Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking

Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee, ***Georgia Institute of Technology***

https://www.usenix.org/conference/usenixsecurity18/presentation/jia-yang

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking

Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing,
Taesoo Kim, Alessandro Orso and Wenke Lee

Georgia Institute of Technology

Abstract

Investigating attacks across multiple hosts is challenging. The true dependencies between security-sensitive files, network endpoints, or memory objects from different hosts can be easily concealed by dependency explosion or undefined program behavior (e.g., memory corruption). Dynamic information flow tracking (DIFT) is a potential solution to this problem, but, existing DIFT techniques only track information flow within a single host and lack an efficient mechanism to maintain and synchronize the data flow tags globally across multiple hosts.

In this paper, we propose RTAG, an efficient data flow tagging and tracking mechanism that enables practical cross-host attack investigations. RTAG is based on three novel techniques. First, by using a record-and-replay technique, it decouples the dependencies between different data flow tags from the analysis, enabling lazy synchronization between independent and parallel DIFT instances of different hosts. Second, it takes advantage of system-call-level provenance information to calculate and allocate the optimal tag map in terms of memory consumption. Third, it embeds tag information into network packets to track cross-host data flows with less than 0.05% network bandwidth overhead. Evaluation results show that RTAG is able to recover the true data flows of realistic cross-host attack scenarios. Performance wise, RTAG reduces the memory consumption of DIFT-based analysis by up to 90% and decreases the overall analysis time by 60%–90% compared with previous investigation systems.

1 Introduction

Advanced attacks tend to involve multiple hosts to conceal real attackers and attack methods by using command-and-control (C&C) channels or proxy servers. For example, in the Operation Aurora [22] attack, a compromised victim's machine connected to a C&C server that resided in the stolen customers' account, and exfiltrated proprietary source code from the source code repositories. Gibler and Beddome demonstrated GitPwnd [32], an attack that takes advantage of the `git` [11] synchronization mechanism to exfiltrate victim's private data through a public `git` server. Unlike common data exfiltration attacks that only involve a victim host, GitPwnd leverages two hosts (victim's host and public `git` server) to complete the exfiltration.

Unfortunately, existing attack investigation systems, also known as provenance systems, are inadequate to figure out the true origin and impact of cross-host attacks. Many provenance analysis systems (such as [19, 35, 45]) are designed to monitor the system-call-level or instruction-level events within each host while ignoring cross-host interactions. In contrast, network provenance systems [64, 68, 69] focus on the interaction between multiple hosts, but, because they lack detailed system-level information, their analysis could result in a *dependency explosion problem* [35, 42]. To fully understand the steps and end-to-end information flow of a cross-host attack, it is necessary to collect accurate flow information from individual hosts and correctly associate them to figure out the real dependency.

Extending existing provenance systems to investigate cross-host attacks is challenging because problems of accuracy, performance, or both can be worse with multiple hosts. Although collecting coarse-grained provenance information (e.g., system-call-level information) introduces negligible performance overhead, it cannot accurately track dependency explosion and undefined program behaviors (e.g., memory corruption) even within a single host. That is, if we associate the coarse-grained provenance information from different hosts using another vague link (e.g., network session [64, 68, 69]), the result will contain too many false dependencies. Fine-grained provenance information, (e.g., instruction-level information from dynamic information flow tracking (DIFT)), is free from such accuracy problems. However, it demands

many additional computations and consumes huge memory, which will increase according to the number of hosts. More seriously, existing cross-host DIFT mechanisms piggyback metadata (i.e., *tags*) on network packets and associate them during runtime [50, 67], which is another source of huge performance degradation.

To perform efficient and accurate information flow analysis in the investigation of cross-host attacks, we propose a record-and-replay-based data flow tagging and tracking system, called RTAG. Performing cross-host information flow analysis using a record-and-replay approach introduces new challenges that cannot be easily addressed using existing solutions [25, 35, 50, 67]: that is, long analysis time and huge memory consumption. First, the communication between different hosts (e.g., through socket communication) introduces information flows that require additional information and procedure for proper analysis. Namely, the DIFT analysis requires transfer of the analysis data (i.e., tags) between the hosts in a synchronized manner. Existing record-and-replay solutions have to *serialize* the communication between hosts to transfer tags because no synchronization mechanism is implemented, leading to longer than necessary analysis time. Second, because a number of processes can run on multiple hosts under analysis, the memory requirement for DIFT instances could become tremendous, especially when multiple processes on different hosts interact with each other.

To overcome these two challenges, RTAG decouples the *tag dependency* (i.e., information flow between hosts) from the analysis with *tag overlay* and *tag switch* techniques (§6), and enables DIFT to be *independent* of any order imposed by the communication. This new approach enables the DIFT analysis to happen for multiple processes on multiple hosts in *parallel* leading to a more efficient analysis. Also, RTAG reduces the memory consumption of the DIFT analysis by carefully designing the *tag map* data structure that tracks the association between tags and associated values. Evaluation results show significant improvement both in analysis time, decreased by 60%–90%, and memory costs, reduced by up to 90%, with realistic cross-host attack scenarios including GitPwnd and SQL injection.

This paper makes the following contributions:

- **A tagging system that supports refinable cross-host investigation.** RTAG solves "tag dependency coupling," a key challenge in using refinable investigation systems for cross-host attack scenarios. RTAG decouples the tag dependency from the analysis which spares the error-prone orchestrating effort on replayed DIFTs and enables DIFT to be performed independently and in parallel.
- **DIFT runtime optimization.** RTAG improves the runtime performance of doing DIFT tasks at replay time in terms of both time and memory. By performing DIFT tasks in parallel, RTAG reduces the analysis time by over 60% in our experiments. By allocating an optimal tag size for DIFT based on system-call-level reachability analysis, RTAG also reduces the memory consumption of DIFT by up to 90% compared with previous DIFT engines.

The rest of paper is organized as follows: §2 describes the background of the techniques that supported RTAG's realization. §3, §4, and §5 present the challenges, an overview and the threat model of RTAG; §6 presents the design of RTAG; More specifically, §6.1 describes the data structure of RTAG, §6.3 explains how RTAG facilitates the independent DIFT; §6.4 describes how RTAG conducts tag switch for DIFT, and §6.6 presents the tag association module and how RTAG tracks the traffic of IPC. §7 gives implementation details and the complexity. §8 presents the results of evaluation. §9 summarizes related work, and §10 concludes this paper.

2 Background

RTAG utilizes concepts from a variety of research areas. This section provides an overview of these concepts needed to understand our system.

2.1 Execution Logging

Attack investigation systems most often rely on logged information to perform their analyses. Different systems use different levels of granularity when logging information for their analyses (e.g., system-call level versus instruction level) as the cost of collecting this information changes based on the selected granularity level. A first category of systems [6, 8, 19, 45] collects information at a high-level of granularity (e.g., system-call level) and generally have low runtime overhead. However, the information collected at this level of granularity might affect the accuracy of their analyses as it does not always provide all of the execution details. A second category of systems improves accuracy by analyzing program executions at the instruction level [24, 44, 66]. These systems provide very accurate results in their analyses. However, they introduce a runtime overhead that is not suitable for production software. Finally, a third category of systems [25, 35] combines the benefits of systems from the previous two categories using record and replay. These systems perform high-level logging/analysis while recording the execution of programs and perform low-level logging/analysis in a replayed execution of the programs. More specifically, RAIN [35] logs system call information about user-level processes using a kernel instrumentation approach. The system then analyzes instructions in a replayed execution of the processes.

2.2 Record and Replay

Record and replay is a technique that aims to store information about the execution of a software system (record phase) and use the stored information to re-execute the software in such a way that it follows the same execution path and also reconstructs the program states as the original execution (replay phase). Record and replay techniques can be grouped under different categories based on the layer of the system in which they perform the record-and-replay task. Some techniques perform record and replay by instrumenting the execution of programs at the user level [9, 33, 51, 58, 59]. These techniques are efficient in their replay phase as they can directly focus on the recorded information for the specific program. However, these techniques either require program source or binary code for instrumentation or have additional space requirements when recording executions of communicating programs (especially through the file system) as the recorded information is stored multiple times. The second category of techniques performs record and replay by observing the behavior of the operating system. Techniques do so by either monitoring the operating system through a hypervisor [20, 23, 56] or emulation [27]. These techniques are efficient in storing the information about different executing programs. However, they usually need to replay every program recorded even when only one program is of interest for attack investigation. Finally, a third category of techniques uses an hybrid approach. This category records information at the operating system level and replays the execution leveraging user-level instrumentation [25, 35] (e.g., by hooking `libc` library) for multi-thread applications. More specifically, Arnold [25] and RAIN [35] reside inside the kernel of operating system and record the non-deterministic inputs of executing programs. The replay task is achieved by combining kernel instrumentation with user-level instrumentation so that replay of a single program is possible.

2.3 Dynamic Information Flow Tracking

Dynamic information flow tracking (DIFT) is a technique that analyzes the information flowing within the execution of a program. This technique does so by: (1) marking with tags the "interesting" values of a program, (2) propagating tags by processing instructions, and (3) checking tags associated with values at specific points of the execution. There are several instantiations of this technique [24, 34, 37, 47, 55, 66]. These instantiations can precisely determine whether two values of the program are related to each other or not. However, because the technique needs to perform additional operations for every executed instruction, that action generally introduce an overhead which makes it unsuitable in production.

