memory space that leads to possible data propagation (exchange between objects), which we later identify using DIFT. By identifying the exact interference situation of each process execution, we become aware of the part of the execution that requires fine-grained refinement as well as a source and a sink. In the DIFT, we fast-forward the replay to the start of interference (e.g., a `read` syscall), and then early-terminate at the sink. Each entry of interference includes the process, the first syscall that reads the inbound file, the last syscall that writes to the outbound file, the inbound file, and the outbound file. To keep track of ordering, the timestamps of inbound and outbound syscalls are logged. As it is most effective when interference occurs at a late execution time or when it is short, we can skip most of DIFT. In §7 we show how to classify associated files with interference into groups so that one pass of DIFT is able to resolve all of the causalities in the group.

7 SELECTIVE CAUSALITY REFINING

To further refine the graph and obtain fine-grained level causalities, we perform selective data tracking on top of the SPS. We re-compute (i.e., replay) user-space executions while performing taint analysis to determine causality in the interference cases (§6.2.3). Our approach entails tainting the bytes loaded to the memory space and tracking the propagation of the tainted bytes at the level of instruction execution. DIFT (or "taint analysis") has been implemented in previous work [18, 27, 29, 37]. We port open-source taint engines to develop our own taint engine that supports object-object, object-process, and process-process causalities (§7.1.2).

Because taint analysis is costly, we find that even offline analysis becomes impractical if we naively perform taint tracking on every process group. Therefore, we aim to minimize the cost of analysis by performing *directional taint tracking* in each group, orchestrating process groups for *information flow-based tainting* for upstream and downstream analyses, and reusing the taint results to avoid duplicate tainting. Rain is able to track fine-grained causality along the *upstream* and *downstream* paths in the SPS according to the causality results in every branch. For each process group we locate the exact target of taint tracking according to the data interference of the presence of objects in the memory space (§6). We present how we conduct taint in every process group in §7.1 and how we orchestrate tracking across process groups in §7.2.

7.1 Directional Intra-process Tainting

According to the tainting targets determined in the SPS, Rain performs DIFT starting from the source of interference (§6.2.3) and ending at the sink in every replay of the process execution. In our current prototype we determine causality to occur as long as any byte of the inbound object is propagated to the outbound object.

7.1.1 Interference Aggregation. Instead of running taint tracking on every pair of interferences, we aggregate them so they can be resolved in a single pass of taint tracking. This spares duplicate taints from propagating in the same execution trace. Aggregation takes place in the same process group. Suppose n interferences in the process group are related to the analysis request. We aggregate them by starting from the earliest interference and ending at the latest one. Then we run the taint tracking *one* time instead of n times. For example, in Figure 3, inside the P3 process, we aggregate the interference of files E and C via `read` and `write`, and files F and C via `mmap`, `write` because they belong to the same process P3 in the SPS. Thus we can resolve the causalities within them in one pass of tainting. When the tainting is performed it starts from the `read` syscall until the `write` syscall with the tagging of both E and F files as sources.

7.1.2 Replay and Taint Propagation. To allow taint tracking to work independently from a replayed execution, we adopt the analysis compensation technique from [10], which is able to differentiate the executions of Intel Pin from that of the program. First, no syscall made by Pin will be mixed by the recorded syscalls because the replayer is aware of their occurrences (i.e., it can differentiate between the two). Second, for memory space separation, the record log is scanned for any memory allocations. They will be allocated first so the replayed execution will not be affected by Pin.

Our DIFT engine is a set of Pin tools that reuses the open-source `libdft` [29], `Dytan` [18], and `dtracker` [49] projects for tracking object-object causality. The taint tags propagate on both data and control flow dependencies. Data dependency is tracked by monitoring the read/write memory operations at the instruction level between memories and registers, and control flow dependency comes from indirect branch dependency and incurs a higher overhead. We also implement tools that track object-process, process-process causalities. For cases in which object-process causality cannot be captured by the `execve` syscall, we taint track the data propagations and their impact on the control flow. We determine this causality when the return address or the `eip` register is tainted by the data of an input object (similar to [41]), which unveils typical memory corruption exploits that hijack the control flow. For the shared memory case in the process-process causality, we monitor shared memory-related syscalls (e.g., `shmget` and `mmap`) to map the shared memory among processes so that we are able to track the memory operation of a data transfer from a private memory space of one process (e.g., stack and heap) to the shared memory space, and then to the other processes. Additionally, we track the data propagation from an inbound object in one process to an outbound object in another process via shared memory.

7.2 Orchestrating Taintings Across Processes

In this section, we present how we perform taint tracking across processes according to the SPS. We regard taint tracking inside the process as a block function. To efficiently accomplish DIFT, we apply optimization techniques that minimize the tainting workloads. Specifically we introduce two methods of handling tainting in the *upstream* and *downstream* directions. Finally, we present how we refine causality paths for a point-to-point analysis by verifying the coarse-grained paths one-by-one and reusing the previous results cumulatively.

7.2.1 Downstream Refinement. Downstream refinement is capable of accurately identifying the impact of an attack, which is critical in both forensic analysis and intrusion recovery. Compared to conventional intrusion recovery approaches (e.g., Retro [31] and Dare [30]), Rain produces accurate causality between involved files so the recovery can be performed only in files with true causality which eliminates false positives; otherwise innocent processes will be "re-executed."