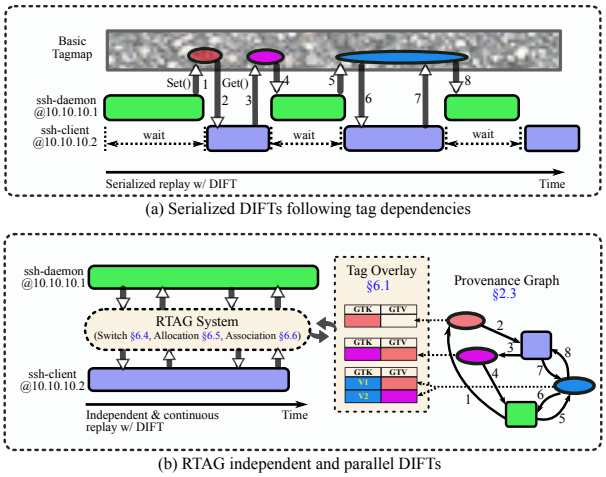

Figure 1: Comparison of the serialized DIFTs and RTAG parallel DIFTs. We highlight the components of RTAG with dashed circles. (a) shows the serialized DIFT for the ssh daemon on the server and the ssh client on another host, both of which follow the tag dependencies same as those were recorded. (b) depicts that RTAG decouples the tag dependency from the replays of processes by using the tag switch, allocation and association techniques so that each process in the offline analysis can be performed independently.

Arnold [25] and RAIN [35] make dynamic information flow tracking feasible by moving the cost of the analysis away from the runtime using a record-and-replay approach that performs DIFT only in the replayed execution. RAIN [35] also improves the efficiency of the analysis when considering an execution that involves multiple programs. RAIN [35] does so by: (1) maintaining a provenance graph that captures the high-level relations between programs; (2) performing reachability analysis on the provenance to discard executions that do not relate to the security task under consideration and instead pinpointing the part of the execution where the data-dependency confusion exists (i.e., memory overlaps, called *interference*); (3) performing DIFT only for interferences by replaying the execution and fast-forwarding to that part.

3 Motivating Example and Challenges

In section, we describe the challenges of performing refinable attack investigation across multiple hosts. We first present a motivating attack example (GitPwnd [32]) involving multiple hosts in a data exfiltration; then, we present what challenges we face with currently available methods.

3.1 The GitPwnd Attack

GitPwnd uses a popular versioning control tool `git` to perform malicious actions on a victim's host and sync the

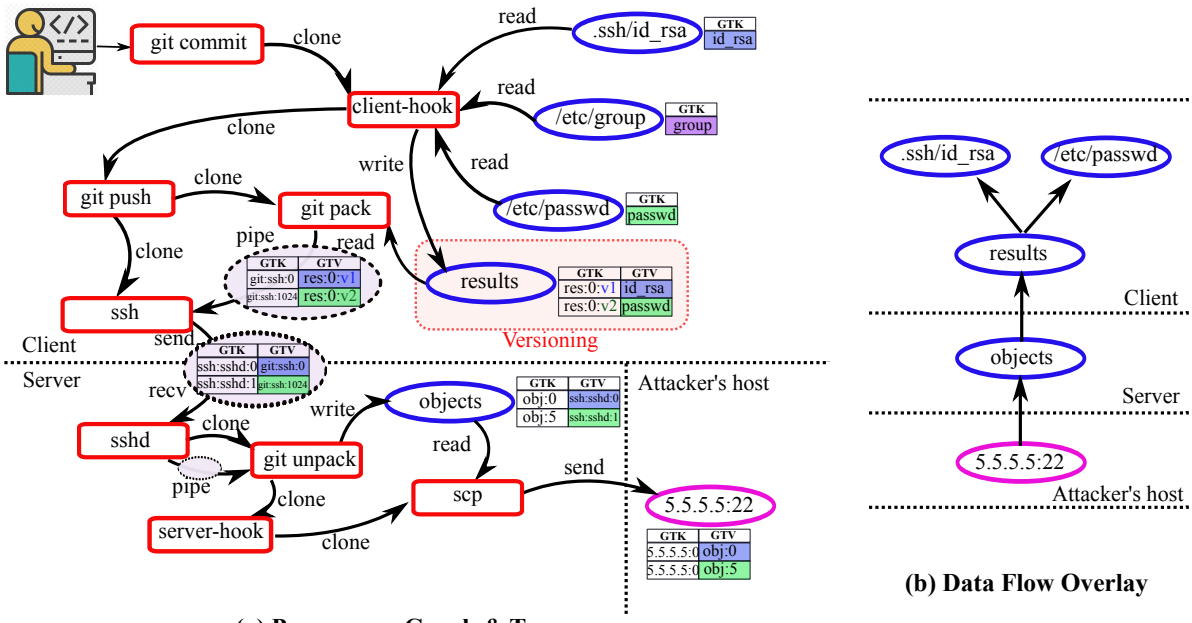

Figure 2: Visualized Pruned Provenance Graph and Tags. (a) is the simplified provenance graph of the GitPwnd attack involving three hosts, of which the `git` client and `git` server are monitored by RTAG. We use red rectangles to represent processes, blue ovals for file objects, and pink ovals for out-of-scope remote host; we use directed edges to represent the data flows and parent-child relations between processes. The tags with dashed circles are the IPC tags for `pipe` and `socket` communication. (b) is the result of a backward query from the attacker's host, the data flow overlay; it appears to be a tree, giving the data flow every step from the exfiltrated private key and `/etc/passwd` (excluding `/etc/group`) to the attacker's host, crossing three hosts.

result to an attacker's controlled host via a `git` server. Unlike conventional data exfiltration attacks, this attack involves multiple hosts (i.e., a victim's host and the `git` server) to achieve the exfiltration. This attack evades an existing network-level intrusion detection system, as the victim's host does not have a direct interaction with any untrusted host (i.e., the attacker's host). In addition, this attack appears to be innocuous inside the developers' network, as `git` operations are usually assumed to be benign. We implement this attack using `gitolite` [12] at the server side and `git` at the client side.

The starting point of the attack is a malicious mirror of a popular `git` repository, which includes a hooking script that clones a command-and-control (C&C) repository for future communication. Whenever a developer (a victim host) happens to clone the malicious mirror, the `git` client will automatically clone the C&C repository as well due to the hooking script. The C&C repository includes agent and payload, whose executions will be triggered by a certain `git` operation (e.g., `git commit`) by the developer. Their execution results are saved and synced to the C&C repository. Note that the C&C repository shares the privilege of the malicious mirror repository, so it also is white-listed by the developer's host. Whenever the C&C repository receives the exploit results (stored into `objects`), it shares the results with the attacker's host (via `scp`). More specifically, this `git push` involves three processes. 1) The `git` first forks an `ssh` process, handling the ssh session with the remote host, and then 2) spawns another `git pack` process packing the related objects of the push. 3) The `pack` process uses `pipe` to transfer the packed data to the `ssh` process. The communication between the C&C repository and the attacker's host is invisible to the victim. We visualize an abbreviated pruned provenance subgraph of the attack in Figure 2(a). We will continue to use this attack as a running example throughout the rest of the paper.

3.2 Challenges

Satisfying both the accuracy and the efficiency for cross-host data flow tracking are challenging. Existing provenance systems that support cross-host accurate data flow capturing [50, 67] rely on performing DIFT at the *runtime*, which *naturally* propagates the tags from the execution of a program to another host without losing any tags and their dependencies. Unfortunately, such systems suffer from 10×–30× runtime overhead, making them impractical in production systems. Instead, to ensure both runtime efficiency and accurate data flow tracking, *refinable* systems [25, 35] record the execution of every process in the system, and *selectively* replay some of them related to

the attack with DIFT instrumentation. However, existing refinable systems are subject to a tag-dependency challenge that requires the replay and DIFT of every process to be performed in the same order as the recording if a dependency exists in tags involved in different replayed processes. The enforcement of the order requires the DIFT tasks to wait for their upstream DIFTs to update the tag values that they depend on. Although the record-and-replay function can faithfully re-construct the program states at replay time, it still takes non-trivial (and error-prone) efforts to serialize and orchestrate the replays of different processes to re-establish the dependencies for tag propagation between different hosts.

The tag-dependency challenge becomes outstanding when we aim to replay processes on multiple hosts to investigate cross-host attacks. This is because the interactive two-way communication (for the purpose of network or application-level protocol) demands the replays to be paused and waiting iteratively for enforcing the same tag dependency as the recording, which further lengthens the waiting time (i.e., analysis time consumption), and increases the complexity of replay orchestration.

Let us look into one example of replay from the Gitpwnd attack [32] (detailed in §3.1) for the communication between the client-side `ssh` and the server-side `sshd` in Figure 1(a). At the server side, the replay of `sshd` needs to be *paused* to wait for the replay of `ssh-client` at the client side to fulfill the propagation results in the tag map for the traffic. Furthermore, this traffic will be used by `sshd` to respond to `ssh` as an `ssh` protocol response, which means the replay of `ssh` needs to be paused and wait for `sshd` as well.

This challenge is exacerbated when many parties are involved in group communication. For example, to enforce the tag dependencies for the operation of searching and downloading a file from a peer-to-peer (P2P) file sharing network (e.g., Gnutella [7]), we need to orchestrate the replays of P2P clients on each node, in which case the approach becomes infeasible particularly when we are faced with hundreds or thousands of nodes. §8 shows the DIFT time cost and compares it with RTAG in Table 1.

To systematically overcome the tag-dependency challenge, we propose RTAG that *decouples* the tag dependencies from the replays by using symbolized tags with optimal size for each *independent* DIFT. We show RTAG effectively solves the challenge while significantly speeding up DIFT tasks and reducing their memory consumption.

4 Overview

We propose a tagging system, RTAG, that decouples the tag dependency from the analysis (i.e., DIFT tasks), which previously was *inlined* along with the program execution or its replayed DIFT, and enables DIFT to be *independent* of any required order—allowing performing DIFT for different processes on multiple hosts in *parallel*. Such independence spares the complex enforcement of orders during the offline analysis. Note that our parallel DIFT concerns *inter*-process (or host) DIFT, which is orthogonal to the *intra*-process parallel DIFT techniques in [46, 47, 55].

RTAG maintains a *tagging overlay* on top of a conventional provenance graph, enabling independent and accurate tag management. First, when DIFT is to be performed, RTAG uses a *tag switch* technique to interchange a *global* tag that is unique across hosts and a *local* tag that is unique for a DIFT instance. Using a local tag for each DIFT disentangles the coupling of tags shared by different DIFT tasks. After the DIFT is complete, RTAG switches the local symbol back to its original global tag. Second, to ensure no tag as well as their propagation to other tags is lost when the tag of a piece of data is updated more than once, RTAG keeps track of each change (*version*) of the data according to system-wide write operations. Each data version has its own tag(s) and each version of tag values can be correctly propagated to other pieces of data. Figure 1(b) depicts how RTAG facilitates the independent replay and DIFT for the cross-host `ssh` daemon and client example with the tag overlay and a set of techniques (i.e., tag switch, allocation, and association).

RTAG not only speeds up the analysis by enabling independent DIFT, but also reduces the memory consumption when DIFT is performed. We allocate local symbols of each DIFT with the *optimal* symbol size that is sufficient to represent the entropy of data involved in the memory overlap (i.e., "interference") in each DIFT (§6.5). For tracking the data communication across hosts, RTAG applies a *tag association* method (§6.6) to map the data that are sent from one host and the ones that are received at another host at byte level, which facilitates the identification of tag propagation across hosts.

5 Threat Model and Assumptions

In this section, we discuss our threat model and assumptions. The goal of our work is to provide a system for refinable cross-host attack investigation through DIFT. This work is under a threat model in which an adversary has a chance to gain remote access to a network of hosts, and will attempt to exfiltrate sensitive data from the hosts or to propagate misinformation (i.e., manipulate data) across the hosts. Our trusted computing base (TCB) consists of the kernel in which RTAG is running, and the storage and network infrastructure used by RTAG to analyze the information collected from the hosts under

analysis. Our TCB surface is similar to the one assumed by other studies [19, 35, 45, 48].

We make the following assumptions. First, attacks will happen only after RTAG is initiated (for collecting the information about attacks from the beginning to the end). Note that partial information about attacks can still be collected even if this assumption is not in place. Second, attacks relying on hardware trojans and side/covert channels are outside the scope of this paper. Although RTAG does not yet consider these attacks, we believe a record-and-replay approach has the potential to detect similar attacks as presented in related work [21, 65]. Third, we assume that although an attacker could compromise the OS or RTAG itself, the analysis for previous executions is still reliable. That is, we assume the attacker cannot tamper with the data collected and stored from program executions of the past. This can be realized by leveraging secure logging mechanisms [18, 68] or by managing the provenance data in a remote analysis server. Finally, we assume that the attacker cannot propagate misinformation by changing the payload of network packets while they are being transferred between two hosts (i.e., there is no man-in-the-middle attack).

6 Tagging System

We present the design of RTAG tagging system in this section. First, we describe the design of the *tag overlay* and how it represents and tracks the data provenance in the cross-host scope §6.1. Second, in §6.2, we recall the reachability analysis from RAIN [35] and how it is extended for the cross-host case and benefits the tag allocation. Third, we explain how RTAG decouples the tag dependencies from the replays (§6.3), and the tag switch technique (§6.4). Fourth, we explain how we optimize the local tag size in pursuit of memory cost reduction in the DIFT. Fifth, we describe how to associate tags in the cross-host communication §6.6. Finally, we present the investigation query interface in §6.7.

6.1 Representing Data Flow and Causality

To track the data flow between files and network flow across different hosts, we build the model of tags as an overlay graph on top of an existing provenance graph (such as RAIN [35]). Within the overlay graph, RTAG associates globally unique tags with interesting files to track their origin and flows at *byte-level* granuality. The tags allow RTAG to trace back to the origin of a file including from a remote host and to track the impacts of a file in the forward direction even to a remote host. With this capability, RTAG extends the coverage of the refinable attack investigation [35] to multiple hosts. The provenance graph is still necessary to track the data flows: 1) from a process to a file; 2) from a process to another process; and 3) from a file to a process. An edge indicates an event between two nodes (e.g., a system call such as one that a process node reads from a file node).

In the overlay tag graph, each byte of a file corresponds to a tag *key*, which uniquely identifies this byte. Each tag key is associated with a vector of *origin* value for this key (i.e., this byte). By recursively retrieving the value of a key, one obtains all of the upstream origins starting from this byte of data in a tree shape extending to the ones at a remote host. Reversely, by recursively retrieving the tag key of a value, the analyst is able to find all the impacts in a tree shape including the ones at a remote host (see Figure 2(b) as an example).

As we log the system-wide executions, RTAG needs to uniquely identify each byte of data in the file system on each host as a "*global tag*." For this requirement, RTAG uses a physical hardware address (i.e., `mac` address) to identify a host, identifiers such as `inode, dev, crtime` to identify a file, and an offset value to indicate the byte-level offset in the file. For example, the physical hardware address (i.e., `mac` address) is 48 bits long. The `inode, dev, crtime` are 64 bits, 32 bits, and 32 bits consecutively. The offset is 32-bits long, which supports a file as large as 4GB. Thus, in total, the size of a global tag can be 208 bits.

6.2 Cross-host Reachability Analysis

RTAG follows the design of reachability analysis in RAIN [35], and extends it to cope with the cross-host scenarios. Given a starting point(s), RTAG prunes the original system-wide provenance graph to extract a subgraph related to the designated attack investigation that contains the causal relations between processes and file/network flow. RTAG relies on the coarse-level data flows in this subgraph to maintain the tag overlay while performing tag switch and optimal allocation. The reachability analysis first follows the time-based data flow to understand the potential processes involved in the attack. Next, it captures the memory overlap of file or network inputs/outputs inside each process and labels them as "interference," to be resolved by DIFT. With accurate interference information, the replay and DIFT are fast forwarded to the beginning of the interference (e.g., a `read` syscall) and early terminated at the end (e.g., a `write` syscall).

For the network communication crossing different hosts, RTAG links the data flow from one host to another by identifying and monitoring the socket session. As we present in §6.6, RTAG tracks the session by matching the IP and port pairing between two hosts. RTAG further tracks the data transfer at byte level via socket communication for both TCP and UDP protocols, which enables the extension of tag propagation across hosts.

Unlike the runtime DIFT system, RTAG has the comprehensive knowledge of source and sink from the recorded file/network IO system-call trace, thus is able to allocate an optimal size of tag for each individual DIFT task. We show in §6.5 that this optimization significantly reduces the memory consumption of DIFT tasks. In addition, to avoid losing any intermediate tag updates to the same resource performed by different processes, RTAG particularly monitors the "overwrite" operations to the same offset of a file and tracks this versioning info, so it accurately knows which version of the tag should be used in the propagation.

6.3 Decoupling Tag Dependency

As a refinable provenance system, RTAG aims to perform DIFT at the offline replay time without adding high overhead to the runtime of the program. The replay reconstructs the same program status as the recording time by enforcing the recorded non-determinism to the replay of process execution. The non-determinism includes the file, network, and IPC inputs which are saved and maintained with a B-tree [25]. Such enforcement enables the program to be faithfully replay-able at process level.

To extend this approach to capture the end-to-end data flow across multiple hosts, we need to figure out how to coordinate replay programs on different hosts to track tag dependencies between them. One possible method is decoupling tag dependencies from each replay of the process, so it can be performed independently with no dependency on other replays. We achieve the decoupling by using *local* (i.e., *symbolized*) tags for each DIFT. Such symbolization needs to distinguish the change of a tag before and after the `write` operation on it, and synchronize the change to other related tags as well. In other words, RTAG needs to track the dynamic change of origin(s) of each tag after each IO operation (i.e., multiple *versions* of the tag are tracked).

Let us illustrate with the data exfiltration in the Gitpwnd attack example in Figure 2(a). The `client-hook` daemon keeps reading data from different files (e.g., `/etc/passwd`, `id_rsa`) and saves them into a `results` file which is recycled over a period of time. Meanwhile the `git pack` application copies from the `results` file whenever the victim does `git commit` operation, and shares data with `ssh` via the `pipe` IPC, which will be shipped off the host. To correctly differentiate the two data flows, `id_rsa`→`results`→`pipe` and `/etc/passwd`→`results`→`pipe`, RTAG needs to maintain two versions of the tags for `results`. The DIFT on `client-hook` stores the origin of `results.v1` to be `id_rsa`, and the origin of `results.v2` to be `/etc/passwd` (circled with red dash line), while the DIFT on `git pack` is able to discriminate the source of the IPC traffic `git:ssh` at offset 0 from `results.v1` and further from `id_rsa`, and the source of the IPC traffic at offset 1024 from `results.v2` and further from `/etc/passwd`. Most importantly, now the `client-hook` and `git pack` DIFT tasks can be performed independently without losing intermediate tag values because of the overwriting on `results`.