Recall that when generating the SPS, Rain also produces a pool of interference entries (§6.2.3) in which potential causalities exist. Starting from a designated point (e.g., a file), we identify the process and interference related to this point and then resolve the fine-grained causality. In the case of object-object causality, we run DIFT on the associated process and determine the outbound object(s) with true causality. From that object, we repeat the procedure to determine further causally related downstream objects bounded by the SPSs.

Take, for example, the data-tampering case in the motivating attack. The SPS reports that the tampered spreadsheet file "agreement.csv" has been later read by the auto-budget script which produces the budget and production plan files. More interestingly, the budget file is then used by the document editor which generates a season report. The triggering point in this case is the tampered spreadsheet file. Further interference entries with the file as *inbound* object will be pinpointed and taint tracking will be performed. We consider this interference situation an entry. Then we conduct taint analysis on the first (closest hop) process to identify the true outbound object and move further downstream making the found outbound object inbound object. As a result, we are able to repeatedly identify the exact downstream causalities and insert them into the provenance graph.

7.2.2 Upstream Refinement. Upstream refinement also begins at the triggering point, but proceeds in the *reverse* order of the execution time. The SPS appears in an acyclic-directed graph shape with the latest point being the triggering point (e.g., the file leaked by the compromised FTP extension). To identify the leaked file and its provenance, we locate the associated process in the SPS. The taint tracking on the replay of the process execution determines the real causal parents so the next rounds of taint tracking are performed only at these parent files. Taint tracking continues recursively until it hits a boundary advised by the SPS. At each branch where multiple inbound objects exist, refinement continues only on the *true* inbound object(s) and ensures that they are outbound objects for the next round of tainting.

In Figure 3, from the attacker's site, we begin running upstream refinement performing tainting at P1 (i.e., the Firefox session). We find that although the "report.doc" file (node B) and the "errors.zip" file (node C) are both inbound, only file C is causally related to the attacker's controlled host (node A). We drop file B and continue refinement on the branch from file C. Again we find that "Gzip" (node P3) has input files "contract2.pdf" (node E) and "loss.csv" (node F), but only file F exhibits causality with file C, so we continue along the F branch. As a result, we eliminate the unnecessary workload of tainting *dead* branches that do not reach the triggering point.

Extending Causality Across Processes. As an optional feature, we keep track of the causality across processes (e.g., in Figure 3, file F to site A). We maintain a *shadow tagging file* for each file that is accessed by more than one process. This tagging file keeps track of the source tag of every byte in the file so that Rain can track the causality between two separate files in different processes. For example, in Figure 3, the shadow tagging file of file C is generated when tainting is performed on P3. The tagging file contains the bytes with causality between files E and F. When we replay and taint track P1, we refer to the tagging of file C and acknowledge that the leaked data includes both files C and F because part of the contents of C originated from file F. Accompanied by upstream refinement, Rain constitutes the result of this Point-to-Upstream analysis that file C has been leaked to the attacker while certain leaked contents of C originate from F. In our current prototype, this feature is optional and on-demand as it incurs higher tainting overhead. Note that tracking the data with tags from previous objects requires full-length tainting from the inbound to outbound objects.

7.2.3 Point-to-Point Refinement. Recall that causality may have been included within the SPS (§6.2.2). With the help of the SPS, heavy DIFT is applied to *verify* the data flow on the path where the fine-grained causalities (e.g., object-object) occur. This filters out many unrelated branches that would have incurred high analysis overhead.

Based on the processes on each path in the SPS, we replay and perform taint tracking on the process groups in the path to verify true causality. From the start to end points along the path, each process group is replayed and taint tracking performed on the specific interference between the inbound and outbound system calls. The inbound object is tagged and the running of the process propagates the tags and monitors whether it hits the outbound object. If it does, verification continues. Otherwise, it terminates and returns a negative result. The verification runs until an end point. If all interferences along the path are positive, we refer to the path as "causality positive." At the end, the refinement returns all the causality-positive paths.

The verification procedure is optimized by reusing the taint results for each group. In the implementation, we store the causality between specific inbound and outbound files in a database. In the remaining verifications, we start by searching the database for existing causality facts. Then we reuse them if possible without performing the same taint tracking again. Because we reuse the results, taint verification takes less time as we verify more paths. Particularly for the point-to-point case, the taint tracking in every process group is optionally run in parallel to accelerate the analysis

Host	Module	LoC
Target host	Kernel Module	2,200 C (Diff)
	Trace Logistics	1,100 C
Analysis host	Provenance Graph	6,800 C++
	Trigger/Prune	1,100 Python
	Selective Refinement	900 Python
	DIFT Pin tools	3,500 C/C++ (Diff)

Table 2: Implementation complexity.

if more computing resources are available. As every process group is independently recorded and independently replay-able, we tag every process group with a *symbolic* tag and will resolve the tag propagation with real tags. The use of more resources with less time consumption represents the cost of this optimization. In this case, refinement time decreases to the level of the longest DIFT.

8 IMPLEMENTATION AND EVALUATION

We implemented a prototype of RAIN in Linux. In the kernel module of the target host, we implemented the system logging logic with comprehensive semantics to build the provenance graph and to support whole-system recording. On the analysis host, we implemented the construction of the provenance graph and trigger/prune methods as well as DIFT that support object-object, object-process, and process-process causalities on top of [18, 29, 49]. The complexity of implementation is summarized in Table 2. We plan to release the source code of RAIN.

Our evaluation addresses the following questions:

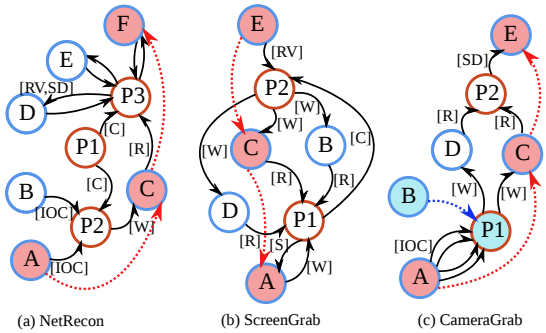
- How well does RAIN detect various attack scenarios (§8.1.1, §8.1.2)?
- How accurately does RAIN prune and refine provenance graphs (§8.1.3)?
- How much overhead associated with analysis, runtime, and storage does RAIN have (§8.2)?

In our evaluation environment, we set up the target and analysis host individually on two bare-metal machines both powered by Intel Xeon(R) CPU W3565 3.2GHz; the target host has an 8GB RAM and 512GB hard drive, and the analysis host has a 32GB RAM and 2TB SSD hard drive. They are connected by a 1GB Ethernet cable. Both machines run Ubuntu 12.04 LTS (i386).

8.1 Security Analysis

Using various attack scenarios, including the motivating example, we evaluate the accuracy gains and conduct a set of red team exercises from the DARPA Transparent Computing program [2].

8.1.1 Motivating Example. We demonstrate the end-to-end procedure and efficacy of RAIN at detecting and analyzing the motivating attack example (§1). The attack exploits the `FireFtp` addon of Firefox to steal a user's data and tamper with downloaded files. At the triggering pinpoint phase of the analysis, the security team of the company was notified that an originally trustworthy site (e.g., white-listed in the firewall policy) was compromised for one week until they confirmed and recovered from the leakage of critical contract details (i.e., *External Signal* in §6.1). In addition, they received complaints from the audit team about abnormal changes in the numbers in a spreadsheet file when they compared the downloaded



(a) NetRecon (b) ScreenGrab (c) CameraGrab

Figure 4: Simplified provenance graph with highlighted accurate causality path (red dotted line for object-object causality and blue dotted line for object-process causality; the nodes on the path are colored accordingly). Notations: (a) A/B: eth0/eth1; C: /tmp/netrecon.log; D/E/F: internal hosts; P1: ImplantCore; P2: NetReCon; P3: Scanner. (b) A: attacker server; B/C/D: screen.png; E: /tmp/.X11-unix/X0; P1: ImplantCore; P2: ScreenGrab. (c) A: /dev/video; B: malicious site that contains CameraGrab payload; C/D: pictures; E: attacker server; P1: Firefox browser; P2: ImplantCore.

spreadsheet file to the original one stored in the archive (i.e., *Customized Comparison* in §6.1). Because of the dependency between documents, the team was also concerned about the impact on other files. With these triggering points, the security team, to accurately determine *what was stolen* and *what was affected* by the attack, queried RAIN for the exact causalities around the compromised site and suspicious files.

To find out what was leaked to the attacker's controlled site, RAIN performs a "Point-to-Upstream" analysis from the site to identify the leaked data. First, RAIN extracts the SPS from the provenance graph by pruning off unrelated nodes and edges and downsizes it to around 20% of the original provenance graph. Then RAIN performs refinement on some of of the process executions, including the Firefox process which communicates with the site. After performing selective refinement, RAIN determines that even though multiple files were accessed, only "ctct1.csv" was leaked. The refinement further reduces the size of the SPS to around 10%, which reveals the few but accurate causalities originating from the malicious site (Figure 3). RAIN also locates the set of files affected by the tampered file as it is used by a finance program to generate reports and other documents. The accurate results generated by RAIN ensure that the company is aware of the scope of data leakage and the impact of the data tampering without panicking or having to carry out unnecessary recovery efforts.

8.1.2 TC Red Team Exercise. We use the set of attack scenarios from the red team exercise of the Transparent Computing (TC) program [2] to continue our evaluation. The attack first installs an *implant-core* on the victim's system via social engineering (e.g., email). After installation, the implant-core communicates with the attacker's host (e.g., the C&C server) and receives and performs future attack tasks. We use four unit attack examples (i.e., NetRecon, ScreenGrab, CameraGrab, and AudioGrab) to demonstrate how RAIN works and what amount of accurate causality is generated.