To facilitate the versioning, we append a 32-bit "version" field to indicate the version of the data in the file with regards to the file IO operation. According to the sequential system-call trace, the version is incremented at every event in which there is a write operation against this certain byte (e.g., `write()`, `writev()`). In the case of memory mapped file operation (e.g., `mmap()`), the version is incremented at the `mmap()` if the `prot` argument is set to be `PROT_WRITE`. The version field is only used when this tag is included in the data interference determined by the reachability analysis. We assign 32 bits for this field that can pinpoint a file IO syscall in around 500 days based on our desktop experiment.

6.4 Switching Global and Local Tags

The entropy of the global tag defined in §6.1 is sufficient enough to identify a byte of a file at a certain version across multiple hosts. However, using the global tag for each DIFT task is a waste of memory because each DIFT task of RTAG only covers a process group such that a *local* tag ensuring process-group-level uniqueness is enough. Thus, for each DIFT task, we use a different tag size based on the entropy of its source symbols. RTAG switches the tags from *global* to *local* before doing DIFT, and switches them back when the DIFT is done. The tag for DIFT is local because it only needs to uniquely identify every byte of the source in the current in-process DIFT, rather than identify a single byte of data across multiple hosts.

Further, the number of sources in each DIFT depends on the reachability analysis result, which is usually largely reduced by data pruning. In other words, the local tag size depends on the *interference* situation. Therefore, the entropy for the local tag is much lower than the global tag. For example, if the program reads only 10 bytes from a file marked as a source in DIFT, in fact as low as four bits are sufficient to represent each of these bytes. Compared against the global tag size (i.e., 208 bits §6.1), the switch brings 52× reduction in tag size (in practice, the reduction can be as large as 26× capped by the compiler-enforced byte-level granuality, which we discuss in detail in §7). Moreover, the tag size affects not only the symbols for the source and sink, but also all the intermediate memory locations and registers because the tags are copied, unioned, or updated along with the execution of each instruction according to the propagation policy of DIFT. Therefore, the

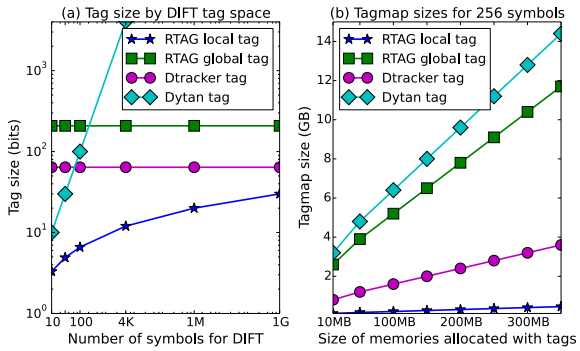

Figure 3: Memory cost for tags in DIFT. The left (a) shows the size of each tag given different numbers of symbols used in DIFT. The right (b) depicts the tagmap sizes based on different sizes of memories being allocated with tags when 256 symbols are used in the DIFT. RTAG local, global, DataTracker, and Dytan tags are compared.

tag size literally affects the memory cost of the whole tag map and tag switching significantly reduces the overall memory cost of DIFT.

6.5 Optimal Local Tag Allocation

The runtime cost of DIFT is high, both in time and storage. DIFT usually takes 10×–30× longer than the original execution because its instrumentation adds additional tag update operations to each executed instruction. Recent studies [34, 47] alleviate this issue by decoupling the instrumentation efforts from the runtime of the program. However, the storage footprint of *tag map*, the data structure used by DIFT to maintain the tag propagation status, can still be very high particularly when there are *multiple* (or *many*) sources.

The cost of tag map in DIFT depends on its supported type of tags and purpose. DIFT engines such as Taintcheck [49], Taintgrind [16], and ShadowReplica [34] use a basic binary tag model for DIFT, which assigns a boolean "tainted" or "not tainted" for each source of DIFT. It is able to tell whether the tainted data is propagated to the sink, which can be used to alarm sensitive data leakage or control-flow hijacking. However, this model is not flexible enough for the goal of RTAG, where the data dependency confusion it aims to resolve involves *multiple* sources.

Dytan [24] and DataTracker [61] provide a customizable model for the data sources and sinks. It allows the allocation of multiple tags to each addressable byte of data at the source or sink. The tag model used by such systems is flexible, but the tag map used to maintain the status of the taint propagation is "over-flexible" thus huge, which inhibits the deployment of such a system in many resource restrained cases. As these systems assume to be running at the runtime of a program, where no knowledge of the data at the source or sink is known prior, they usually assign a fixed size for each tag such that they are confident it is safely big enough. For example, DataTracker [61] uses 32 bits to identify an inbound file, and another 32 bits to identify the offset of the data (totally 64 bits). The size is sufficient for identifying every byte in a normal desktop. Dytan [24] represents whether one source is tainted or not as one bit and stores all the bits in a bit vector as the tag. Thus the size of each tag is *linear* to the number of sources, which can be huge in the case of a high number of sources. Note that the tag map not only stores the tags for the source and sink, but all the intermediate memory locations and registers as well. Since most implementations of DIFT maintain the tag map in memory to pursue faster instrumentation, such high use of memory has a possibility to cause the DIFT to crash before it is complete. This problem is elevated when the scope of investigation extends to multiple hosts since the workload of DIFT increases in proportion.

In contrast to the previous works that perform DIFT at the program runtime, RTAG is a record-replay based system in which the knowledge of data source and sink is *known* to us when we perform DIFT at replay time. In other words, we know which (bytes of) data need to be involved in the DIFT. Thus, we can adjust the tag size based on the entropy of the data dependency confusion, rather than use a fixed-size tag. Figure 3 compares the memory cost for tag map in different DIFT engines: (a) shows that the local tag of RTAG grows in logarithm while others are either linear or constant; (b) presents the total tag map size under different sizes of memories that are tainted (i.e., allocated with tags) where the memory cost introduced by RTAG is the lowest (by significant difference). Before DIFT, RTAG computes the optimal local tag needed to mark the source and substitute the global tag for the local one when a source is loaded to the memory space of the process (e.g., via `read()` syscall). While performing DIFT, RTAG allocates the tags for intermediate locations *lazily* when a memory location or register becomes tainted with some tag. When the propagation arrives at a sink (e.g., via a `write()` syscall), RTAG replaces the local tag with the original global one, and updates the tag value of the sink. We observe significant memory cost reduction by applying this optimal tag allocation method (see §8.2.1).

6.6 Tag Association

In order to track the data flow between different hosts, we additionally hook the socket handling of the operating kernel to enable the cross-host tagging. Prior studies adopt an "out-of-band" method to track the data flow communication (e.g., [38, 50]). Though this method is

more straightforward when identifying and managing the tags across hosts, it requires additional bookkeeping that incurs both complexity and overhead to the hosts. In contrast, we propose an "in-band" method to track the data flow among hosts, which particularly fits the system-level reachability analysis as well as the DIFT.

We design the cross-host tagging method based on the characteristics of the socket protocols. Our current tagging scheme supports the two major types of protocols (i.e., TCP [54] and UDP [53]). For TCP, as the data stream delivery is guaranteed between the two hosts, we rely on the order of bytes in the TCP session between source and destination to identify the data flow at byte level, which can be uniquely identified using a pair of IP addresses and port numbers. Such tracking silently links the outbound traffic from the source host with the inbound traffic at the destination host, which does not incur additional traffic. Note that although TCP regulates the data stream order, the sender or receiver may run different numbers of system calls in sending and receiving the data. For example, the sender may perform five `writev()` system calls to send 10,000 bytes of data (2,000 bytes each call), while the receiver may conduct 10 `read()` calls (1,000 bytes each call) to retrieve the complete data. This is why counting sent or received bytes is necessary, instead of counting the number of system calls.

In the case of UDP, since the data delivery is not guaranteed, some UDP packets could be lost during transmission. So we cannot rely on the order of transferred bytes because the destination host has no knowledge of which data are supposed to arrive and which have been lost. To support UDP, we embed a small "cross-host" tag at each `send` related system call by the source host, and parse the tag at `receive` related system calls by the destination host. The tag is inserted into the beginning of the datagram as a part of the user datagram before the checksum is calculated. If the datagram is transferred successfully, RTAG knows a certain length of data goes from the source to the destination. If the destination host finds the received datagram is broken, or totally lost, it will discard this datagram, hence RTAG is also aware of the loss and erases this inbound data from the reachability analysis and DIFT. As we will show in §8, the communication cost for TCP case is 0, while the cost for UDP is also marginal in the benchmark measurement.

The cross-host tag represents the byte-level data in the socket communication between two processes across hosts. Each tag key represents the data traffic in one socket session using the source and destination process credentials, plus the offset that indicates the data at byte level. For the uniqueness of session, we use the process identifier (`pid`) and the process creation time (`start_time` in the `task` structure) to identify each process. The tag values represent the origin of the tag key, which is determined by the DIFT and updated to the global tag map. The cross-host tags are also switched away before DIFT is performed and restored afterward. For the hosts on which RTAG does not run, we treat them as a black box, and identify them using the IP address and port number. The IP and port are retrieved from the `socket` structure inside the kernel.