Analysis Stages		Coarse Level Pruning			Fine Level Refinement					False Positive Rate		
Items		Nodes/Edges			Nodes/Edges		Paths			Coarse%	Fine%	REDUC%
Attacks	Analysis	ProvGraph	SPS	Prune%	Result	Added%	SPS	Result	Added%			
	A(O-Up)		3,024/26,749	15.4%/19.7%	342/2,621	11.3%/9.8%	-	-	-	**67.0%**	**0.0%**	**100.0%**
MotivExp	A(O-Dn)	19,634/135,474	1,822/13,981	9.3%/10.3%	46/336	2.5%/2.4%	-	-	-	**55.6%**	**0.0%**	**100.0%**
(3h02m)	A(O-O)		389/733	1.9%/0.5%	98/222	25.2%/20.2%	51	19	37.3%	**69.1%**	**0.0%**	**100.0%**
	A(O-Up)		2,394/17,691	18.5%/20.5%	198/210	8.3%/11.9%	-	-	-	**70.3%**	**23.4%**	**66.8%**
NetRecon	A(P-Dn)	12,892/86,376	1,234/8,880	9.6%/10.3%	86/799	7.7%/9.0%	-	-	-	**84.7%**	**13.0%**	**84.7%**
(2h38m)	A(O-O)		147/287	1.1%/0.3%	34/66	23.2%/23.0%	12	4	33.3%	**66.6%**	**0.0%**	**100.0%**
	A(P-Up)		1,348/9,189	18.4%/19.8%	156/952	8.2%/7.9%	-	-	-	**90.5%**	**0.0%**	**100.0%**
ScreenGrab	A(O-Dn)	7,327/46,367	895/4,877	12.2%/10.5%	72/351	8.1%/7.2%	-	-	-	**82.1%**	**0.0%**	**100.0%**
(1h13m)	A(O-O)		21/30	0.28%/0.07%	5/4	23.8%/13.3%	9	5	55.5%	**77.4%**	**0.0%**	**100.0%**
	A(P-Up)		1,603/11,102	30.2%/33.2%	96/477	6.0%/4.3%	-	-	-	**32.0%**	**0.0%**	**100.0%**
CameraGrab	A(O-Dn)	5,308/33,440	589/3,317	11.0%/9.9%	59/70	10.5%/2.1%	-	-	-	**29.8%**	**0.0%**	**100.0%**
(39m)	A(O-P)		101/268	1.9%/0.8%	24/59	24.1%/22.0%	9	7	77.7%	**44.2%**	**0.0%**	**100.0%**
	A(O-Up)		992/6,846	20.2%/20.5%	49/232	4.9%3.4%	-	-	-	**39.7%**	**0.0%**	**100.0%**
AudioGrab	A(P-Dn)	4,909/33,382	415/3,394	8.5%/10.1%	31/161	7.4%/4.7%	-	-	-	**48.2%**	**0.0%**	**100.0%**
(30m)	A(P-P)		230/1,392	4.7%/4.2%	84/519	36.5%/37.3%	22	18	81.8%	**29.3%**	**0.0%**	**100.0%**

Table 3: Incremental evolvement with pruning and refining. The Nodes/Edges column of provenance graph shows the whole system executions during this period; Nodes/Edges of SPS tells the size of pruned data in different queries; Prune% gives the portion of pruned data to the original one; In the fine-level refinement stage, we show the number of nodes and the number of added causality edges in Nodes/Edges, along with the Added% rate. Particularly for the "Point-to-Point" analysis, the Paths column tells the number of paths in the SPS and the numbers after refinement. Column FP-Coarse%, FP-Fine%, and REDUC% indicate the false positive rate with coarse-level provenance data, with RAIN refinement, and the reduction percentage.

Schemes	Analysis	1st Point	2nd Point	Time (s)
	A(O-Up)	AttkSvr	-	-
MotivExp	A(O-Dn)	TmpedFile1	-	-
	A(O-O)	CtctA.csv	AttkSvr	305
	A(O-Up)	NR.log	-	-
NetRecon	A(P-Dn)	NetScan	-	-
	A(O-O)	AttkSvr	NR.log	94
	A(P-Up)	ScrnGrab	-	-
ScreenGrab	A(O-Dn)	X11Svr	-	-
	A(O-O)	X11Svr	AttkSvr	68
	A(P-Up)	ImpCore	-	-
CameraGrab	A(O-Dn)	CmrGrab(file)	-	-
	A(O-P)	CmrGrab(file)	ImpCore	38
	A(O-Up)	AttkSvr	-	-
AudioGrab	A(P-Dn)	Firefox	-	-
	A(P-P)	Firefox	RptGen	418

Table 4: Analysis request details.

NetRecon. First, the implant-core clones a process called NetRecon which collects network configuration information that it saves to a temporary file. Second, the implant-core clones another process that scans neighboring hosts based on network configuration information. The triggering analysis finds suspicious collecting behavior by spotting a downloaded file conducting a series of `ioctl` requests SIOCGIFHWADDR and SIOCGIFBRDADDR. The results of the "point-to-downstream" analysis shows that the cloned process reads the temporary file and tries to connect other internal hosts that are determined by the temporary file. In addition, the point-to-point analysis between the "NetRecon.log" and neighboring hosts shows the effectiveness of RAIN involving control flow dependency. Figure 4(a) highlights the key causality between `eth0` and another neighbor host. Additionally, we perform other types of analyses and list the incremental results in Table 3.

ScreenGrab. The implant-core downloads a "ScreenGrab" program that occasionally captures the screenshot of the victim's desktop and selects certain shots to send to the attacker's server. While the attack occurs, the user performs various background desktop actions such as web browsing. Our triggering analysis learns that a site is controlled by the attacker. Starting from the malicious site, RAIN conducts an upstream analysis in order to identify a causal relationship with the triggering point. It begins by extracting the SPS from the provenance graph, and then performs a fine-grained analysis to refine the SPS to obtain an accurate causality subgraph. We can see the executable ScreenGrab has multiple inbound traffic from the X11 server (i.e., via Unix domain socket `/tmp/.X11-unix/X0`) and outbound traffic to a file. Then the implant-core sends this file to the attacker's host. RAIN is able to identify exactly which file is sent. We highlight the SPS with refined causality in Figure 4(b) in red.