Handling IPC. RTAG tracks the data transfer of IPC communication between two processes as well. For the IPC that uses system call as a controlling interface (e.g., `pipe`, and System V IPC: `message queues`, `semaphores`), RTAG hooks these system calls to track the data being transferred. When a process uses `pipe` to send data to the child process, RTAG monitors the `read` and `write` system calls to track the transferred data in bytes. During reachability analysis, we create tag keys to label every byte sent from the parent to the child. The tag values are fulfilled by DIFT. For example, in Figure 2, although the `git pack` and `ssh` processes have IPC dependency, RTAG is able to perform the replay and DIFT independently on them since RTAG caches the inbound data reads from the pipe and feeds them back during the replay. Also, by tracking the `inode` associated with the file descriptors (rather than tracking `pipe, dup(2)` and child inheritance relationships), we identify the data transmitted via the pipe at byte level and the processes at its two ends. RTAG *implicitly* tracks the IPC based on shared memory. Instead of trapping the replay of a process for each read from a shared memory, RTAG replays the processes having shared memory as a *group* as RAIN [35] and Arnold [25] do, so that the tag propagation of this shared memory is performed within the process' memory locations. No separate tag allocation is needed for these processes.

6.7 Query Results

The query result will be returned after all the tag values of the interfering data are updated. The result represents the data causalities of involved objects in a tree structure. For example, in Figure 2, a backward query on the attacker's controlled host `5.5.5.5:22` will return the tree-shape data flow overlay depicted in Figure 2(b), consisting of all the segments of the flow from the key to all of its upstream origins. Also, a forward query returns every segment of the data flow from the queried tag key to all of its impact(s). It relies on a *reversed* map where the tag key and value are swapped to locate the downstream impact from a file. For example, a forward query on the private key `id_rsa` on the client side returns a flow: `id_rsa`→`results.v1`→`objects`→`5.5.5.5:22`. A point-to-point query gives the detailed data flow between two nodes in the provenance graph by performing a forward and backward query on these two nodes, then computing the intersection of the two resulting trees.

7 Implementation

The implementation of RTAG is based on a single-host refinable information flow tracking system RAIN [35], with extended development of the tagging system. Specifically, our implementation adds 830 lines of C code to the Linux kernel for the tag association module, 2,500 lines of C++ code to the DIFT engine for the tag switch mechanism, 1,100 lines of C++ code for the maintenance of tags, 900 lines of C++ code for the query handler, and 500 lines of Python code for the reachability analysis for tag allocation. Currently, RTAG runs on both the 32-bit and 64-bit Ubuntu 12.04 LTS. Accordingly, our DIFT engine supports both x86 and x86_64 architectures, which is based on libdft [37] and its extended x86_64 version from [43]. We use a graph database Neo4j [10] for storing and analyzing coarse-level provenance graphs, and a relational database PostgreSQL [3] for global tags with multiple indexing on host (i.e., MAC address) and file credentials (i.e., `inode`, `dev`, `crtime`). Particularly, we supplement the tag data structure §6.4 and how we track socket session §6.6 with implementation details in the following.

Tag Data Structure. In the current implementation, RTAG maintains local tags for individual bytes. RTAG uses C++'s `vector` as the multi-tag container for one memory location or register and uses sorting and binary search in the case of `insert` operation. `vector` has storage efficiency, although its insertion overhead is higher than that of the `set` data structure, which was used by DataTracker [61]. We make this choice based on x86 instruction statistics [4] that show the most popularly used instructions are `mov`, `push`, and `pop` of which the propagation policy copies the tag(s), while instructions that involve insertion, such as `add` and `and`, are much less frequent. Our evaluation affirms this choice that the time overhead for single DIFT is similar between RTAG and previous work [61].

Tracking Socket Session. The implementation of tracking the socket communication session refers to the `socket` structure inside the kernel for IP and port of the host and the peer. If the type of socket is `SOCK_STREAM` (i.e., TCP), we use a counter counting the total number of bytes sent or received by tracking the return value of `send` or `write` system calls. If the type is `SOCK_DGRAM` (i.e., `UDP`), our implementation embeds a four-byte incrementing sequence number within the same peer IP and port number at the beginning of the payload buffer inside an in-kernel function `sendmsg` rather than the system call functions such as `send` and `recv` to avoid affecting the interface to the user program as well as the checksum computation. At the receiver side, we strip the sequence number in the `recvmsg` after the checksum verification and present the original payload to the program. As shown in §8.2.3, the hooking at this level incurs almost no overhead in either bandwidth or socket handling time. It also avoids the complicated fragmentation procedure at the lower level.

8 Evaluation

Our evaluation addresses the following questions:

- How well does RTAG handle the data flow queries (forward, backward, and point-to-point) for cross-host attack investigations? (§8.1)
- How well does RTAG improve the efficiency of DIFT-based analysis in terms of time and memory consumption? (§8.2.1)
- How much overhead does RTAG cause to system runtime including the network bandwidth? (§8.2.2, §8.2.3) What is the storage footprint of running RTAG? (§8.2.4)

Settings. We run RTAG based on the Ubuntu 12.04 64-bit LTS with 4-core Intel Xeon CPU, 4GB RAM and 1TB SSD hard drive on a virtual machine using `KVM` [14] for the target hosts where system-wide executions are recorded. On the analysis host, we use a machine with 8-core Intel Xeon CPU W3565, 192 GB RAM, and 2TB SSD hard drive installed with Ubuntu 12.04 64-bits for handling the query and performing DIFT tasks in parallel. We use `NFS` [15] to share the log data between the target and the analysis host.

8.1 Security Applications

Table 1 summarizes the statistics in every stage of processing a query for an attack investigation: the original provenance graph covering all the hosts, the pruned graph where the unrelated causalities are filtered out by the reachability analysis, and the data flow overlay where the tags store the origins of each byte of data involved in the query. Table 2 also summarizes how long each of the queries took and their memory consumption.

8.1.1 GitPwnd

We first present how RTAG handles the queries on the Gitpwnd example (described in §3.1). To handle a query, we replay the involved processes independently based on reachability analysis results while performing DIFT on the interfering parts. We run RTAG on both client and server hosts involved in this attack, while treating the attacker-controlled host as a black box. We perform three queries: a forward query asking for where the leaked `/etc/passwd` goes to, a backward query inquiring the sources of data flow that reaches the attacker's controlled host, and a point-to-point query aiming to particular data

Items		Prov Graph		Pruned Graph		DF Overlay		Accuracy
Attack	**Query**	**Node**	**Edge**	**Node**	**Edge**	**Tags**	**C-Tags**	
GitPwnd	FW: /etc/passwd	8.3K	109K	39	557	**28,960**	**10,700**	**100%**
	BW: attacker host			55	1,661	**32,660**	**18,032**	**100%**
	PP: results - objects			22	418	**23,193**	**7,317**	**100%**
SQLi-1	FW: exploit html	5.3K	89K	33	711	**6,799**	**882**	**100%**
	BW: payroll record			29	683	**8,257**	**882**	**100%**
	PP: html - db file			27	490	**3,197**	**882**	**100%**
SQLi-2	FW: db file	5.2K	87K	80	2,251	**510,466**	**420,121**	**100%**
	BW: dump file			72	1,997	**530,004**	**420,121**	**100%**
CSRF	FW: exploit html	2.8K	34K	89	2,379	**9,224**	**1,766**	**100%**
	BW: salary record			97	2,270	**7,700**	**1,766**	**100%**
XSS	FW: exploit html	2.9K	24K	71	1,145	**432,845**	**420,755**	**100%**
	BW: attacker host			63	863	**435,716**	**420,700**	**100%**
	PP: html - a-host			55	782	**421,106**	**420,700**	**100%**
P2P	BW: mp4@12th node	13K	730K	74	240K	**759,302**	**630,228**	**100%**
	FW: mp4@1st node			182	490K	**3,088,102**	**2,532,920**	**100%**

Table 1: Statistics in terms of the effectiveness and performance of cross-host attack investigation. **Prov Graph** are the original graph containing the system-wide executions of every process. **Pruned Graph** are the subgraph where nodes and edges that are unrelated to the attack are pruned out; **DF Overlay** are results from the RTAG tagging system; **Tags** gives the number of generated tag entries; **C-Tags** gives the number of tags of which the key and value(s) are **C**ross-host (i.e., from different hosts); **Accuracy** shows the percentage of how many data flows are matched with the ground truth.

flow paths between the `results` file on the client side and the `objects` file on the server side. In Table 1, we show the statistics of using RTAG in every step. Particularly, we show the number of tags RTAG creates at the tag overlay. In the forward query, RTAG generates 28,960 tag entries totally, 10,700 of which are cross-host ones meaning the tag key and value are from different hosts. We compare the query result with ground truth of the attack and RTAG achieves 100% accuracy in every query. We also evaluate the performance improvement for DIFT, summarized in Table 2. In general, thanks to the parallelizing of DIFT tasks, RTAG reduces the time cost by more than 70% in most cases.

8.1.2 Web-based Attacks

We also use a set of web-based attacks to evaluate the effectiveness of RTAG in tracking the data flow between the server (e.g., a web server `Apache`), and the client (e.g., a browser `Firefox`). The web app facilitates the checking and updating of employees' personal financial information. The employees typically manage their bank account number and routing number via the web app. The attacks include two SQL injections, one cross-site request forgery (CSRF), and one cross-site scripting (XSS). We set up RTAG on both server and client. We run an `Apache` server with `SQLite` as its database. At the client, we load exploit pages with either a data transfer tool `Curl` or the `Firefox` browser. For each attack, we perform three types of queries and compare the query results with the ground truth.

Items		DIFT Perf			
Attack	**Query**	**Tasks**	**Mem(MB)**	**Time(s)**	**TReduc%**
GitPwnd	FW	10	497	95	**87%**
	BW	27	912	113	**86%**
	PP	8	322	79	**72%**
SQLi-1	FW	14	2,513	342	**70%**
	BW	11	2,336	339	**64%**
	PP	9	1,997	309	**76%**
SQLi-2	FW	41	7,655	695	**83%**
	BW	39	6,804	677	**82%**
CSRF	FW	33	6,537	499	**78%**
	BW	49	7,122	504	**84%**
XSS	FW	26	4,850	687	**77%**
	BW	28	5,391	705	**77%**
	PP	19	4,107	677	**72%**
P2P	BW	12	6,371	201	**92%**
	FW	12	9,855	236	**91%**

Table 2: DIFT performance using RTAG. **Tasks** stands for the number of processes that are replayed with DIFT; **Memory** gives the sum of virtual memory cost for each task; **Time** gives the time duration RTAG spends to perform the DIFT tasks in parallel; **TReduc%** shows the reduction rate from the time of performing the same DIFT tasks serially.

SQL injections. The exploit takes advantage of a vulnerability at the server's SQL parsing filter to execute illegal query statements that steal or tamper the server database. The first attack (**SQLi-1**) injects an entry of user profile to the database. The added profile is further used by another

financial program to generate payroll records. The analyst performs a forward query from the loaded `html` file with the exploit, and RTAG returns the data flows from the file at the client to the data in the payroll records. The second attack (**SQLi-2**) steals data entries in the database from the user and exploits a vulnerability in `Firefox` to dump the entries to a file. With a backward query from the dump file at the user side, RTAG pinpoints the segments of the database file that has been exfiltrated.

Cross-site request forgery. The exploit uses a vulnerability of the server that miscalculates the CSRF challenge response to submit a form impersonating the user. The form updates the profile contents (e.g., account number), and later the tampered profile is accessed by several other programs that process the user's payroll information. RTAG helps determine the data flow between the user's loaded file and one of the payroll record that is considered to have been tampered.

Cross-site scripting. The reflection-based cross-scripting relies on dependency of an html element to user input to append a script that reads the sensitive data from the `DOM` tree of a page, packs some of the data, and sends an email to the attacker's external host. After the investigation determines the attacker's host to be malicious, it makes a backward query from that host and finds the data exfiltration from the user's loaded page, as well as from a certain offset of the database storage file at the server. Notably, the resulting overlay shows the route of some tags tracing back first to the server side (i.e., `Apache`), then further back to the client side browser and the exploit html file, which recovers the *reflection* nature of the attack.

8.1.3 Attacks Involving Memory Corruptions

To evaluate RTAG for the cases when the attacker exploits memory corruptions, we additionally modified the Git-Pwnd attack §3.1 by compiling the `ssh` daemon with earlier versions containing memory-based vulnerabilities: one integer overflow based on CVE-2001-0144 and one buffer overflow based on CVE-2002-0640. For the integer overflow, we patched the `ssh` client side code to exploit the vulnerability [1] and remotely executed `scp` command at the server to copy files to the attacker's controlled host. For the buffer overflow, we crafted a malicious response for the OpenSSH (v3.0) challenge-response mechanism and remotely executed commands [2]. We note that memory-corruption-based attacks usually involve undefined behavior of the program that violates the assumption of many previous investigation systems using source or binary semantics (e.g., [34, 42, 47]). However, RTAG successfully reconstructs the program state of the overflow for the DIFT to recover the fine-grained data flow.

8.1.4 File Spreading in Peer-to-Peer Network

We also run RTAG to track the data flows in a malicious-file-spreading incident on top of a P2P network, which is regarded an increasing threat in the decentralized file sharing, according to a report by BitSight Insight [5]. This allows us to demonstrate RTAG's ability to handle a complex cross-host data-flow analysis involving multiple parties, which is infeasible with existing approaches. We use Gtk-Gnutella [7](v1.1.13) to set up a P2P network in a local network of 12 nodes with RTAG running on them. We perform two operations. First, we have two nodes online; one node shares a malicious audio `mp4` file, and another node searches for the file, discovers it and downloads it. Later, we shutdown the first node and let a third node download the file from the second node. We performed this type of single-hop relay iteratively until five nodes have this file. Second, we use these five nodes as "seeds" and let the remaining nodes search, discover, and download the file. During this process, we intentionally shutdown parts of the nodes to introduce "resume" procedures. Finally, we perform a backward query from the audio file at the last node to search for the origin of the file, and a forward query from the first node to uncover how the file spread across the network with fine-grained-level data flows. RTAG returns the results with 100% accuracy. Particularly, the result also shows the data flow between each pair of nodes for each iteration of the file sharing procedure. The statistics of this experiment are summarized in Table 1.

8.2 Performance

8.2.1 DIFT Runtime Performance

We compare the memory consumption and execution time of RTAG with previous DIFT systems. For the memory efficiency, we evaluated two state-of-the-art DIFT engines that provide multi-color symbols, Dytan [24] and DataTracker [61]. Table 3 shows the peak memory consumption of the tag map for various DIFT tasks we used in evaluating the security application in §8.1. The peak memory consumption is useful as it indicates the required resource for a certain type of DIFT. Notably, all the tag sizes for representing the DIFT symbols determined by reachability analysis are within three bytes (i.e., up to 16,777,216 symbols), with a majority being two bytes (i.e., up to 65,536 symbols). This means the data pruning and reachability analysis effectively narrow down the scope of the DIFT symbols and pinpoint the exact bytes of data that causes the data confusion for DIFT to resolve. The savings from the tag map consumption of RTAG is between 70% and 95%. The effect of improvement on the general memory consumption varies across different programs in terms of their own memory usage.

Programs	#Symbols	Peak TagMap Cost (MB)			Reduc%
		DataTracker	Dytan	RTAG	
git-core	247	12	19	**4.8**	**60 / 74**
ssh	16,983	5.9	630	**2.6**	**55 / 99**
cli-hook	1,983	17	140	**8.0**	**53 / 94**
Curl	56,010	4.8	1,050	**2.3**	**52 / 99**
Firefox	4,091,773	155	NA	**67.5**	**56 / NA**
Apache	2,128,700	133	NA	**41.7**	**68 / NA**

Table 3: DIFT Tag Map Overhead in Practice. **#Symbols** denotes the number of symbols used in performing the DIFT task; **NA** means the DIFT is not complete so the peak memory cost is not available.

In our experiments, DIFT reduced total memory usage 10% to 50% when compared with DataTracker [61], and by 30% to 90% compared with Dytan [24]. Since these DIFT systems are designed with the scope of one host, in order for proper comparison against previous DIFT systems, we only measured the cases where all the tags are within one host. Note that this approach only compares DIFT runtime performance side by side, but does not indicate or suggest that RTAG can only handle single-host cases. For evaluating the time efficiency in performing DIFT tasks, we assign the same DIFT tasks to RTAG as well as to the DIFT engine used by RAIN [35]. Since RAIN [35] does not support cross-host investigation, we use RAIN [35] to run the DIFT tasks, sequentially simulating the time consumption it needs to serialize the network interaction and orchestrating the replays. We observe that the parallel DIFT of RTAG takes 60%–90% less time than RAIN [35] (Table 2).

Discussion. For the memory consumption, we find the taint propagation is mainly composed of copy operations such that the tag map is just updated with another value. Combination operation for merging the tags of two locations is not frequent. Hence, though bit-vector (used in [24]) ensures a constant length of tag for each location even after combination, the benefit is not obvious. On the contrary, its fixed size is linear to the number of symbols, which causes out-of-memory crash when there are many symbols to tag or (and) the many memory locations are propagated during the execution. Using `set` eases the implementation complexity as it natively supports the combination operation with a good performance. However, it incurs higher metadata cost (on x86 Linux, storing every 4-byte data in the `set` incurs over 14 bytes). For the time consumption savings in RTAG, the total time consumption depends on the longest DIFT task (e.g., `Firefox` session). We are looking into integrating in-process parallel DIFT techniques to RTAG that could further bring down the time consumption.

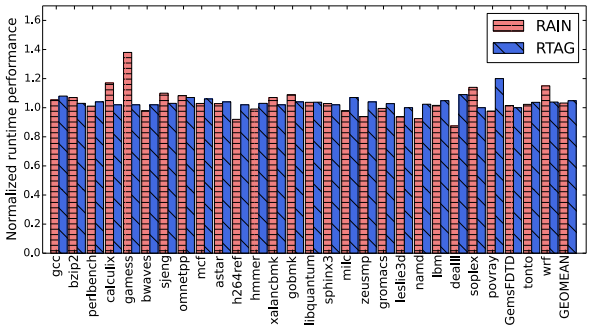

Figure 4: Comparison of normalized runtime performance between RAIN [35] and RTAG with CPU bound benchmark SPEC CPU2006. "GEOMEAN" gives the geometric mean of the performance numbers.