CameraGrab and AudioGrab. The victim's Firefox browser is exploited with a zero-day exploit and its control-flow is hijacked to the CameraGrab and AudioGrab payloads. The exploited browser then uses a camera and a microphone to spy on the user's behavior and saves it in images and audio files respectively. Finally, the implant-core selects certain files and sends them to the attacker server. During this process, the user sometimes finds that the LED light on the camera is on despite having no intention of using the camera. The triggering point is `ioctl` syscalls which communicates with the device. To determine the root cause, we perform "point-to-upstream" analysis to check for the specific object-process causality that causes the exploitation of Firefox. The results (Figure 4(c)) indicate a causality between the CameraGrab payload and the browser as the instruction pointer of Firefox goes to the payload. A further check of Firefox reveals that the main page has become a malicious site, so the browser is exploited every time it is started.

		Time (s)			#Taint		
Attack	**Analysis**	**PruT**	**RefiT**	**T(P+R)**	**Rain**	**None**	**Fraction%**
MotivExp	A(O-O)	759	2,321	3,080	34	720	**4.7%**
NetRecon	A(O-O)	140	1,320	1,460	13	138	**9.4%**
ScreenGrab	A(O-O)	127	253	380	5	99	**5.0%**
CameraGrab	A(O-P)	326	757	1,083	19	141	**13.4%**
AudioGrab	A(P-P)	301	687	988	11	310	**3.5%**

Table 5: Analysis cost comparison. The PruT column lists the time used to extract the subgraph prior to fine-grained refinement. RefiT shows the time it takes to selectively refine causality using taint analysis; T(P+R) represents the total analysis time and #Taint-Rain how many process groups are replayed and taint tracked with the point-to-point refinement algorithm (§7.2.3); #Taint-None provides the number of process groups to be replayed and tainted between the two time points without reachability and selective DIFT algorithms. The average fraction ratio is 5.8%.

8.1.3 Pruning and refinement. In general, the resulting subgraph is substantially smaller than the global graph (> 90%) as well as the SPS that is computed by the coarse-level analysis. More importantly, because of DIFT, the analysis reveals the *true* causalities. We analyzed several attack scenarios and summarize their incremental pruning and refinement results in Table 3 (the specifics of the analysis requests are listed in Table 4). In particular, we list the false positive rates using coarse-level data with Rain refinement and the reduction ratio. The potential causalities (denominator) are counted according to the "dependency explosion" [33] definition in which each output is assumed to depend on all the earlier inputs. With Rain, most false positives in the provenance graph are eliminated (i.e., a 100% reduction), but we also encountered two cases in which false positives remained after refinement. When we took a closer look at the DIFT, we observed the "over-tainting" situation that occurs during control flow-based propagation which is a known limitation of DIFT. In general, Rain effectively improves the precision of attack investigation.

8.2 Performance

8.2.1 Analysis Performance. To fairly examine the time duration and tainting workload induced by Rain, we evaluate the cost of analysis using bounded point-to-point queries. In Table 5, we first show the time duration for Rain to prune (column **PruT**) and refine (**RefiT**) the data using the point-to-point refinement in parallel (i.e., the longest duration among instances of tainting).

We then evaluate the performance of the reachability analysis and selective DIFT. We first list the number of all of the process groups between the two points in the **None** column. If one attempted to refine the causalities without applying the pruning (§6) or the selective DIFT algorithms (§7), this number would represent the load. It would also reflect the user-land part of the taint workload in full-system DIFT systems (e.g., [50, 54]). The number of tainting instances that Rain performs for the same task is listed in the **Rain** column. We find that our algorithm is effective, significantly reducing the tainting workload to a fraction of 5.8% on average. Note that we focus on the factual tainting workload the analysis must take, rather than the total time. After all, one can parallelize the workload on multiple machines to reduce time consumption.

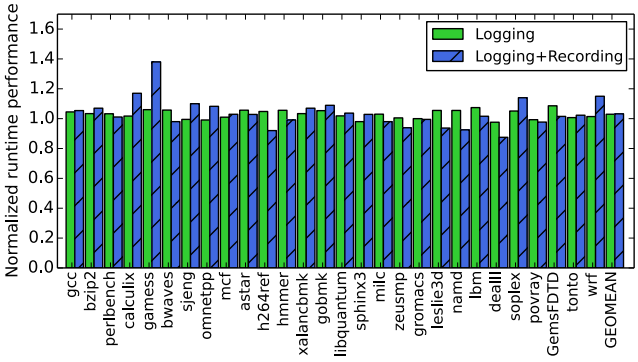


Figure 5: Normalized runtime performance with SPEC CPU 2006.

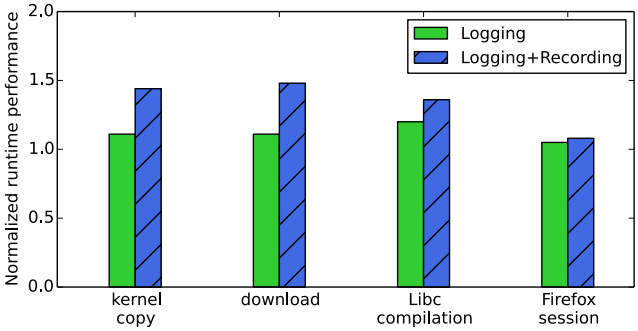


Figure 6: Normalized runtime performance with I/O intensive applications.