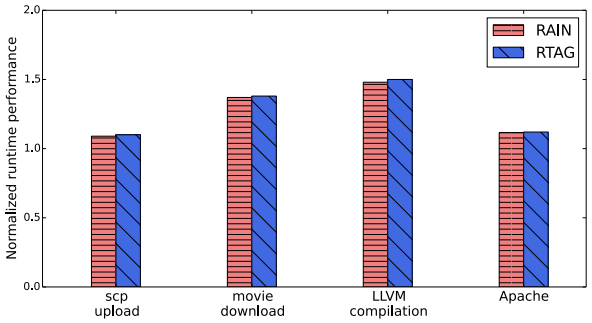

Figure 5: Comparison of normalized runtime performance between RAIN [35] and RTAG with IO bound benchmarks.

8.2.2 Runtime Overhead

We measure the runtime overhead of RTAG using two sets of benchmarks: the SPEC CPU2006 benchmark for CPU-bound use cases and the IO-intensive benchmarks for IO bound cases. The measurements are performed on two systems, one without RTAG and one with RTAG enabled. The result of SPEC benchmark is given in Figure 4 with RAIN [35] as reference. The geometric mean of the runtime overhead is 4.84%, which shows RTAG has similar low runtime overhead to previous refinable systems. We also measure the runtime overhead using IO-intensive applications to test the performance in IO bound cases. The benchmark is composed of four scenarios: using `scp` to upload a 500MB archive file, using `wget` downloading a 2GB `mov` movie file, compiling LLVM 3.8, and using `Apache` to serve an `http` service for file downloading. The result of IO-intensive applications is shown in Figure 5. The overhead of all the items is at most 50%. We reason that the cause of the higher overhead during file downloading and compiling is because network and file inputs are cached during the recording time.

Protocol	Setting	Bandwidth%	RTT%
TCP	Window: 128KB	0%	+0.03%
	256KB	0%	+0.01%
	512KB	0%	+0.012%
UDP	Buffer: 512B	-0.8%	+0.02%
	8KB	-0.05%	+0.01%
	128KB	-0.01%	+0.012%

Table 4: Bandwidth impact of RTAG. The bandwidth and round-trip-time (**RTT**) are measured with iperf3 benchmark using different settings for TCP and UDP protocols.

8.2.3 Network Performance Impact

We use iperf3 [13] to test the bandwidth impact of applying RTAG to typical network protocol settings. For TCP, we measure the bandwidth both with and without having RTAG running at different window sizes. For UDP, we set the buffer size to be similar with real applications such as DNS (512B), RTP (128KB). We also measure the performance impact in the term of the end-to-end round-trip-time (RTT) for one datagram to be delivered to the server and echoed back to the client. Both impacts are negligible. The results are summarized in Table 4.

8.2.4 Storage Footprint

As a refinable system, RTAG has the storage overhead for the non-deterministic logs that are used for faithful replay of the recorded system-wide process executions. This ensures the completeness of retroactive analysis particularly for the advanced low and slow attacks. The storage footprint varies according to the workload on each host and is comparable with the upstream system RAIN [35]. Note that only the input data are stored as non-determinism, thus in the multi-host case, the traffic from a sender to a receiver are only stored at the receiver side, avoiding duplicated storage usage. In the use of RTAG, we observe around 2.5GB–4GB storage overhead per day for a desktop used by a lab student (e.g., programming, web browsing); and around 1.5GB storage overhead per day for a server hosting gitolite used internally by five lab students for version controlling on course projects.

9 Related Work

Dynamic Information Flow Tracking. Dynamic taint analysis [24, 29, 37, 49, 62] is a well-known technique for tracking information flow instruction by instruction at the runtime of a program without relying on the semantic of a program source or binary. DIFT is useful for policy enforcement [49], malware analysis [66], and detecting privacy leaks [29, 62]. To support intra-process tainting,

DIFT Systems	Cross Host	Inst Time	Tag Dep	Run Over	DIFT Over(T/M)
Dytan [24]	×	Runtime	Inlined	High	High/High
DataTracker [61]	×	Runtime	Inlined	High	High/High
Panorama [66]	×	Runtime	Inlined	High	High/High
ShadowReplica [34]	×	Runtime	Inlined	High	Low/High
Taintpipe [47]	×	Runtime	Inlined	High	Low/High
Panda [27, 28]	×	Replay	Inlined	High	High/High
Arnold [25]	×	Replay	Inlined	Low	High/High
RAIN [35]	×	Replay	Inlined	Low	High/High
Jetstream [55]	×	Replay	Inlined	Low	Low/High
TaintExchange [67]	✓	Runtime	Inlined	High	High/High
Cloudfence [50]	✓	Runtime	Inlined	High	High/High
RTAG	✓	**Replay**	**Decoupled**	**Low**	**Low/Low**

Table 5: Comparison of DIFT-based provenance systems. "**Cross Host**" tells whether the system covers cross-host analysis; "**Inst Time**" represents when the instrumentation is performed (i.e., runtime or replay); "**Tag Dep**" shows how the tag dependency is handled; "**Run Over**" shows the runtime overhead; "**DIFT Over(T/M)**" presents the overhead of performing DIFT in terms of **T**ime and **M**emory cost in which RTAG both achieves reductions significantly.

Dytan [24] provides a customizable framework for multi-color tags. DataTracker adapts standard taint tracking to provide adequate taint marks for provenance tracking. However, taint-tracking suffers from excessive performance overhead (e.g., the overhead of one state-of-the-art implementation, libdft [37] is six times as high as native execution), which makes it difficult to use in a runtime environment. To solve this problem, several approaches have been proposed to decouple DIFT from the program runtime [34, 46, 47, 55, 57]. For example, Taintpipe [47], Straight-taint [46] and ShadowReplica [34] pre-compute propagation models from the program source and use them to speed up the DIFT at runtime. However, their dependency on program source disables these systems to analyze undefined behavior. In contrast to these DIFT systems, RTAG provides both efficient runtime (recording) and the ability to reliably replay and perform DIFT on the undefined behavior (e.g., memory corruptions) commonly seen in recent attacks. Jetstream [55] records the normal runtime execution and defers tainting until replay by splitting an application into several epochs. DTAM [30] uses dynamic taint analysis to find the relevant program inputs to its control flow and has a potential to reduce the workload of a record-replay system. Similar to RTAG, TaintExchange [67] and Cloudfence [50] provide multi-host information-flow analysis at runtime, but incur significant overhead (20× in some cases). We summarize the comparisons between RTAG and previous DIFT-based provenance systems in Table 5.

Provenance Capturing. Using data provenance [60] to investigate advanced attacks, such at APTs, has become a popular area of research [8, 31, 36, 39, 40, 42, 45, 48, 52]. For example, the Linux Audit System [8], Hi-Fi [52], and PASS [48] capture system-level provenance with less than

10% overhead. Linux provenance modules (LPM) [19] allows developers to develop customized provenance rules to create Linux Security Modules and LSM-like modules. SPADE [31] decouples the generation and collection of provenance data to provide a distributed provenance platform, and ProvThings [63] generates provenance data for IoT devices. Unfortunately, these systems are restricted to coarse-grained provenance, which generate many false dependencies. To reduce false positives and logging sizes, Protracer [45] improves BEEP [42] to switch between unit-level tainting and provenance propagation. In contrast, MCI [40] determines fine-grained dependencies ahead-of-time by inferring implicit dependencies using LDX [39] and creating causal models. DataTracker [61] leverages DIFT to provide fine-grained data, but incurs significant overhead. Finally, RAIN [35] uses record and replay to defer DIFT until replay, then uses reachability analysis to refine the dependency graph before tainting. However, none of these systems can provide fine-grained cross-host provenance like RTAG because they have no tag association mechanism to support cross-host DIFT.

Network Provenance. In addition to system-wide tracking, provenance at network level is a well-researched area [64, 68, 69]. For example, ExSPAN [69] provides a distributed data model for storing network provenance. One challenge network provenance faces is that it obviously cannot detect most system-level causality on end nodes. Technically, network provenance and RTAG are orthogonal to each other, so that we can use both approaches together to further enhance attack detection.

Record Replay System. Deterministic record-and-replay has been a well-researched area [17, 20, 26, 41, 56]. In addition to providing faithful replay, the current state-of-the-art techniques allow instrumentation of programs during the replay of execution [23, 25, 27]. Arnold [25] provides efficient runtime because it is a kernel based solution and can efficiently record nondeterministic events. Aftersight [23] and PANDA [27] are hypervisor-based solutions. Aftersite is based on VMware hypervisor (record) and QEMU (replay) while PANDA is purely based on QEMU. Similar to RAIN [35], RTAG leverages Arnold to provide efficient recording performance, however the goals and functionality of RTAG are unique from to Arnold and could be implemented on other systems.

10 Conclusion

When investigating information flow-based cross-host attacks, analysts need to manually analyze the information flow generated by the processes running on multiple hosts. This is a time consuming, error prone, and challenging task, due to the high number of processes and consequently flows involved. To help analysts in this task, we propose RTAG, a system for accurate and efficient information flow analysis that makes cross-host attack investigation practical. We implemented and empirically evaluated RTAG by using the system to analyze a set of real-world attacks including GitPwnd, a state-of-the-art cross-host data infiltration attack. The system was able to provide accurate results while reducing memory consumption by 90% and also reducing the time consumption by 60-90% compared to related work. We have a plan to release the source code of RTAG.

We foresee several directions for future work. First, we plan to make hosts running RTAG interoperable with hosts not running the system. To do so, we plan to embed tag information in an optional field of the UDP header. Second, we plan to identify information flow techniques that are resilient to the fact that RTAG might not be running on every host in a given network. Third, we plan to integrate in-process parallel DIFT techniques to RTAG to further optimize the analysis time. Fourth, we plan to reduce the storage requirement for non-deterministic inputs. To do so, we plan to investigate ways to optimize the storage of similar executions across different hosts. Finally, we plan to extend the queries supported by RTAG so that it is possible to compare the information flow associated with different executions of the same program. In this way, it will be possible to pinpoint when and where a program was compromised.

11 Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research was supported in part by NSF, under awards CNS-0831300, CNS-1017265, CCF-1548856, CNS-1563848, CRI-1629851, CNS-1704701, and CNS-1749711, ONR, under grants N000140911042, N000141512162, N000141612710, and N000141712895, DARPA TC (No. DARPA FA8650-15-C-7556), NRF-2017R1A6A3A03002506, ETRI IITP/KEIT [2014-0-00035], and gifts from Facebook, Mozilla, and Intel.

References

[1] Ssh 1.2.x - crc-32 compensation attack detector, Feb. 2001. https://www.exploit-db.com/exploits/20617.

[2] Openssh 3.x - challenge-response buffer overflow, 2002. https://www.exploit-db.com/exploits/21578.

[3] Postgresql, Oct. 2014. https://www.postgresql.org.

[4] x86 machine code statistics, Oct. 2014. https://www.strchr.com/x86_machine_code_statistics.

[5] Peer-to-peer peril: How peer-to-peer sharing impacts vendor risk and security benchmarking, Dec. 2015. https://info.bitsighttech.com/how-peer-to-peer-file-sharing-impacts-vendor-risk-security-benchmarking.

[6] Event tracing for windows, Oct. 2017. https://docs.microsoft.com/en-us/dotnet/framework/wcf/samples/etw-tracing.

[7] Gtk-gnutella, Oct. 2017. http://gtk-gnutella.sourceforge.net.

[8] Linux audit, Oct. 2017. https://linux.die.net/man/8/auditd.

[9] Mozilla rr, Oct. 2017. http://rr-project.org.

[10] Neo4j graph database, Oct. 2017. http://neo4j.com.

[11] Git: a free and open source distributed version control system, Feb. 2018. https://git-scm.com/.

[12] Gitolite, Feb. 2018. https://www.gitolite.com.

[13] iperf3, Feb. 2018. https://iperf.fr.

[14] Kernel-based virtual machine, Oct. 2018. https://www.linux-kvm.org.

[15] Linux network file system, Oct. 2018. http://nfs.sourceforge.net/.

[16] Taintgrind: a valgrind taint analysis tool, Feb. 2018. https://github.com/wmkhoo/taintgrind.

[17] BACON, D. F., AND GOLDSTEIN, S. C. *Hardware-assisted replay of multiprocessor programs*. Santa Cruz, CA, 1991.

[18] BATES, A., BUTLER, K., HAEBERLEN, A., SHERR, M., AND ZHOU, W. Let sdn be your eyes: Secure forensics in data center networks. In *2014 NDSS Workshop on Security of Emerging Network Technologies (SENT)* (2014).

[19] BATES, A., TIAN, D. J., BUTLER, K. R., AND MOYER, T. Trustworthy whole-system provenance for the Linux kernel. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[20] BURTSEV, A., JOHNSON, D., HIBLER, M., EIDE, E., AND REGEHR, J. Abstractions for practical virtual machine replay. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Atlanta, GA, 2016).

[21] CHEN, A., MOORE, W. B., XIAO, H., HAEBERLEN, A., PHAN, L. T. X., SHERR, M., AND ZHOU, W. Detecting covert timing channels with time-deterministic replay. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[22] CHEN, P., DESMET, L., AND HUYGENS, C. A study on advanced persistent threats. In *IFIP International Conference on Communications and Multimedia Security* (2014), Springer, pp. 63–72.

[23] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference (ATC)* (Boston, MA, June 2008).

[24] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (London, UK, July 2007).

[25] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[26] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News* (New York, NY, 2009), ACM.

[27] DOLAN-GAVITT, B., HODOSH, J., HULIN, P., LEEK, T., AND WHELAN, R. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW)* (2015).

[28] DOLAN-GAVITT, B., LEEK, T., HODOSH, J., AND LEE, W. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)* (Berlin, Germany, Oct. 2013).

[29] 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. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[30] GANAI, M., LEE, D., AND GUPTA, A. Dtam: dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (Cary, NC, Nov. 2012).

[31] GEHANI, A., AND TARIQ, D. SPADE: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference (Middleware)* (2012).

[32] GIBLER, C., AND BEDDOME, N. Gitpwnd, Oct. 2017. https://github.com/nccgroup/gitpwnd.

[33] GOMEZ, L., NEAMTIU, I., AZIM, T., AND MILLSTEIN, T. Reran: Timing-and touch-sensitive record and replay for android. In *Software Engineering (ICSE), 2013 35th International Conference on* (2013).

[34] JEE, K., KEMERLIS, V. P., KEROMYTIS, A. D., AND PORTOKALIDIS, G. ShadowReplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)* (Berlin, Germany, Oct. 2013).

[35] JI, Y., LEE, S., DOWNING, E., WANG, W., FAZZINI, M., KIM, T., ORSO, A., AND LEE, W. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 24rd ACM Conference on Computer and Communications Security (CCS)* (Dallas, Texas, Oct. 2017).

[36] JI, Y., LEE, S., AND LEE, W. RecProv: Towards provenance-aware user space record and replay. In *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)* (Mclean, VA, 2016).

[37] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., AND KEROMYTIS, A. D. 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, 2012).

[38] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Recovering from intrusions in distributed systems with DARE. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)* (Seoul, South Korea, July 2012).

[39] KWON, Y., KIM, D., SUMNER, W. N., KIM, K., SALTAFORMAGGIO, B., ZHANG, X., AND XU, D. LDX: Causality inference by lightweight dual execution. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, Apr. 2016).

[40] KWON, Y., WANG, F., WANG, W., LEE, K. H., LEE, W.-C., MA, S., ZHANG, X., XU, D., JHA, S., CIOCARLIE, G., ET AL. Mci: Modeling-based causality inference in audit logging for attack investigation. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2018).

[41] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2010), SIGMETRICS '10.

[42] LEE, K. H., ZHANG, X., AND XU, D. 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, Feb. 2013).

[43] LONG, F., SIDIROGLOU-DOUSKOS, S., AND RINARD, M. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 227–238.

[44] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation.

[45] MA, S., ZHANG, X., AND XU, D. 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, Feb. 2016).

[46] MING, J., WU, D., WANG, J., XIAO, G., AND LIU, P. StraightTaint: decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Singapore, Sept. 2016).

[47] MING, J., WU, D., XIAO, G., WANG, J., AND LIU, P. TaintPipe: pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[48] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC)* (Boston, MA, May–June 2006).

[49] NEWSOME, J., AND SONG, D. 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, Feb. 2005).

[50] PAPPAS, V., KEMERLIS, V. P., ZAVOU, A., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Cloudfence: Data flow tracking as a cloud service. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (Saint Lucia, Oct. 2013).

[51] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. 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)* (2010).

[52] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K. Hi-fi: collecting high-fidelity whole-system provenance. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2012), pp. 259–268.

[53] POSTEL, J. User datagram protocol.

[54] POSTEL, J. Transmission control protocol.

[55] QUINN, A., DEVECSERY, D., CHEN, P. M., AND FLINN, J. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, Nov. 2016).

[56] REN, S., TAN, L., LI, C., XIAO, Z., AND SONG, W. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)* (Denver, CO, June 2016).

[57] RUWASE, O., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., CHEN, S., KOZUCH, M., AND RYAN, M. Parallelizing dynamic information flow tracking.

[58] SAITO, Y. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging* (2005).

[59] SEN, K., KALASAPUR, S., BRUTCH, T., AND GIBBS, S. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013).

[60] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A survey of data provenance in e-science. *ACM Sigmod Record 34*, 3 (June 2005), 31–36.

[61] STAMATOGIANNAKIS, M., GROTH, P., AND BOS, H. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)* (Cologne, Germany, 2014).

[62] SUN, M., WEI, T., AND LUI, J. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016), ACM, pp. 331–342.

[63] WANG, Q., HASSAN, W. U., BATES, A., AND GUNTER, C. Fear and logging in the internet of things. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2018).

[64] WU, Y., ZHAO, M., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM Computer Communication Review* (Snowbird, Utah, USA, June 2014), vol. 44, pp. 383–394.

[65] YAN, M., SHALABI, Y., AND TORRELLAS, J. 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, Oct. 2016).

[66] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. 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, Oct.–Nov. 2007).

[67] ZAVOU, A., PORTOKALIDIS, G., AND KEROMYTIS, A. Taint-exchange: a generic system for cross-process and cross-host taint tracking. In *Advances in Information and Computer Security* (2011), Springer, pp. 113–128.

[68] ZHOU, W., FEI, Q., NARAYAN, A., HAEBERLEN, A., LOO, B. T., AND SHERR, M. Secure network provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, Oct. 2011).

[69] ZHOU, W., SHERR, M., TAO, T., LI, X., LOO, B. T., AND MAO, Y. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD/PODS Conference* (Indianapolis, IN, June 2010), ACM, pp. 615–626.