8.2.2 Runtime Performance. We evaluate the runtime performance of Rain with SPEC CPU2006 benchmarks listed in Figure 5. The runtime overhead of only running system logging is represented by the green bars for various testing items. The overhead of logging plus recording is listed in the blue bars. The geometric mean of runtime overhead in a logging+recording mode is 3.22%.

Besides the CPU intensive benchmark, we also run I/O intensive applications as Rain hooks system calls and caches file or a network I/O. We compare the runtime performance of four applications: copying the Linux kernel 3.5.0 archive with `cp`, downloading a 450MB video `mp4` file from a local area network with `wget`, compiling the `eglibc-2.15` library, and loading `cnn.com` in Firefox. Figure 6 illustrates the normalized overhead breakdown in terms of system logging and full mode (logging+recording). In these I/O intensive cases, Rain incurs no more than 50% overhead.

To evaluate the runtime performance of Rain in multi-core machines, we run the SPLASH-3 [46] multi-core benchmark with a 4-core CPU and summarize the results in Figure 7. The geometric mean of runtime overhead (logging+recording) is 5.35%, and Rain is able to faithfully replay all the benchmarks without divergence.

8.2.3 Storage Cost in Scenarios. We measure the storage cost of Rain with the scenarios used in §8.1 and a high workload case (i.e., compiling `eglibc-2.15`), the results of which are summarized in Table 6. The compiled `libc` is around 235MB, which is smaller than either the system or record log. This is because Rain not only cached the target files that were built but also the temporary files generated during the compiling. Even though the log size is larger than the

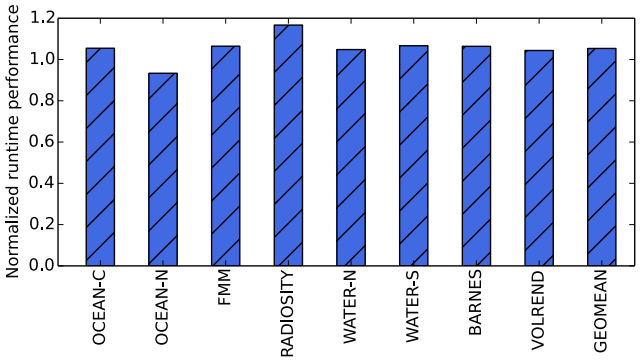


Figure 7: Normalized runtime performance (logging+recording) with SPLASH-3 multi-core benchmark (4-core CPU). OCEAN-C: contiguous; OCEAN-N: non-contiguous; WATER-N: nsquared; WATER-S: spatial.

	Storage usage (MB)		
Case	**Log**	**Record**	**Total**
MotivExp	45.6	155	201
NetRecon	29	137.1	166
ScreenGrab	16.6	97.3	114
CameraGrab	15.8	89.2	105
AudioGrab	18.2	115.4	133
Libc compiliation	327	413	740

Table 6: Storage cost (MB).

resulting size of compilation, it shows Rain's ability to re-construct (and analyze) a transient state of a complex program execution (we presented the storage footprint of daily usage in §4.3).

9 LIMITATIONS AND DISCUSSION

This section presents the limitations of Rain and directions of future work. One limitation of Rain is that it is a kernel-based system. It is able to record, replay, and analyze the activities of user-level processes but unable to monitor the kernel activities because it trusts the kernel. If the kernel is compromised, Rain is no longer able to create reliable provenance data. Thus if such an incident ever occurred, Rain could use kernel-integrity monitoring techniques [25, 32, 38, 43, 47] that detect and filter out misinformation. In the future, we will port Rain to a hypervisor that records and replays kernel activities while reducing attack surfaces. This approach will allow Rain to support commercial off-the-shelf (COTS) OSes (e.g., Windows). In addition, the semantic information of Rain poses a limitation because it is less comprehensive than that of source-code-instrumentation-based approaches [33, 34] since it assumes no assistance from software developers (i.e., annotation). We believe that these approaches are orthogonal: while Rain is successful at extracting fine-grained information from COTS programs or unknown binaries (e.g., malware), instrumentation-based approaches are effective at collecting semantically rich information from supportive programs. Another limitation is the over-tainting issue that we encountered, particularly when dealing with control flow-based propagation. This problem, which has always plagued DIFT approaches needs to be addressed. Since Rain relies on triggering analysis to initiate a fine-grained analysis, it could either miss or delay the detection of some stealthier attacks Further, faulty triggers could simply waste the time and energy resources of Rain. To solve this problem, we plan to develop an anomaly-based self-triggering mechanism that automatically initiates a fine-grained analysis. Lastly, the storage overhead of Rain is greater than that of other systems such as ProTracer [34]. Unlike such systems, Rain records all kinds of system calls (§4.3) to support replay-able execution, so the additional storage overhead appears to be unavoidable. We plan to explore a further reduction of storage overhead, for example, by compression and deduplication.

Prov Systems	**Data Granularity**	**Runtime Overhead**	**Requirement**
PASS [39]	Syscall (Workflow)	Low (<30%)	None (Source)
SPADE [24]	Syscall (Workflow)	Low (<10%)	None (Source)
LPM [12]	Syscall (Workflow)	Low (<10%)	None (Source)
DTrace [3]	Syscall (Workflow)	Low (<30%)	None (Source)
RecProv [28]	Syscall	Low (20%)	None
BEEP [33]	Unit	Low (<2%)	Binary
ProTracer [34]	Unit	Low (<7%)	Source
Panorama [54]	Instruction	High (20×)	None
DataTracker [49]	Instruction	High (4–6×)	None
PROV-Tracer [50]	Instruction	High (5×)	None
Rain	**Instruction**	**Low (3.22%)**	**None**

Table 7: Comparison of full-system provenance systems. We compare the existing systems and Rain in terms of provenance granularity, runtime overhead and requirement. "Workflow" in the brackets is another mode that monitors user-land applications, but requires source code instrumentation. Rain achieves both efficient runtime and instruction level analysis granularity while does not require source or binary instrumentation.

10 RELATED WORK

Full-system Provenance Logging. Full-system provenance logging is essential to detecting complicated attacks. For example, Linux supports the Linux Audit system [4], which records information about system events. PASS [39, 40] is a storage system that automatically logs and maintains provenance data. SPADE [24] is a cross-platform system that logs provenance data across distributed systems, and Linux provenance modules (LPM) [12] is a generic framework that allows developers to write Linux security module (LSM)-like modules that define custom provenance rules. In addition, ProTracer [34] is a lightweight provenance tracking system that supports system event logging and unit-level taint propagation, which is based on BEEP [33]. To reducing logging workload, ProTracer "taints" in order to keep track of the units, which fundamentally differs from the dynamic instrumentation-based taint tracking that we apply. However, none of the systems provides the instruction-level fine-grained provenance data that Rain provides because they cannot achieve one of their main goals—minimizing runtime overhead—should they provide instruction-level provenance data. RecProv [28] relies on a user-level record and replay technique to recover syscall-level provenance, but its replay does not perform instruction-level instrumentation, so it provides no finer-grained causality. DataTracker [49] performs taint tracking and provides fine-grained causality data on individual files, but because of the high execution overhead of taint analysis, using it as an analysis system instead of a production system is impractical.

PROV-Tracer [50] is built on top of PANDA [20], which can also provides instruction-level granularity. However, the QEMU emulator it bases on is around 5× slower than a native execution. We summarize the comparisons between Rain and previous provenance systems in Table 7.

Network Provenance Systems. Network provenance systems [11, 15, 16, 55] focus on tracking network-level provenance between computing hosts belonging to the same distributed or enterprise network environment. Their main goal is to find faulty (or compromised) hosts that attempt to attack other hosts in the same network by monitoring and analyzing network traffic, and to ascertain how the faulty hosts were compromised. Since they focus on network traffic, system-level fine-grained provenance (e.g., CPU instructions that were executed) are beyond the scope of the work that proposed these systems. Thus, Rain is orthogonal to these network provenance systems such that both can be simultaneously used to fully cover intra- and inter-system provenance.

Replay-Based Decoupled Analysis. Unlike the deterministic replay technique that faithfully replays the previous execution (e.g., in [21, 42, 45, 48]), the replay-based decoupled analysis technique enables sophisticated analysis during replay. Arnold [19] is a state-of-the-art record-and-replay system supporting decoupled analysis during replay. The two main advantages of Arnold over two similar systems, Aftersight [17] and PANDA [20], are 1) its minimal recording overhead and 2) its process-group-wise replay with Intel Pin. These advantages stem from the implementation of Arnold inside the Linux kernel. By contrast, Aftersight is based on both a VMWare hypervisor (record) and QEMU (replay), and PANDA is purely based on QEMU. The main disadvantage of Arnold is that, unlike the other two systems, it is built inside the kernel, so it cannot record and replay the execution of the kernel.

We choose Arnold [19] as the base system of Rain mainly because of its efficiency. Note that Rain's functionalities (e.g., full-system recording, provenance data generation, reachability analysis, and refinable DIFT) are orthogonal to Arnold, so we can apply them to other systems easily. In fact, we have PANDA-based Rain, which provides the same functionalities even though its recording overhead is excessive (five times as high) mainly resulting from QEMU.

Decoupled Taint Analysis. Dynamic taint analysis [18, 22, 29, 41, 51, 54] is a well-known technique for tracking the data flow from a source to a sink. Taint analysis is useful for runtime security policy enforcement [41, 51], malware analysis [54], and privacy leakage detection [22]. However, because of its excessive performance overhead (e.g., the overhead of one state-of-the-art implementation, libdft [29], is six times as high), it is difficult to use it in a general computing environment. To solve this performance problem, several studies have proposed decoupled taint analysis techniques [27, 36, 37, 44]. The purpose of these techniques is to run a target process with a CPU core while performing taint analysis for a process with other idle CPU cores.

11 CONCLUSION

We presented Rain, a practical attack investigation system with runtime efficiency and refinable granularity (from system call to instruction). By leveraging a record-and-replay technique, Rain achieves efficiency using graph-based analysis to prune out unrelated executions, and it performs DIFT only on relevant executions to identify fine-grained causality. We demonstrated Rain's effectiveness by applying it to an evaluation dataset to perform a precise causality analysis of sophisticated attacks.

12 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This research was supported in part by NSF, under awards CNS-0831300, CNS-1017265, DGE-1500084, CCF-1548856, CNS-1563848, SFS-1565523, CRI-1629851, and CNS-1704701, ONR, under grants N000140911042, N000141512162, and N000141612710, DARPA TC (No. DARPA FA8650-15-C-7556) and XD3 programs (No. DARPA HR0011-16-C-0059), NRF-2017R1A6A3A03002506, ETRI IITP/KEIT [B0101-17-0644], and gifts from Facebook, Mozilla, and Intel.

REFERENCES

[1] 2017. Apache Avro. (Oct. 2017). https://avro.apache.org.
[2] 2017. DARPA Transparent Computing Program. (Oct. 2017). http://www.darpa.mil/program/transparent-computing.
[3] 2017. DTrace. (Oct. 2017). https://www.dtrace.org.
[4] 2017. Linux Audit. (Oct. 2017). https://linux.die.net/man/8/auditd.
[5] 2017. Mozilla rr. (Oct. 2017). http://rr-project.org.
[6] 2017. Neo4j Graph Database. (Oct. 2017). http://neo4j.com.
[7] 2017. Snort. (Oct. 2017). https://www.snort.org.
[8] 2017. Squid. (Oct. 2017). http://www.squid-cache.org.
[9] 2017. Sysdig. (Oct. 2017). https://www.sysdig.org.
[10] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA.
[11] Adam Bates, Kevin Butler, Andreas Haeberlen, Micah Sherr, and Wenchao Zhou. 2014. Let SDN be your eyes: Secure forensics in data center networks. In *2014 NDSS Workshop on Security of Emerging Network Technologies (SENT)*.
[12] Adam Bates, Dave (Jing) Tian, Kevin R.B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC.
[13] Kurt Baumgartner. 2017. On the StrongPity Waterhole Attacks Targeting Italian and Belgian Encryption Users. (Oct. 2017). https://securelist.com/blog/research/76147.
[14] Ang Chen, W. Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. 2014. Detecting Covert Timing Channels with Time-Deterministic Replay. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado.
[15] Ang Chen, Yang Wu, Andreas Haeberlen, Boon Thau Loo, and Wenchao Zhou. 2017. Data Provenance at Internet Scale: Architecture, Experiences, and the Road Ahead. In *Conference on Innovative Data Systems Research (CIDR'17)*.
[16] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of the 2016 ACM SIGCOMM*. Florianopolis, Brazil.
[17] Jim Chow, Tal Garfinkel, and Peter M. Chen. 2008. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference (ATC)*. Boston, MA.
[18] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. London, UK.
[19] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. 2014. Eidetic systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado.
[20] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW)*.
[21] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA.
[22] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings*

of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Vancouver, Canada.

[23] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX.

[24] Ashish Gehani and Dawood Tariq. 2012. SPADE: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference (Middleware)*.

[25] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. 2011. Ensuring Operating System Kernel Integrity with OSck. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA.

[26] Yeongjin Jang, Simon P Chung, Bryan D Payne, and Wenke Lee. 2014. Gyrus: A Framework for User-Intent Monitoring of Text-based Networked Applications. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[27] Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. 2013. ShadowReplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*. Berlin, Germany.

[28] Yang Ji, Sangho Lee, and Wenke Lee. 2016. RecProv: Towards Provenance-Aware User Space Record and Replay. In *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)*. Mclean, VA.

[29] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. London, UK.

[30] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. 2012. Recovering from intrusions in distributed systems with DARE. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*. Seoul, South Korea.

[31] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada.

[32] Hojoon Lee, HyunGon Moon, DaeHee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. 2013. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *Proceedings of the 22th USENIX Security Symposium (Security)*. Washington, DC.

[33] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[34] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[35] Emaad A Manzoor, Sadegh Momeni, and Leman Akoglu. 2016. Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs. In *Proceedings of the 22nd ACM SIGKDD Knowledge Discovery and Data Mining (KDD)*. San Francisco, CA.

[36] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. 2016. StraightTaint: decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Singapore.

[37] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC.

[38] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. 2012. Vigilare: Toward Snoop-based Kernel integrity Monitor. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. Raleigh, NC.

[39] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A Holland, Peter Macko, Diana L MacLean, Daniel W Margo, Margo I Seltzer, and Robin Smogor. 2009. Layering in Provenance Systems. In *Proceedings of the 2009 USENIX Annual Technical Conference (ATC)*. San Diego, CA.

[40] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-Aware Storage Systems. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[41] James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[42] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[43] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. 2004. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium (Security)*. San Diego, CA.

[44] Andrew Quinn, Dave Devecsery, Peter M. Chen, and Jason Flinn. 2016. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA.

[45] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. 2016. Samsara: Efficient Deterministic Replay in Multiprocessor Environments with Hardware Virtualization Extensions. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. Denver, CO.

[46] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS'16)*.

[47] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integirty for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*. Stevenson, WA.

[48] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, and Yuanyuan Zhou. 2004. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[49] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2014. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)*. Cologne, Germany.

[50] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2015. Decoupling Provenace Capture and Analysis from Execution. In *Proceedings of the 7th USENIX Workshop on the Theory and Practice on Provenance (TaPP)*. Edinburgh, Scotland.

[51] G. Edward Suh, Jae W. Lee, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Boston, MA.

[52] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[53] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. 2016. ReplayConfusion: Detecting Cache-based Covert Channel Attacks Using Record and Replay. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Taipei, Taiwan.

[54] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. Alexandria, VA.

[55] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure network provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal.