A New Tag-Based Approach for Real-Time Detection of Advanced Cyber Attacks

A Dissertation presented

by

Md Nahid Hossain

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

January 2022

Stony Brook University

The Graduate School

Md Nahid Hossain

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dr. R. Sekar - Dissertation Advisor
Professor, Department of Computer Science

Dr. Scott D. Stoller - Chairperson of Defense
Professor, Department of Computer Science

Dr. Michalis Polychronakis - Committee Member
Associate Professor, Department of Computer Science

Dr. Venkat Venkatakrishnan - External Committee Member
Professor, Department of Computer Science
University of Illinois at Chicago

This dissertation is accepted by the Graduate School

Eric Wertheimer
Dean of the Graduate School

Abstract of the Dissertation

A New Tag-Based Approach for Real-Time Detection of Advanced Cyber Attacks

by

Md Nahid Hossain

Doctor of Philosophy

in

Computer Science

Stony Brook University

2022

Abstract

We are witnessing a rapid escalation in targeted cyber-attacks, often called "Advanced and Persistent Threats" (APTs), carried out by skilled adversaries. By combining social engineering (e.g.,spear-phishing) with advanced exploit techniques, these adversaries routinely bypass widely-deployed software protections such as address space randomization. Consequently, enterprises have come to rely on second-line defenses such as security information and event management (SIEM) tools. While generally useful, these tools generate vast quantities of information, making it difficult for a security analyst to distinguish attacks from background noise. Moreover, analysts lack the tools to "connect the dots" to piece together fragments of an attack campaign that spans multiple applications, hosts, and time periods. It is no wonder that many APT campaigns go undetected for weeks to months.

Researchers have proposed the use of causal dependencies, also called provenance, to bring more automation to cyber attack detection. Provenance provides additional context to prune away false positives, and can link together disparate attack steps. However, a straight-forward application of provenance leads to campaign summaries that are many orders of magnitude larger than what can be visualized or understood by a cyber analyst. Moreover, provenance data consists of billions of events, posing major challenges for real-time analysis.

In this thesis, we first propose novel techniques that achieve two orders of magni-

tude reduction in the size of dependence graphs, while provably preserving analysis results. This makes it feasible to analyze scenarios consisting of tens of billions of events in main memory, where graph traversals can be implemented efficiently. To speed up detection and scenario reconstruction, we observed that these techniques typically compute and use global context at each graph node. We introduced the notion of tags to compactly summarize global context, and propagate these tags efficiently from ancestor nodes to descendant nodes using local computations. We have introduced several novel tags and propagation semantics, each offering different trade-offs in terms of efficiency and accuracy. Our experimental evaluation, carried out through several DARPA-sponsored red team exercises, demonstrates that our techniques are (a) effective in identifying stealthy attack campaigns, (b) reduce false alarm rates by more than an order of magnitude, and (c) yield compact attack scenarios consisting of tens to hundreds of events while sifting through event logs with tens to hundreds of millions of events.

Dedication

To my mother, Khursheed E Nasim
and
my father, Md Moazzem Hossain

Contents

List of Figures

Acknowledgments

First of all, I would like to express my deepest gratitude to my advisor Professor R. Sekar for his immense support during my entire PhD. His enthusiasm and attitude for research deeply influenced me and helped me grow as a researcher and made this thesis a reality. Without his guidance and motivation, this journey would not have been possible for me to undertake. He always pushed me to strive for excellence no matter what the task was. I am thankful for his patience and his constant belief in me.

I would also like to thank my committee members, Dr. Scott Stoller, Dr. Michalis Polychronakis, and Dr. Venkat Venkatakrishnan for their time serving on the committee of my thesis and for the great feedback and insightful comments they gave. Some of the works in this dissertation were done in collaboration with Professor Scott and Professor Venkat. I am truly grateful for their guidance and support not only during the collaboration but also throughout my PhD. On that note, I would also like to thank each and every one of the faculty and staff members from the Computer Science department at Stony Brook University with whom I came in contact throughout the years for their amazing support.

During my PhD, I was fortunate enough to collaborate and work with numerous outstanding researchers as part of the DARPA Transparent Computing program. I am truly fortunate to be a part of this program which helped shape my thesis by exposing me to tackle real-world cyber security challenges. I would like to acknowledge and thank my amazing collaborators, Sanaz Sheikhi, Sadegh Momeni Milajerdi, Rigel Gjomemo, and Birhanu Eshete. I would also like to thank Jayesh Ranjan and Vineeth Polamreddy for helping me design an amazing user interface for our system.

I owe my sincerest gratitude to all my labmates and colleagues at Stony Brook University. I am extremely fortunate that I was able to make some amazing friends during this journey whom I will be cherishing throughout my entire life. I would also like to thank all my friends and family from Bangladesh who supported me from thousands of miles away through all the ups and downs.

Most importantly, I would like to thank my mother. I was able to pass through every single hurdle of this journey because of her motivation and unwavering faith in me. Without her constant support and sacrifice, this thesis would not have been possible.

Chapter 1

Introduction

We are witnessing a rapid escalation in targeted cyber-attacks, often called "Advanced and Persistent Threats" (APTs) [2]. These attacks are carried out by skilled attackers, often backed by the resources of criminal organizations or nation states. Unlike traditional malware attacks that are indiscriminate, APTs choose their targets deliberately: often, the goal is to steal intellectual property, databases, credit card or user account information, or to gain access to physical equipment, infrastructure, or industrial control systems. By combining social engineering techniques (e.g., spear-phishing) with zero-day vulnerabilities and advanced exploit techniques, APT actors routinely bypass widely-deployed software protections such as ASLR, DEP and sandboxes. To defend against these threats, enterprises have come to rely on *state-of-the-art* intrusion detection systems (IDS), often called Security Information and Event Management (SIEM) systems, e.g., Splunk [11], LogRhythm [7], ArcSight [9] and IBM QRadar [5]. The heuristics and rules used by these SIEMs are typically based on a single event. Unfortunately, many of these heuristics can also be triggered by some benign activities, thereby leading to numerous false alarms.

Another key issue with existing SIEMs is the lack of tools for "connecting the dots," i.e., piecing together fragments of an attack campaign. APTs are long-running in nature and are carried out in multiple stages. The perpetrators of these attacks remain below the radar for long periods, while exploring the organization's IT infrastructure and exfiltrating or compromising sensitive data. For instance, attackers remained undetected for 2.5 months in the Equifax attack [1], while the attack in the OPM data breach [10] was carried out over 8+ months.

Ultimately when the attack is discovered, a forensic analysis is initiated to identify the entry points of the attack and its system-wide impact. Existing tools do support time-based event correlation. However, temporal correlation is not effective on APT campaigns due to their slow moving nature. This weakness in connecting the dots, when combined with the high volume of false alarms, overwhelms cyber analysts with a flood of low quality information. It is no wonder that most APT campaigns remain undetected for months.

To perform accurate forensic analysis, system wide activity logging is required across the entire enterprise. Logs should be detailed enough to track dependencies between events occurring on different hosts and at different times. Most contemporary operating systems provide tools for such logging, including the *auditd* auditing daemon for Linux and the *ETW* (Event Tracing for Windows) system for Microsoft Windows. Unfortunately, these logs can be very large, e.g., a single host can generate gigabytes of audit data in a single day [63, 162]. As a result, storage requirements can go up to the petabyte range per year for an enterprise with thousands of hosts. In addition to storage costs, this large volume also slows down forensic analysis enormously.

In summary, stealthy APT campaigns pose the following challenges to existing IDS tools and techniques:

- *Needle-in-a-haystack*: Due to the stealthy nature of APTs, today's systems employ a wide range of detectors that capture isolated bad behaviors. Unfortunately, many of these behaviors resemble benign activities as well. Given the rarity of attack events compared to the volume of benign events, existing systems generate numerous false positives. The resulting volume of false alarms can cause analysts to miss real attacks.

- *Connecting the dots*: APT attacks are conducted in multiple stages and run for months. Even if existing systems can identify individual malicious events, they don't provide much help in understanding the overall "big picture." To respond to sophisticated APTs in real-time, techniques are needed to connect these disparate events to uncover the overall campaign.

- *Scaling and performance*: Even a relatively small enterprise with hundreds of hosts can generate terabytes of audit and security event logs every day [63, 162]. Many detection techniques and analysis tasks require sifting through billions of log records. These analyses can be very expensive to perform, making it difficult to meet real-time goals.

The above factors motivate the need for new attack detection and analysis techniques that will enable an analyst to identify an ongoing attack campaign quickly. These new techniques need to have a low false alarm rate along with the ability to store and analyze vast quantities of logs produced in an enterprise. Finally, the entire attack campaign should be presented in real-time to cyber analysts using a compact graphical representation.

Provenance-based Systems and Forensic Analysis

Researchers have proposed the use of causal dependencies or provenance to connect together attacker activities. *Backtracker* [72] used coarse-grained provenance from

system-call logs to construct a provenance graph of system events. Subsequent research [86, 98] focused on improving the precision of these provenance graphs using fine grained provenance information. To reconstruct the entire attack scenario from a system event that corresponds to an attack, the following steps are taken:

- *Backward Analysis:* A backward graph search from the attack event is initiated to identify the potential entry point (e.g., IP address) of this attack.
- *Forward Analysis:* Once the entry point is identified, a forward search from the entry point can be used to understand the totality of the attacker's activities.

These *backward* and *forward analyses* serve to connect all of the attacker's steps, thus presenting the overall attack scenario to a cyber analyst. We refer to this process as *attack scenario reconstruction*.

Challenges of Provenance-based Techniques

A major drawback of provenance based systems such as *Backtracker* is that the attack scenario reconstruction is performed in a purely forensic setting due to the massive size provenance graphs [38, 87, 96, 147, 162]. Forward and backward reachability analyses on such massive graphs is extremely slow. Moreover, optimizations such as precomputing/caching don't work on provenance graphs. This is because reachability changes over time, e.g., a process that had no communication with an attacker site at one point in time may end up communicating with that site later on.

Another major challenge faced by provenance-based techniques is the *dependence explosion* problem. Most forensic analysis techniques [55, 61, 72, 92, 105] operate on the basis of coarse-grained provenance. In particular, if a subject (i.e., a process) reads from a network source, then all subsequent writes by the subject are treated as (potentially) dependent on the network source. This leads to a *dependence explosion,* as every output of a process becomes dependent on every earlier input operation. The impact of this explosion is severe for long-running processes such as web browsers, email readers, and network servers. Unfortunately, such long-running processes generate the bulk of the activities on today's systems. Consequently, a straight-forward application of forward analysis can result in a graph with millions of nodes, a size far too large to be understood by an analyst. Although fine-grained information flow tracking [18, 65, 66, 70, 97, 98, 113, 160] can mitigate dependence explosion, these techniques suffer from large performance overheads that discourage deployment. Moreover, fine-grained provenance requires all applications to be instrumented for information flow tracking. Unfortunately, software vendors do not ship their applications with such tracking, and are unlikely to do so in the foreseeable future. Consequently, coarse-grained provenance remains the only practical option for enterprises.

1.1 Thesis Statement

This thesis develops two new ideas to overcome the challenges faced by provenance-based techniques: *versioned provenance graphs* and *provenance tags.* Although new nodes and edges get added to versioned graphs over time, reachability between nodes does not change, and hence, global dependences can be cached and reused. This enables the use of small tags at each node to *compactly* summarize important global dependencies. As a result, global dependence can (efficiently) be used in attack detection rules, obviating the false positives of previous detection techniques that relied just on local information. Moreover, tags enable graph reduction techniques that cut down storage requirements by orders of magnitude, thereby permitting provenance graphs to be stored and analyzed in real-time on main memory. Finally, by varying the rules for the assignment and propagation of tags, dependence explosion can be effectively mitigated and compact scenario graphs generated.

1.2 Summary of Contribution

We make the following contributions in this thesis:

Versioned Graphs and Dependence Compaction

Dependency information of processes and files change over time. Due to the use of timestamped edges in provenance graphs, this dependency information cannot be computed and cached for subsequent use. As a result, real-time forensic analyses become infeasible. To overcome the computational challenges posed by timestamped graphs, we transform them into standard graphs using versions which we discuss in Chapter 3.

Another challenge for performing real-time forensic analysis is the requirement for the entire provenance graph to be kept in the main memory. But this is very challenging due to the massive size of the logs involved. To mitigate this issue we develop a compact in-memory graph representation in Chapter 3. Three important optimizations underpin our compact representation: redundant edge optimization (REO), redundant node optimization (RNO) and cycle-collapsing optimization (CCO). We show that these techniques provably preserve forensic analysis results, and hence don't sacrifice accuracy for performance. Using our representation techniques, we are able to compactly store an entire provenance graph in memory, requiring about 2 bytes per event on average. As a result, a data set with 72M events utilized only 111MB of memory in one of our experiments. We also generate compact event logs using a space-efficient log format that is about **8**× smaller than a Linux audit log containing roughly the same information.

Attack Detection

Another contribution of this thesis is the development of tag-based attack detection policies introduced in Chapter 4 and refined further in subsequent chapters. Our detection techniques focus on the high-level objectives of most attackers. Specifically, we combine reasoning about an attacker's *motives* and *means*. If an activity can help the attacker achieve his/her key high-level objectives, that would provide the motivation and justification for including that activity in an attack. Examples of such activities include deploying and running malware on a victim system, replacing/modifying important password files or cryptographic keys, exfiltrating sensitive data, etc. But many of these activities may be also occur during normal use of the system by legitimate users. To discriminate attacks from such benign background activity, we examine if the attacker has the *means* to control these activities. We associate *integrity tags* with files and processes to indicate whether they are under the influence or control of potential attackers. These tags are derived from global dependency information in the provenance graph. By combining both the motive and the means, our policies tend to produce far fewer false positives than previous techniques that rely just on the activities (i.e., motives). Moreover, our techniques don't require any application-specific customization or training in order to be effective. Obtaining good training data is a major challenge in this domain. Some of the widely available data sets, such as the DARPA Transparent Computing dataset used in our evaluation, do not include much representative training data. We are able to circumvented this challenge using our training-free detection approach.

Tag Propagation Semantics and Dependence Explosion

We explore three different tag semantics representing different trade-offs between simplicity, speed, and effectiveness of these techniques on attack detection and attack scenario reconstruction. Each of these semantics also incorporates techniques on dealing with dependence explosion problem.

- SLEUTH: In Chapter 4 we developed a fully automated attack detection and scenario reconstruction system using tags with multiple level of *integrity* and *confidentiality*. The integrity tag is further split to lower the number of false positives. Specifically, a subject (i.e., a process) is given two integrity tags: one that captures its code integrity and another for its data integrity. This separation significantly improves attack detection. More importantly, it can significantly speed up forensic analysis by focusing on fewer suspicious events, while substantially reducing the size of the reconstructed scenario. We also associate a cost measure with each edge in the provenance graph. In combination with the split integrity tag, this cost-based search deals with dependence explosion problem by pruning away higher-cost paths to arrive at compact scenario graphs summarizing attacker activity.

SLEUTH uses dependence information to track the flow of untrusted and/or sensitive content in the system. This is done by assigning integrity and confidentiality tags to data and processes, and propagating these tags in the direction of information flow. Attack detection triggers a forensic backward analysis of the dependence graph to identify the source of an attack, typically an internet connection. It also triggers a forward analysis in the graph to identify the entities that may have been compromised.

- MORSE: While the tag semantics for SLEUTH is effective for fast moving attacks, for long-running attacks, it can produce graphs with numerous benign nodes. For example, if a subject (i.e., a process) reads from a network source, then all subsequent writes by the subject are treated as (potentially) dependent on the network source. This leads to a dependence explosion, as every output of a process becomes dependent on every earlier input operation. In Chapter 5 we introduce MORSE where the core idea behind the approach is to modulate tag propagation using *subject tags* which is related to SLEUTH's code integrity tag. In particular, the tag propagation rules are lenient on benign subjects, and take advantage of their typical behaviors in order to reduce dependence explosion. At the same time, the tag propagation rules for suspicious subjects that may be under the direct control of attackers are treated conservatively.

  We introduce two key concepts, *tag attenuation* and *tag decay* that mitigate dependence explosion through benign processes. Tag decay captures the intuition that a benign subject, if it is subverted and becomes malicious, will do so soon after consuming suspicious input that contains an exploit. For this reason, we allow the data tags of benign subjects to decay gradually and become benign over time, unless they exhibit suspicious behavior. This feature breaks the dependency between suspicious inputs and outputs of a benign subject after a certain threshold of time.

  Tag attenuation captures the intuition that objects serve as imperfect intermediaries for propagating malicious behavior through benign subjects. In particular, each such propagation requires the intermediary object to contain an exploit that compromises the subject that consumes it. To capture the difficulty of creating a series of such exploits, we attenuate data tags of a benign subject before propagating it to the object that it writes into.

- PROBABILISTIC TAGS: Another way of improving attack detection and generating concise attack graph is instead of working with a discrete set of tags, we can give them a probabilistic interpretation. For instance, we can let the confidentiality tag of a file denote the probability of a file containing information sought by an attacker. In Chapter 6 we show how probabilistic view of confidentiality tags can be used to identify stealthy APTs.

  Our probabilistic view of tags provides many benefits. First, it allows us to

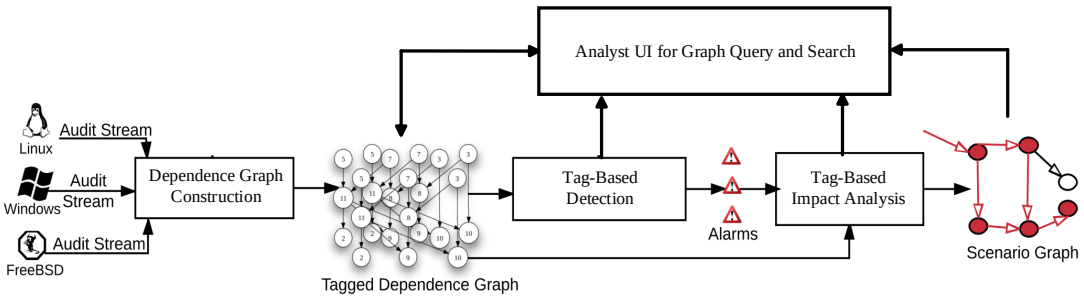

Figure 1.1: System Architecture

capture a continuous range in confidentiality tags, instead of being limited to a few discrete values. Second, it becomes possible to define meaningful propagation rules to handle a variety of situations. For instance, we developed a confidentiality accumulation technique, which, along with entry point detection, is able to accurately identify two key steps included in almost every APT campaign: *reconnaissance* and *data exfiltration*. This system is able to detect recent stealthy attacks that have moved away from file-based malware and are carried out mostly with benign system tools.

System Implementation and Evaluation

As shown in Fig 1.1, our system consists of five components that implement its core functionality, together with an UI for an analyst. The first component consists of data consumers and graph construction. The data consumers process input from COTS auditing systems such as the Linux Auditd, Windows ETW and FreeBSD DTrace systems and generates the provenance graph on the fly. The dependency graph uses different compaction techniques that provably preserve forensic analysis results.

The second component incorporates different tag-based semantics using different tag initialization and propagation techniques. The third and fourth component are tag-based attack detection and forensic analysis respectively. All three of these components are discussed in detail in Chapters 4, 5 and 6.

The fifth layer mainly consists of a user interface for analysts to monitor alarms, run queries on the graph, construct scenario graphs, etc. This web-based UI is capable of performing these functionalities in real-time by communicating with the second, third and fourth component of the system without hampering the graph construction.

The final contribution of this thesis is our experimental evaluation, based mainly on a red team evaluation organized by DARPA as part of its Transparent Computing program. In this evaluation, attack campaigns resembling modern APTs were carried out on Windows, FreeBSD and Linux hosts ranging from days to several weeks period.

Our system is extremely fast and is able to consume these data sets at the rate of about a million events per second. It is also able to achieve on average $7x$ reduction on the number of events. Our compact in-memory versioned graph generates just about 1.3 versions per node and requires 2 bytes on average per event. SLEUTH's split of integrity tags into code vs data tags achieved $7x$ to 500x reduction of alarms and over a $1000x$ reduction on attack graph scenario. Using MORSE we achieve a further $12x$ reduction in alarms and $35x$ reduction in attack graph size using *tag attenuation* and *tag decay*. All of these reductions were achieved without introducing false negatives.

1.3 Organization of the Thesis

In Chapter 2 we discuss background and related research. In Chapter 3 we look at our compaction techniques for storing the massive amount of data in-memory and also on disk. We demonstrate how forensic analyses are speeded up using these techniques, while preserving the fidelity of results.

Chapter 4 discusses the first iteration of our system implementation, called SLEUTH. It develops the concepts of tags and tag propagation and attack detection policies, and formulates the forensic analysis in terms of cost-based graph reachability. A novel contribution in this work is the splitting of integrity tags into code and data integrity. The results were evaluated on the first DARPA Transparent Computing dataset.

Chapter 5 describes the second iteration of our system called MORSE. We introduce two key techniques called *tag attenuation* and *tag decay* in this chapter to mitigate the dependence explosion problem for stealthy attacks. These new techniques were evaluated on DARPA Transparent Computing datasets 3 and 4. Included in these datasets were attacks that relied on preexisting malicious software and kernel modules on the victim systems.

We look at a probabilistic approach on confidentiality tags and how to automatically infer them from training in Chapter 6. In combination with tag accumulation, we demonstrate the effectiveness of these new techniaues in detecting reconnaissance and data exfiltration steps. These techniques were evaluated on DARPA Transparent Computing datasets 3, 4 and 5, as well as other datasets that were created in our research lab.

Lastly, concluding remarks appear in Chapter 7, together with directions for future work.

Chapter 2

Background and Related Work

In this chapter we discuss some of the technical background for this thesis, and survey previous research in related areas.

2.1 Log Collection

Accurate forensic analysis or attack scenario reconstruction requires logging of system activity across the enterprise. Logs should be detailed enough to track dependencies between events occurring on different hosts and at different times, and hence needs to capture all information-flow causing operations such as network/file accesses and program executions. There are three main options for collecting such logs: (1) instrumenting individual applications, (2) instrumenting the operating system (OS), or (3) using network capture techniques.

In the 90s, researchers started investigating sensors that observed network packets. Network intrusion detection systems (NIDS) operated on the basis of such sensors deployed on the network. This contrasted with earlier Host-based IDS (HIDS) that operated on the basis of log files produced by software running on host computers. Commercially, network-based sensors proved far more viable than host-based sensors. Network sensors can be deployed in just a few locations, e.g., network gateways, yet monitor an entire enterprise network. In contrast, host-based sensors need to be deployed on every host. More importantly, their implementation will differ with the specific OS and software deployed on a host. The drawback of network sensors is that they provide very limited insight into software systems that are both the targets as well as agents of intrusions. Trying to diagnose intrusions from just the network sensors is akin to troubleshooting a car from a record of its GPS tracking data, instead of deploying a rich array of sensors within the car. In other words, host-based mechanisms are essential for intrusion detection, and must be realized despite roadblocks to deployment. While early NIDS used only the network packet headers, subsequent (so-called DPI or deep packet inspection) systems have tried to compensate for the lack of host-level information by examining more and more of the

payload data. This offers better visibility into the operation of servers and clients, but still falls far short of what is achievable with host-based sensors. Moreover, the increasing use of end-to-end encryption is rapidly shrinking the share of network traffic that can be monitored this way, once again highlighting the need for host-level sensors.

Moreover, host-level sensors or OS-layer logging can track the activities of *all processes* on a host, including any malware that may be installed by the attackers. In contrast, application-layer logs are limited to a handful of benign applications (e.g.,network servers) that contain the instrumentation for detailed logging.

2.1.1 System Calls

The earliest IDS works relied on OS auditing mechanisms to serve as sensors. Unfortunately, some OSes — in particular, Linux — did not include auditing mechanisms that were available on Sun Solaris. Moreover, even among the OSes that provide auditing, there is no consistency in terms of the security events that are audited, or the format of the audit log.

Against this backdrop, in the mid-90s, researchers began to investigate the use of system-call monitoring for intrusion detection [47, 88, 134] and other security-related applications such as sandboxing. Note that (a) all attacks (with very rare exceptions) target user-level processes, and (b) all security-relevant operations made by processes need to be mediated by the OS and hence must involve system calls. For these reasons, system calls contain all of the information necessary for detecting intrusions. Moreover, since system calls involve a transition into the OS kernel, they can be securely logged without application cooperation. Finally, Linux, which was gradually becoming the the dominant UNIX version, provided several easy ways for logging system calls. For all these reasons, system call logging became the de facto host-based intrusion sensing mechanism in the 00s.

Subsequent research [132] showed that the accuracy of intrusion detection models can be significantly improved by having the sensors record the call stack data[1] in addition to the system calls. While the call stack only records return addresses, further improvements in the models can be achieved by recording calls as well [53]. Instead of incurring performance penalties for runtime recording of calls, a simple static analysis can be used to infer calls from the return addresses. In the extreme case, a static control-flow graph could be used to infer potential calls and returns without any need for additional runtime information [152], but the algorithms for detection become far too expensive. Thus, recording of return addresses is necessary to support efficient and accurate system-call based intrusion detection.

Mimicry attacks [153] and subsequent work in automating them (e.g., [78]) were another factor that favored the development of so-called *gray-box* techniques that

[1]Typically, just the top few return addresses on the stack needed to be recorded, not the entire stack.

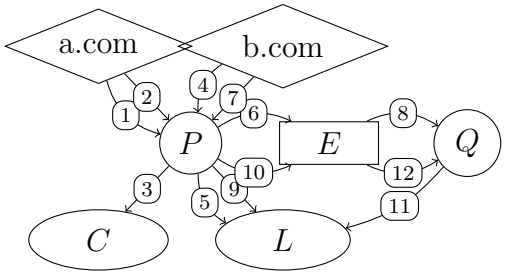

Figure 2.1: An example (time-stamped) provenance graph.

relied on additional runtime information such as the call stacks [46, 48, 49, 53, 132], and properties of sytem call arguments [26, 80, 149]. Under the right conditions, an advanced form of mimicry attack called persistent interposition attack [118] could defeat all previous system-call IDS, as well as those that could realistically put into practice in the future. This demonstrated the need for monitoring more than just system calls. Control-flow integrity research [12] suggested that right primitive would be indirect control-flows. Interestingly, earlier research [85] had suggested monitoring indirect branches[2] in order to reduce false negatives and to improve performance of IDS models.

2.1.2 Provenance Graph

Abstractly, host-based intrusion sensors observe and report system-level *events*, together with *event attributes* that capture a snapshot of the relevant subset of a system's internal state. One such event attributes is provenance information. Using this attribute it is helpful and common to regard the log as defining a graph, called *provenance graph*. System logs refer to two kinds of entities: subjects and objects. Subjects are processes, while objects correspond to passive entities such as files, network connections and so on. Entries in the log correspond to events, which represent actions (typically, system calls) performed by subjects, e.g., read, write, and execute. In most work on forensic analysis [72, 73, 162], the log contents are interpreted as a provenance graph: nodes in the graph correspond to entities, while edges correspond to events. Edges are oriented in the direction of information flow and have timestamps. When multiple instances of an event are aggregated into a single instance, its timestamp becomes the interval between the first and last instances. Fig. 2.1 shows a sample provenance graph, with circles denoting subjects, and the other shapes denoting objects. Among objects, network connections are indicated by a diamond, files by ovals, and pipes by rectangles. Edges are timestamped, but their names omitted. Implicitly, in-edges of subjects denote reads, and out-edges of subjects denote writes.

[2]By "indirect branch," we mean all indirect control-flow transfers, including indirect calls and returns.

Given the huge volume of logs that are collected, provenance based forensic analysis techniques becomes computationally expensive. The following research focused on reducing the log size.

2.1.3 Log Reduction

LogGC [87] proposed an interesting approach for log reduction based on the concept of garbage collection, i.e., removing operations involving removed files ("garbage"). Additional restrictions were imposed to ensure that files of interest in forensic analysis, such as malware downloads, aren't treated as garbage. They report remarkable log reduction with this approach, provided it is used in conjunction with their unit instrumentation. Without such fine-grained instrumentation, the savings they obtain are modest.

ProTracer [98] proposed another new reduction mechanism that was based on logging only the write operations. Read operations, as well as some memory-related operations tracked by their unit instrumentation, were not logged. In the presence of their unit instrumentation, they once again show a dramatic reduction in log sizes using their strategy. However, this strategy of selective logging of writes can actually increase log sizes in the absence of unit instrumentation. Also, as we mentioned earlier, reducing the volume of logs does not necessarily decrease the forensic analysis time significantly.

Xu et al [162] explore a complementary strategy that can remove some (repeated) events on any object. They developed the concept of *trackability equivalence* of events in the audit log, and proved that, among a set of equivalent events, all but one can be removed without affecting forensic analysis results. Across a collection of several tens of Linux and Windows hosts, their technique achieved about a 2× reduction in log size. On the other hand our compact graph and log representation techniques allow significantly more reduction.

Provenance capture systems, starting from PASS [108], incorporate simple reduction techniques such as the removal of duplicate records. PASS also describes the problem of cyclic dependencies and their potential to generate a very large number of versions. They avoid cycles involving multiple processes by merging the nodes for those processes. Our cycle-collapsing optimization is based on a very similar idea.

ProvWalls [22] is targeted at systems that enforce Mandatory Access Control (MAC) policies. It leverages the confinement properties provided by the MAC policy to identify the subset of provenance data that can be safely omitted, leading to significant savings on such systems.

Winnower [147] learns compact automata-based behavioral models for hosts running similar workloads in a cluster. Only the subset of provenance records that deviate from the model need to be reported to a central monitoring node, thereby dramatically reducing the network bandwidth and storage space needed for intrusion detection across the cluster. These models contain sufficient detail for intrusion

detection but not forensics. Therefore, Winnower also stores each host's full provenance graph locally at the host. In contrast, our systems generate compact logs that preserve all the information needed for forensics.

2.1.4 File Versioning

The main challenge for file versioning systems is to control the number of versions, while the challenge for forensic analysis is to avoid false dependencies. Unfortunately, these goals conflict. Existing strategies that avoid false dependencies, e.g., creating a new version of a file on each write [137], generate too many versions. Strategies that significantly reduce the number of versions, e.g., open-close versioning [129],can introduce false dependencies.

Many provenance capture systems use versioning as well. Like versioning file systems, they typically use either simple versioning that creates many versions (e.g., [21, 119]) or coarse-grained versioning that does not accurately preserve dependencies (e.g., [108]).

Provenance capture systems try to avoid cycles in the provenance graph, since cyclic provenance is meaningless. Causality-based versioning [107] discusses two techniques for cycle avoidance. The first of these performs global cycle detection across all objects and subjects on a system. The second operates with a view that is local to an object. It uses a technique similar to our redundant edge optimization, but is aimed at cycle avoidance rather than dependency preservation. They also do not consider redundant node optimization in their approach.

2.1.5 Graph Compression and Summarization

Several techniques have been proposed to compress data provenance graphs by sharing identical substructures and storing only the differences between similar substructures, e.g., [37, 38, 157]. Bao et al. [19] compress provenance trees for relational query results by optimizing the selection of query tree nodes where provenance information is stored. These compression techniques, which preserve every detail of the graph, are orthogonal to our techniques, which can drop or merge edges.

Graph summarization [111, 145] is intended mainly to facilitate understanding of large graphs but can also be regarded as lossy graph compression. However, these techniques are not applicable in our context because they do not preserve dependencies.

2.2 Attack Detection

A number of recent research efforts on attack detection/prevention focus on "inline" techniques that are incorporated into the protected system, e.g., various guarding techniques [34, 39, 40, 45, 125, 127], randomization [15, 20, 25, 27, 28, 29, 69, 71, 89,

101, 126, 158], bounds-checking [13, 14, 43, 58, 67, 82, 110, 112, 120, 161, 163], control-flow integrity [12, 116, 146, 166, 167, 168, 169], taint-based defenses [31, 35, 113, 122, 130, 131, 138, 151, 160] and so on. While most of these defenses are aimed at memory corruption exploits [144], the goals of offline intrusion detection has been broader. This line of research has also been studied for a much longer period [42, 47, 95], particularly on host-based IDS using system-call monitoring [46, 79, 88, 132, 152, 155]. The detection techniques mainly fall into three categories: (i) *misuse detection* [76, 81, 124, 150], which relies on patterns of bad behaviors ("signatures") associated with known attacks; (ii) *anomaly detection* [26, 46, 47, 48, 80, 88, 132, 136], which relies on learning a model of benign behavior and detecting deviations from this behavior; and (iii) *specification-based detection* [32, 74, 75, 133, 148], which relies on specifications of expected behaviors.

Misuse detection is based on specifications of bad behaviors known to be associated with specific attacks. Typically, rules encoding "attack signatures" are used for specification, and pattern-matching algorithms are used in their implementation. Alerts produced by these systems are relatively easy to interpret, as these rules can encode attack-specific information such as an attack name or id. Moreover, the rate of false alerts can be controlled by tuning the signatures. One of their downsides is that they require expert knowledge and effort to write down misuse signatures. A more fundamental drawback is that they can only detect previously known attacks; no signatures will exist for novel attacks that have never been witnessed before. Instead of using expert knowledge Atlas [16] proposes a sequence-based learning approach for generating this signatures.

Anomaly-based intrusion detection is based on the assumption that intrusions result in observable changes in behavior. Anomaly detection techniques construct a model of normal behavior by observing the operation of a system during a training phase. This model can subsequently be used to detect anomalies during the detection phase, i.e., the operational phase of the IDS. During this phase, observed behaviors that deviate from the trained model are anomalies. The strength of anomaly detection is its potential to detect novel attacks. Its drawbacks include: (a) it can be difficult to control false alert rates, since anomalies can occur in the absence of attacks, especially because the training phase is rarely sufficient to capture all legitimate behaviors; and, (b) the alerts from an anomaly detector don't provide specific information that can identify an attack. StreamSpot [99] and Unicorn [57] train on benign data to identify deviations and detect such anomalous behaviors. Specification-based techniques have the potential to detect novel attacks while holding down false positives, but they require application-specific behavior specifications that are time-consuming to develop.

2.2.1 Alarm Clustering

Alarm clustering is just that: define some notion of distance between alarms, and combine the alarms that are very close to each other into a cluster. One obvious dimension for the distance metric is time: we only cluster alarms that are temporally close to each other. More generally, if two alerts X and Y are characterized by a name and a set of attributes, then alerts to be clustered will share many of the same attributes, say, $a_1, ..., a_n$, and for each of these attributes, the values of $X[a_i]$ and $Y[a_i]$ will be "close."

We still need to define what it means for two attribute values to be close. Numeric difference may be a good measure for some attributes. For others, such as an IP address, a hamming distance may be a good measure of distance: two hosts on the same class C network will have a hamming distance less than 8. But in other instances, the notion of distance may be less structured. For instance, a certain small set of IP addresses may be associated with the DMZ of an organization's network, and hence these attribute values would be considered "close" for clustering purposes, even though the IP addresses themselves may be far apart. One way to capture this is to define a generalization hierarchy, and define distance in terms of shortest path in the tree between two nodes [68].

Some times, attributes (or collections of attributes) with different names may be considered close to each other. For instance, different sensors may observe different attributes [79], e.g., a network sensor may observe IP addresses and ports, while at the system-call layer, events are described in terms of userids, process ids, and so on. Some external help/information is needed so as to correlate the two views, or equivalently, to define a notion of distance between attributes with different names.

The main approaches, often used together, are clustering of similar alerts, prioritization, and statistical correlation [41, 68, 79, 115, 117, 121, 128, 154]. Industry tools also use similar techniques in building SIEMs [5, 7, 11] for alarm clustering and enforcement based on disparate logs. RapSheet [59] performs alarm clustering based on the TTPs generated by the EDR systems.

These techniques exploit structural similarities between alerts (e.g., common IP addresses, ports, etc.) and temporal proximity for correlation. In addition, some techniques rely on manually specified prerequisites and consequences of attack steps [114], or models that capture typical progression of attacks [56]. For multi-stage attacks, provenance provides a more principled (and often, far more accurate) basis to correlate and cluster attack steps [72, 165]. For this reason, recent works have come to rely on provenance to correlate attack steps.

2.3 Forensic Analysis and Dependence Explosion

Forensic analysis is concerned with the questions of *what, when* and *how*. The *what* question concerns the *origin* of a suspected attack, and the entities that have been

impacted during an attack. The origin can be identified using *backward analysis,* starting from an entity flagged as suspicious, and tracing backward in the graph. This analysis, first proposed in BackTracker [72], uses event timestamps to focus on paths in provenance graphs that represent causal chains of events. In Figure 2.1 A backward analysis from file C at time 5 will identify P and a.com. Of these, a.com is a source node, i.e., an object with no parent nodes, and hence identified as the likely entry point of any attack on C.

Although b.com is backward reachable from C in the standard graph-theoretic sense, it is excluded because the path from b.com to C does not always go forward in time.

The set of entities impacted by the attack can be found using *forward analysis* [17, 73, 170] (a.k.a. *impact analysis*), typically starting from an entry point identified by backward analysis. In the sample provenance graph, forward analysis from network connection a.com will reach all nodes in the graph, while a forward analysis from b.com will leave out C.

The *when* question asks when each step in the attack occurred. Its answer is based on the timestamps of edges in the subgraph computed by forward and backward analyses. The *how* question is concerned with understanding the steps in an attack in sufficient detail. To enable this, audit logs need to capture all key operations (e.g., important system calls), together with key arguments such as file names, IP addresses and ports, command-line options to processes, etc.

2.3.1 Coarse-grained Tracking

Several logging and provenance tracking systems have been built to monitor the activities of a system [21, 33, 52, 54, 55, 108, 123] and build *provenance graphs.* Among these, *Backtracker* [72, 73] was one of the first works that used dependence graphs to trace back to the root causes of intrusions. These graphs are built by correlating coarse grained events collected by a logging system and by determining the causality among system entities, to help in forensic analysis after an attack is detected. HERCULE [121] uses community discovery techniques to correlate attack steps that may be dispersed across multiple logs. Another key issue with these works on attack investigation [55, 72, 73, 165] and provenance [52, 54, 108, 123] is that they suffer from dependence explosion problem.

HOLMES [105] aims for a much higher level summary of an APT campaign. Individual steps are recognized using a hybrid approach that combines *motive and mean*-style detection policies with signatures based on MITRE's Adversarial Tactics, Techniques and Common Knowledge Base (ATT&CK) [106]. It relies on information flow to link these steps and construct a *high-level scenario graph (HSG)* that maps the attacker's actions to the APT kill-chain [8]. To mitigate dependence explosion, HOLMES discards paths with a *path factor* greater than 3. Path factor is more sophisticated than MORSE's attenuation, but shares the same rationale, i.e., objects

serve as imperfect intermediaries for propagating malicious behavior. At the same time, there is no equivalent of MORSE's decay in HOLMES.

PRIOTRACKER [92] speeds up forward analysis by using a prioritized graph exploration that assigns higher priority to edges representing unusual events. NODOZE [60] improves on it by prioritizing entire paths based on rareness, rather than individual events. Only such rare paths are presented to the analyst, together with the alerts raised on those paths. The main drawback of both approaches is their assumption that processes involved in attacks, including those that may be running attacker's own malware, will exhibit unusual behavior. However, attackers have a great deal of control over their malware, and can alter their behavior to blend in with benign background activity.

2.3.2 Fine-grained Tracking

Fine-grained taint tracking [18, 65, 66, 70, 83, 113, 159, 160] avoids dependence explosion by accurately tracking the source of each output byte to a single input operation (or a few). Although these techniques can be evaded by malware [36], they are very effective in mitigating dependence explosion that typically involves benign applications such as browsers. However, they have a high performance cost, slowing down programs by 2x to 10x or more. BEEP [86], PROTRACER [98] and MPI [97] developed a novel and efficient mechanism called *execution-partitioning,* targeting applications such as servers and web browsers that are prone to dependence explosion. MCI [84] and PROPATROL [103] perform fine-grained taint tracking using model-based inference. Unfortunately, these techniques can require some manual assistance, and moreover, make optimistic assumptions about program behavior that may not hold under attacks. The main drawback of all fine-grained tracking approaches is the need for extensive instrumentation of applications. Since vendors don't ship their application with such instrumentation, fine-grained taint tracking is not an option for enterprises.

2.4 Information Flow Control (IFC)

IFC techniques assign security labels and propagate them in a manner similar to our tags. Early works, such as Bell-LaPadula [24] and Biba [30], relied on strict policies. These strict policies impact usability and hence have not found favor among contemporary OSes. Although IFC is available in SELinux [94], it is not often used, as users prefer its access control framework based on domain-and-type enforcement. While most above works centralize IFC, *decentralized IFC* (DIFC) techniques [44, 77, 164] emphasize the ability of principals to define and create new labels. This flexibility comes with the cost of nontrivial changes to application and/or OS code.

Although our tags are conceptually similar to those in IFC systems, the central research challenges faced in these systems are very different from SLEUTH. In particular, the focus of IFC systems is enforcement and prevention. A challenge for

IFC enforcement is that their policies tend to break applications. Thus, most recent efforts [90, 91, 100, 139, 140, 141, 142, 143] in this regard focus on refinement and relaxation of policies so that compatibility can be preserved without weakening security. In contrast, neither enforcement nor compatibility pose challenges in our setting. On the other hand, IFC systems do not need to address the question of what happens when policies are violated. Yet, this is the central challenge we face: how to distinguish attacks from the vast number of normal activities on the system; and more importantly, once attacks do take place, how to tease apart attack actions from the vast amounts of audit data.

2.5 Threat Hunting

The techniques described above are geared at automating forensic analysis of APT campaigns without requiring prior knowledge about them. It is to be expected that fully automated approaches may fail at times, so organizations have to rely on human experts as their second line of defense. These experts need to "hunt down" attacks, based on their past experience, reports on recent vulnerabilities and exploits, the configuration of the victim's network, and most importantly, the alerts emitted by detectors deployed in the organization. Researchers have begun to build tools and frameworks to assist such *threat hunting* efforts. Gao et al. [50, 51] present query languages for threat hunters, and a system for processing their queries. Shu et al. [135] model threat hunting as a graph computation problem, and present a domain-specific language that simplifies the development of custom graph searches.

Instead of a manual approach, POIROT [104] aims to automate searches for attacks that have been seen before, e.g., in threat intelligence reports. These known attacks are described using query graphs. They develop efficient approximate graph matching algorithms to match query graphs against the data from audit logs.

Chapter 3

Techniques for Space-Efficient Representation of Provenance Graphs

In this chapter we discuss some of the techniques we developed in [63] on how to solve the issue of dependency changing over time in provenance graphs. We also present space efficient in-memory representation techniques so that our systems can store the entire provenance graph in memory for the entire duration of an APT attack. This allows for faster forensic analysis.

3.1 Versioned Graph

Provenance of processes or files rely on global properties of graph reachability. Such global properties are expensive to compute, taking time that can be linear in the size of the (very large) provenance graph. Moreover, due to the use of timestamped edges, provenance changes over time and hence must be computed many times. This mutability also means that results cannot be computed once and cached for subsequent use, unlike standard graphs, where we can determine once that v is a descendant of u and reuse this result in the future.

To overcome these computational challenges posed by timestamped graph, we show how to transform them into standard graphs. The basic idea is to construct a graph in which objects as well as subjects are *versioned.* Versioning is widely used in many domains, including software configuration management, concurrency control, file systems [109, 129] and provenance [21, 107, 108, 119]. In these domains, versioning systems typically intervene to create file versions, with the goal of increased recoverability or reproducibility. In contrast, we operate in a forensic setting, where we can only observe the order in which objects (as well as subjects) were accessed. Provenance capture systems may additionally intervene to break cyclic dependencies [107, 108], since cyclic provenance is generally considered meaningless.

Our goal is to (a) make sound inferences about dependencies through these observations, and (b) encode these dependencies in a standard (rather than time-stamped) graph. This encoding serves as the basis for developing efficient algorithms for log reduction. Specifically, we make the following contributions:

- *Efficient reduction algorithms.* By working with versioned graphs, we achieve algorithms that typically take constant time per event. In our experiments, we were able to process close to a million events per second on a single-core on a typical laptop computer. In contrast, on timestamped graphs that would be unacceptably slow[1].

- *Minimizing the number of versions.* We present several optimization techniques in Section 3.1.4 to reduce the number of versions. Whereas naive version generation leads to an explosion in the number of versions, our optimizations are very effective, bringing down the average number of versions per object and subject to about 1.3.

- *Avoiding spurious dependencies.* While it is important to reduce the space overhead of versions, this should not come at the cost of inaccurate forensic analysis. We therefore establish formally that results of forensic analysis (specifically, forward and backward analyses) are fully preserved by our reduction.

- *Optimality.* We show that edges and versions retained by our reduction algorithm cannot be removed without introducing spurious dependencies.

An interesting aspect of our work is that we use versioning to reduce storage and runtime, whereas versioning is normally viewed as a performance cost to be paid for better recoverability or reproducibility. Using versioning, we realize algorithms that are both faster and use less storage than their unversioned counterparts. Specifically, we realize substantial reduction in the size of the provenance graph by relying on versioning.

3.1.1 Dependence Preserving Reductions

We define a *reduction* of a time-stamped dependence graph G to be another graph G' that contains the same nodes but a subset of the events. Such a reduction may remove "redundant" events, and/or combine similar events. As a result, some events in G may be dropped in G', while others may be aggregated into a single event. When events are combined, their timestamps are coalesced into a range that (minimally) covers all of them.
A log reduction needs to satisfy the following conditions:

[1]In order to determine if an edge e is redundant, we would potentially have to consider every path in the graph containing e; the number of such paths can be exponential in the size of the graph.

- it won't change forensic analysis results, and
- it won't affect our understanding of the results.

To satisfy the second requirement, we apply reductions only to *read, write*[2], and *load* events. All other events, e.g., *fork*, *execve*, *remove*, *rename* and *chmod,* are preserved. Despite being limited to reads, writes and loads, our reduction techniques are very effective in practice, as these events typically constitute over 95% of total events.

For the first requirement, our aim is to preserve the results of forward and backward forensic analysis. We ensure this by preserving forward and backward reachability across the original graph G and the reduced graph G'. We begin by formally defining reachability in these graphs.

3.1.2 Reachability in time-stamped dependence graphs

Dependence graph G is a pair (V, E) where V denotes the nodes in the graph and E denotes a set of directed edges. Each edge e is associated with a start time $start(e)$ and an end time $end(e)$. Reachability in this graph is defined as follows:

Definition 1 (Causal Path and Reachability)**.** *A node v is reachable from another node u if and only if there is (directed) path $e_1, e_2, \ldots, e_n$ from u to v such that:*

$$\forall 1 \leq i < n \quad start(e_i) \leq end(e_{i+1}) \tag{3.1}$$

We refer to a path satisfying this condition as a *causal path.* It captures the intuition that information arriving at a node through event e_i can possibly flow out through the event e_{i+1}, i.e., successive events on this path $e_1, e_2, \ldots, e_n$ *can* be causally related. In Fig. 2.1, the path consisting of edges with timestamps $1, 6, 8$ and 11 is causal, so L is reachable from a.com. In contrast, the path corresponding to the timestamp sequence $4, 3$ is not causal because the first edge occurs later than the second. Hence C is unreachable from b.com.

In forensics, we are interested in reachability of a node at a given time, so we extend the above definition as follows:

Definition 2 (Forward/Backward Reachability at t)**.**

- *A node v is forward reachable from a node u at time t, denoted $u@t \longrightarrow v$, iff there is a causal path $e_1, e_2, \ldots, e_n$ from u to v such that $t \leq end(e_i)$ for all i.*
- *A node u is said to be backward reachable from v at time t, denoted $u \longrightarrow v@t$, iff there is a causal path $e_1, e_2, \ldots, e_n$ from u to v such that $t \geq start(e_i)$ for all i.*

[2]There can be many types of read or write events, some used on files, others used on network sockets, and so on. For example, Linux audit system can log over a dozen distinct system calls used for input or output of data. For the purposes of this description, we map them all into reads and writes.

Intuitively, $u@t \longrightarrow v$ means u's state at time t can impact v. Similarly, $u \longrightarrow v@t$ means v's state at t can be caused/explained by u. In Fig. 2.1, $P@6 \longrightarrow Q$, but $P@11 \not\longrightarrow Q$. Similarly, a.com $\longrightarrow C@3$ but b.com $\not\longrightarrow C@3$.

3.1.3 Naive Versioned Dependence Graphs

The simplest approach for versioning is to create a new version of a node whenever it gets a new incoming edge, similar to creating a new file version each time the file is written. Fig. 3.1 shows an example of an unversioned graph and its corresponding naive versioned graph. Versions of a node are stacked vertically in the example so as to make it easier to see the correspondence between nodes in the timestamped and versioned graphs.

Note that timestamps in versioned graphs are associated with nodes (versions), not with edges. A version's start time is the start time of the event that caused its creation. We show this time using a superscript on the node label.

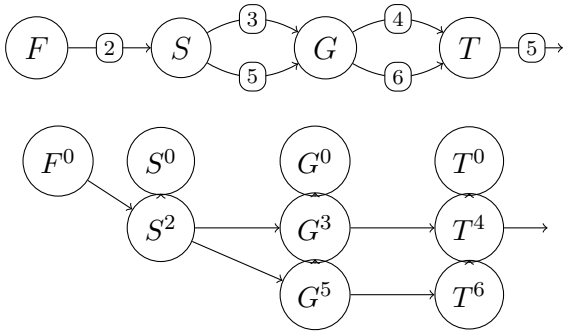

Figure 3.1: A timestamped graph and equivalent naive versioned graph.

Algorithm for naive versioned graph construction

We treat the contents of the audit log as a timestamped graph $G = (V, E_T)$. The subscript T on E is a reminder that the edges are timestamped. The corresponding (naive) versioned graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is constructed using the algorithm shown below. Without loss of generality, we assume that every edge in the audit log has a unique timestamp and/or sequence number. We denote a directed edge from u to v with timestamp t as a triple (u, v, t). Let $u^{<t}$ denote the latest version of u in the versioned graph before t.

We intend $BuildVer$ and its optimized versions to be *online algorithms,* i.e., they need to examine edges one-at-a-time, and decide immediately whether to create a new version, or to add a new edge.

```
1. BuildVer(V, E_T)
2.    V = {v^0 | v ∈ V}; E = {};
3.    for each (u, v, t) ∈ E_T
4.        add v^t to V
5.        add (u^<t, v^t) to E
6.        add (v^<t, v^t) to E
7.    return (V, E)
```

For each entity v, an initial version v^0 is added to the graph at line 2.[3] The for-loop processes log entries (edges) in the order of increasing timestamps. For an edge (u, v) with timestamp t, a new version v^t of the target node v is added to the graph at line 4. Then an edge is created from the latest version of u to this new node (line 5), and another edge created to link the last version of v to this new version (line 6).

Forensic analysis on versioned graphs

In a naive versioned graph, each object and subject gets split into many versions, with each version corresponding to the time period between two consecutive incoming edges to that entity in the unversioned graph. To flag an entity v as suspicious at time t, the analyst marks the latest version $v^{\leq t}$ of v at or before t as suspicious. Then the analyst can use standard graph reachability in the versioned graph to perform backward and forward analysis. For the theorem and proof, we use the notation $v^{<\infty}$ to refer to the latest version of v so far. In addition, we make the following observation that readily follows from the description of $BuildVer$.

Observation 3. *For any two node versions u^t and u^s, there is a path from u^t to u^s if and only if $s \geq t$.*

Theorem 4. *Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be the versioned graph constructed from $G = (V, E_T)$. For all nodes u, v and times t:*

- *v is forward reachable from $u@t$ iff there is a simple path in $\mathbf{G}$ from $u^{\leq t}$ to $v^{<\infty}$; and*
- *u is backward reachable from $v@t$ iff there is a path in $\mathbf{G}$ from u^0 to $v^{\leq t}$.*

Proof: For uniformity of notation in the proof, let $t = t_0, u = w_0$ and $v = w_n$. The definition of reachability in timestamped graphs (specifically, Definitions 1 and 2), when limited to instantaneous events, states that $w_0@t \longrightarrow w_n$ holds in G if and only if there is a path

$$(w_0, w_1, t_1), (w_1, w_2, t_2), \ldots, (w_{n-1}, w_n, t_n)$$

[3]This is a logical simplification — in reality, initial version of v will be added to the graph at the first occurrence of v in the audit stream.

in G such that $t_{i-1} \leq t_i$ for $1 \leq i \leq n$. For each timestamped edge (w_{i-1}, w_i, t_i), *BuildVer* adds a (standard) edge $(w_{i-1}^{<t_i}, w_i^{t_i})$ to $\mathbf{G}$. In addition, by Observation 3, there is a path from $w_i^{t_i}$ to $w_i^{<t_{i+1}}$. Putting these edges and paths together, we can construct a path in $\mathbf{G}$ from $w_0^{<t_0}$ to $w_n^{t_n}$. Also, by Observation 3, there is a path from $w_n^{t_n}$ to $w_n^{<\infty}$. Putting all these pieces together, we have a path from $w_0^{<t_0} = u^{<t_0}$ to $w_n^{<\infty} = v^{<\infty}$. A path from $u^{<t_0}$ to $v^{<\infty}$ clearly implies a path from $u^{\leq t_0}$ to $v^{<\infty}$, thus satisfying the "only if" part of the forward reachability condition.

Note that the "only if" proof constructed a one-to-one correspondence between the paths in G and $\mathbf{G}$. This correspondence can be used to establish the "if" part of the forward reachability condition as well.

The proof of the backward reachability condition follows the same steps as the proof of forward reachability, so we omit the details. ∎

3.1.4 Optimized Versioning

Naive versioning is simple but offers no benefits in terms of data reduction. In fact, it increases storage requirements. In this section, we introduce several optimizations that reduce the number of versions and edges. These optimizations cause node timestamps to expand to an interval. A node v with timestamp interval $[t, s]$ will be denoted $v^{t,s}$.

Redundant edge optimization (REO)

Before adding a new edge between u and v, we check if there is already an edge from the latest version of u to some version of v. In this case, the new edge is redundant: in particular, reachability is unaffected by the addition of the edge, so we discard the edge. This also means that no new version of v is generated. Specifically, consider the addition of an event (u, v, t) to the graph. Let $u^{r,s}$ be the latest version of u. We check if there is already an edge from $u^{r,s}$ to an existing version of v. If so, we simply discard this event. We leave the node timestamp unchanged. Thus, for a node $u^{r,s} \in \mathbf{G}$, r represents the timestamp of the first edge coming into this node, while s represents the timestamp of the last. Alternatively, r denotes the start time of this version, while s denotes the last time it acquired a new incoming edge (i.e., an edge that wasn't eliminated by a reduction operation). Fig. 3.2 illustrates redundant edge (REO) optimization.

Global redundant edge optimization (REO*)

With REO, we check whether there is already a *direct edge* from u to v before deciding to add a new edge. With REO*, we generalize to check whether u is an *ancestor* of v. Specifically, before adding an event (u, v, t) to the graph, we check whether the

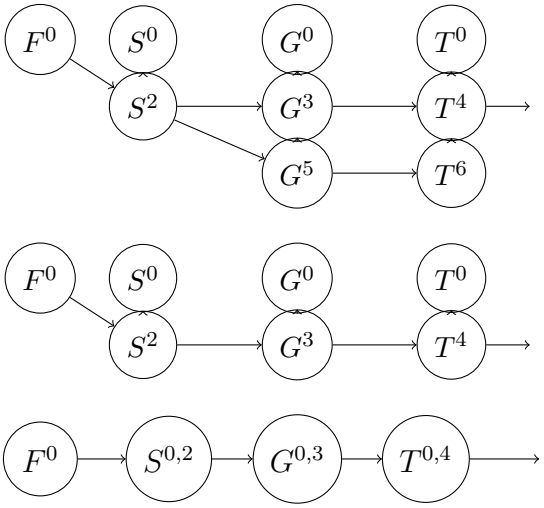

Figure 3.2: The naive versioned graph from Fig. 3.1 (top), and the result of applying redundant edge optimization (REO) (middle) and then redundant node optimization (RNO) (bottom) to it. When adding the edge $(S, G, 5)$, we find that there is already an edge from the latest version S^2 of S to G, so we skip this edge. For the same reason, the edge $(G, T, 6)$ can be skipped, and this results in the graph shown in the middle. For the bottom graph, note that when adding the edge $(F, S, 2)$, S has no descendants, so we simply update S^0 by $S^{0,2}$, and avoid the generation of a new version. For the same reason, we can update G^0 and T^0 as well, resulting in the graph at the bottom.

latest version of u is already an ancestor of the latest version of v. If so, we simply discard the event.

The condition in REO* optimization is more expensive to check: it may take time linear in the size of the graph. Also, it did not lead to any significant improvement over REO in our experiments, so we did not evaluate it in detail. However, it is of conceptual significance because the resulting graph is optimal with respect to FD, i.e., any further reduction would violate FD-preservation.

Redundant node optimization (RNO)

The goal of this optimization is to avoid generating additional versions if they aren't necessary for preserving dependence. We create a new version v^s of a vertex because, in general, the descendants of v^s could be different from those of v^l, the latest version of v so far. If we overzealously combine v^l and v^s, then a false dependency will be introduced, e.g., a descendant of v^l may backtrack to a node that is an ancestor of v^s but not v^l. This possibility exists as long as (a) the ancestors of v^l and v^s aren't identical, and (b) v^l has non-zero number of descendants. We already considered (a) in designing REO optimizations described above, so we consider (b) here. Note that RNO needs to be checked only on edges that aren't eliminated by REO(or REO*).

Specifically, let $v^{r,s}$ be the latest version of v so far. Before creating a new version of v due to an event at time t, we check whether $v^{r,s}$ has any outgoing edge (i.e., any descendants). If not, we replace $v^{r,s}$ with $v^{r,t}$, instead of creating a new version of ve Fig. 3.2 illustrates the result of applying this optimization.

RNO preserves dependence for descendants of v, but it can change backward reachability of the node v itself. For instance, consider the addition of an edge at time t from $u^{p,q}$ to $v^{r,s}$. This edge is being added because it is not redundant, i.e., a backward search from $v@s$ does not reach $u^{p,q}$. However, when we add the new edge and update the timestamp to $v^{r,t}$, there is now a backward path from $v@s$ to $u^{p,q}$. The simplest solution is to retain the edge timestamp on edges added with RNO, and use them to prune out false dependencies.[4]

Cycle-Collapsing Optimization (CCO)

Occasionally, cyclic dependencies are observed, e.g., a process that writes to and reads from the same file, or two processes that have bidirectional communication. As observed by previous researchers [107, 108], such dependencies can lead to an explosion in the number of versions. The typical approach is to detect cycles, and treat the nodes involved as an equivalence class. A simple way to implement this approach is as follows. Before adding an edge from a version u^r to v^s, we check if

[4]Note that these timestamps need to be used only when an edge added with RNO is the first hop in a backward traversal. If a node v subject to RNO gets a child x, this child would have been added after the end timestamp of v. So, when we do a backward traversal from x, all parents of v should in fact be backward reachable.

there is a cycle involving u and v. If so, we simply discard the edge. Our experimental results show that cycle detection has a dramatic effect on some data sets.

Cycle detection can take time linear in the size of the graph. Since the dependence graph is very large, it is expensive to run full cycle detection before the addition of each edge. Instead, our implementation only checks for cycles involving two entities. We found that this was enough to address most sources of version explosion. An alternative would be to search for larger cycles when a spurt in version creation is observed.

Effectiveness of the optimizations

REO and RNO optimizations avoid new versions in most common scenarios that lead to an explosion of versions with naive versioning:

- *Output files:* Typically, these files are written by a single subject, and not read until the writes are completed. Since all the write operations are performed by one subject, REO avoids creating multiple versions. In addition, all the write operations are combined.
- *Log files:* Typically, log files are written by multiple subjects, but are rarely read, and hence by RNO, no new versions need to be created.
- *Pipes:* Pipes are typically written by one subject and read by another. Since the set of writers does not change, a single version is sufficient, as a result of REO. Moreover, all the writes on the pipe can be combined into one operation, and so can all the reads.

We found that most savings were obtained by REO, RNO, and CCO. As mentioned above, REO* is more significantly more expensive than REO and provided little additional benefit. Another undesirable aspect of REO* (as well as the SD optimization) is that it may change the paths generated during a backward or forward analysis. Such changes have the potential to make attack interpretation more difficult. In contrast, REO, RNO and CCO preserve all cycle-free paths.

3.1.5 Dependency-Preserving Reductions

We have introduced some optimization in the previous section but what do these optimizations actually achieve? Also, how optimal are these techniques? To answer these questions we first present three dependency preserving reduction techniques based on reachability according to Definitions 1 and 2.

Continuous dependence (CD) preservation

This reduction aims to preserve forward and backward reachability at every instant of time.

Definition 5 (Continuous Dependence Preservation)**.** *Let G be a dependence graph and G' be a reduction of G. G' is said to preserve continuous dependence iff forward and backward reachability is identical in both graphs for every pair of nodes at all times.*

In Fig. 3.3, S reads from a file F at $t = 2$ and $t = 4$, and writes to another file F' at $t = 3$ and $t = 6$. Based on the above definition, continuous dependence is preserved when the reads by S are combined, as are the writes, as shown in the lower graph.

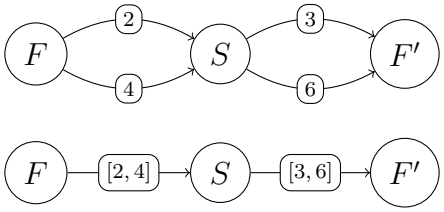

Figure 3.3: Reduction that preserves continuous dependence.

Fig. 3.4 shows a reduction that does *not* preserve continuous dependence. In the original graph, $F@3 \not\rightarrow H$: the earliest time $F@3$ can affect S is at $t = 4$, and this effect can propagate to $F'@6$, but by this time, the event from F' to H has already terminated. In contrast, in the reduced graph, $F@3$ affects $H@5$.

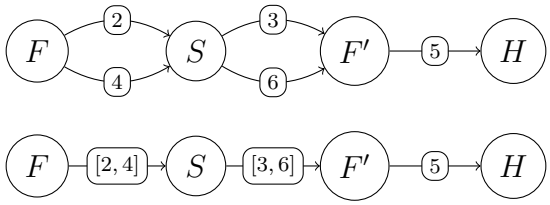

Figure 3.4: Reduction that violates continuous dependence.

Full Dependence (FD) Preservation

CD does not permit the reduction in Fig. 3.4, because it changes whether the state of F at $t = 3$ propagates to H. But does this difference really matter in the context of forensic analysis? To answer this question, note that there is no way for F to become compromised at $t = 3$ if it was not already compromised before. Indeed, there is no basis for the state of F to change between $t = 0$ and $t = 6$ because nothing happens to F during this period.

More generally, subjects and objects don't spontaneously become compromised. Instead, compromises happen due to input consumption from a compromised entity,

such as a network connection, compromised file, or user[5]. This observation implies that keeping track of dependencies between entities at times strictly in between events is unnecessary, because nothing relevant changes at those times. Therefore, we focus on preserving dependencies at times when a node could become compromised, namely, when it acquires a new dependency.

Formally, let $Anc(v,t)$ denote the set of ancestor nodes of v at time t, i.e., they are backward reachable from v at t.

$$Anc(v,t) = \{u \mid u \longrightarrow v@t\}.$$

Let $NewAnc(v)$ be the set of times when this set changes, i.e.:

$$NewAnc(v) = \{t \mid \forall t' < t,\ Anc(v,t) \supset Anc(v,t')\}.$$

We define $NewAnc(v)$ to always include $t = 0$.

Definition 6 (Full Dependence (FD) Preservation)**.** *A reduction G' of G is said to preserve full dependence iff for every pair of nodes u and v:*

- *forward reachability from $u@t$ to v is preserved for all $t \in NewAnc(u)$, and*
- *backward reachability of u from $v@t$ is preserved at all t.*

In other words, when FD-preserving reductions are applied:

- the result of backward forensic analysis from any node v will identify the exact same set of nodes before and after the reduction.
- the result of forward analysis carried out from any node u will yield the exact same set of nodes, as long as the analysis is carried out at any of the times when there is a basis for u to get compromised.

To illustrate the definition, observe that FD preservation allows the reduction in Fig. 3.4, since
(a) backward reachability is unchanged for every node, and
(b) $NewAnc(F) = \{0\}$, and $F@0$ flows into S, F' and H in the original as well as the reduced graphs.

[5]We aren't suggesting that a compromised process must *immediately* exhibit suspicious behavior. However, in order to fully investigate the extent of an attack, forensic analysis needs to focus on the earliest time a node could have been compromised, rather than the time when suspicious behavior is spotted. Otherwise, the analysis may miss effects that may have gone unnoticed between the time of compromise and the time suspicious behavior was observed.

Source Dependence (SD) Preservation

We consider further relaxation of dependence preservation criteria in order to support more aggressive reduction, based on the following observation about the typical way forensic analysis is applied. An analyst typically flags an entity as being suspicious, then performs a backward analysis to identify likely root causes. Root causes are *source* nodes in the graph, i.e., nodes without incoming edges. Source nodes represent network connections, preexisting files, processes started before the audit subsystem, plugable media devices, and user (e.g., terminal) input. Then, the analyst performs an impact (i.e., forward) analysis from these source nodes. To carry out this task accurately, we need to preserve only information flows from source nodes; preserving dependencies between all pairs of internal nodes is unnecessary.

Definition 7 (Source Dependence (SD) Preservation)**.** *A reduction G' of G is said to preserve source dependence iff for every node v and a* ***source node*** *u:*

- *forward reachability from $u@0$ to v is preserved, and*
- *backward reachability of u from $v@t$ is preserved at all t.*

Note that SD coincides with FD applied to source nodes. The second conditions are, in fact, identical. The first conditions coincide as well, when we take into account that $NewAnc(u) = \{0\}$ for any source node u. (A source node does not have any ancestors, but since we have defined $NewAnc$ to always include zero, $NewAnc$ of source nodes is always $\{0\}$.)

Fig. 3.5 shows a reduction that preserves SD but not FD. In the figure, F and F' are two distinct files, while S, S' and S'' denote three distinct processes. Note that FD isn't preserved because a new flow arrives at S' at $t = 2$, and this flow can reach F' in the original graph but not in the reduced graph. However, SD is preserved because the reachability of S, S', S'' and F' from the source node F is unchanged.

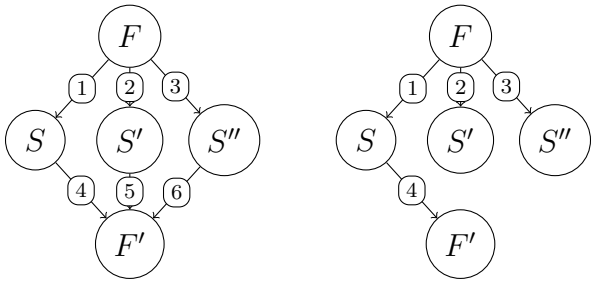

Figure 3.5: Source dependence preserving reduction.

Efficient Computation of Reductions

Using the optimizations on versioned graph discussed earlier we can efficiently perform reduction for FD and SD preservation. The naive versioning technique creates a new version of node with every new incoming edge. But based on the definition of FD an SD these is redundant. If a node v has no descendant there is no need for creating a new version of the node.

Naive versioning also retains every single edge in the dependency graph which is also redundant for FD and SD preservation. REO(or REO*) only adds an edge between two nodes u and v if u is not an ancestor of v for FD preservation. In case of SD the edge between node u and v is added only if it changes the sources that are backward reachable from v.

In our paper [63], we formally prove that BuildVer, together with REO, RNO, and CCO optimizations, preserves both full dependence (FD) and source dependence (SD).

3.2 Compact Representation of Reduced Logs

Logs in their original format e.g., Linux audit records aren't space-efficient, so we developed a simple yet compact format called CSR. CSR stands for Common Semantic Representation, signifying that a unified format is used for representing audit data from multiple OSes, such as Linux and Windows. Translators can easily be developed to translate CSR to standard log formats, so that standard log analyzers, or simple tools such as grep, can be used.

In CSR, all subjects and objects are referenced using a numeric index. Complex data values that get used repeatedly, such as file names, are also turned into indices. A CSR file begins with a table that maps strings to indices. Following this table is a sequence of operations, each of which correspond to the definition of an object (e.g., a file, network connection, etc.) or a forensic-relevant operation such as open, read, write, chmod, fork, execve, etc.

Each operation record consist of abbreviated operation name, arguments (mostly numeric indices or integers), and a timestamp. All this data is represented in ASCII format for simplicity. Standard file compression can be applied on top of this format to obtain further significant size reduction, but this is orthogonal to our work.

3.3 Compact Main Memory Representation

A commonly suggested approach for forensic analysis is to store the provenance graph in a graph database. The database's query capabilities can then be used to perform backward or forward searches, or any other custom forensic analysis. Graph databases such as OrientDB, Neo4j and Titan are designed to provide efficient support for graph

queries, but experience suggests that their performance degrades dramatically on graphs that are large relative to main memory. For instance, a performance evaluation study on graph databases [102] found that they are unable to complete simple tasks, such as finding shortest paths on graphs with 128M edges, even when running on a computer with 256GB main memory and sufficient disk storage. Log reduction techniques can help, but may not be sufficient on their own. Over the span of an APT (many months or a year), graph sizes can approach a billion edges *even after log reduction.*

Forensic analysis requires queries over the provenance graph, e.g., finding shortest path(s) to the entry node of an attack, or a depth-first search to identify impacted nodes. The graph contains roughly the same information that might be found in Linux audit logs. In particular, the graph captures information pertaining to most significant system calls. Key argument values are stored (e.g., command lines for execve, file names, and permissions), while the rest are ignored (e.g., the contents of buffers in read and write operations).

Nodes in the provenance graph correspond to subjects and objects. Nodes are connected by *bidirectional* edges corresponding to events (typically, system calls). To obtain a compact representation, subjects, objects, and most importantly edges must be compactly encoded. Edges typically outnumber nodes by one to two orders of magnitude, so compactness of edges is paramount.

The starting point for our compact memory representation is to use compact identifiers for referencing nodes and node attributes. Also using versioned graphs and the optimizations described in Section 3.1 to achieve compactness helps improve performance.

Specifically, the main techniques we rely on to reduce memory use are:

- *Edge reductions:* The biggest source of compaction is the redundant edge optimization. Savings are also achieved because we don't need timestamps on most edges. Instead, timestamps are moved to nodes (subject or object versions). This enables most stored edges to use just 6 bytes in our implementation, encoding an event name and about a 40-bit subject or object identifier.

- *Node reductions:* The second biggest source of compaction is node reduction, achieved using RNO and CCO optimizations. In addition, our design divides nodes into two types: base versions and subsequent versions. Base versions include attributes such as name, owner, command line, etc. New base versions are created only when these attributes change. Attribute values such as names and command lines tend to be reused across many nodes, so we encode them using compact ids. This enables a base version to be stored in 32 bytes or less.

- *Compact representation for versions:* Subsequent versions derived from base versions don't store node attributes, but just the starting and ending timestamps. By using relative timestamps and sticking to a 10ms timestamp granu-

larity[6], we are able to represent a timestamp using 16-bits in most cases. This enables a version to fit within the same size as an edge, and hence it can be stored within the edge list of a base version. In particular, let S be the set of edges occurring between a version v and the next version appearing in the edge list. Then S is the set of edges incident on version v in the graph.

Edge lists are maintained as vectors that can grow dynamically for active nodes (i.e., running processes and open files) but are frozen at their current size for inactive nodes. This technique, together with the technique of storing versions within the edge list, reduces fragmentation significantly. As a result, we achieve a very compact representation that often takes just a few bytes per edge in the original data.

3.4 Evaluation

3.4.1 Data Sets

Our evaluation uses data from live servers in a small laboratory, and from a red team evaluation led by a government agency. We describe these data sets below.

Data from Red Team Engagement

This data was collected as part of the 2$^{\text{nd}}$ adversarial engagement organized in the DARPA Transparent Computing program. Several teams were responsible for instrumenting OSes and collecting data, while our team (and others) performed attack detection and forensic analysis using this data. The red team carried out attack campaigns that extended over a period of about a week. The red team also generated benign background activity, such as web browsing, emailing, and editing files.

Linux Engagement Data (Linux Desktop). Linux data (Linux Desktop) captures activity on an Ubuntu desktop machine over two weeks. The principal data source was the built-in Linux auditing framework. The audit data was transformed into a OS-neutral format by another team and then given to us for analysis. The data includes all system calls considered important for forensic analysis, including open, close, clone, execve, read, write, chmod, rm, rename, and so on. Table 3.1 shows the total number of events in the data, along with a breakdown of important event types. Since reads and writes provide finer granularity information about dependencies than open/close, we omitted open/close from our analysis and do not include them in our figures.

[6]This is the granularity typically available on most of our data sets.

Dataset	Total Events	Read	Write	Clone/ Exec	Other
Linux Desktop	72.6M	72.4%	26.2%	0.5%	0.9%
Windows Desktop	14.6M	77.1%	14.5%	1.2%	7.2%
SSH/File Server	14.4M	38.2%	58.3%	1.2%	2.3%
Web Server	2.8M	64.3%	30.3%	1.5%	3.9%
Mail Server	3M	70%	23.6%	1.7%	4.7%

Table 3.1: Data sets used in evaluation.

Windows Engagement Data (Windows Desktop). Windows data covers a period of about 8 days. The primary source of this data is Event Tracing for Windows (ETW). Events captured in this data set are similar to those captured on Linux. The data was provided to us in the same OS-neutral format as the Linux data. Nevertheless, some differences remained. For examples, network reads and network writes were omitted (but network connects and accepts were reported). Also reported were a few Windows-specific events, such as CreateRemoteThread. Registry events were mapped into file operations. From Table 3.1, it can be seen that the system call distribution is similar as for Linux, except for a much higher volume of "other" calls, due to higher numbers of renames and removes.

Data From Laboratory Servers

An important benefit of the red team data is that it was collected by teams with expertise in instrumenting and collecting data for forensic analysis. A downside is that some details of their audit system configurations are unknown to us. To compensate for this, we supplemented the engagement data sets with audit logs collected in our research lab. Audit data was collected on a production web server, mail server, and general purpose file and remote access server (SSH/File Server) used by a dozen users in a small academic research laboratory. All of these systems were running Ubuntu Linux. Audit data was collected over a period of one week using the Linux audit system, configured to record open, close, read, write, rename, link, unlink, chmod, etc.

3.4.2 Log Size Reduction

Table 3.2 shows the effectiveness of our techniques in reducing the on-disk size of log data. The second column shows the size of the original data, i.e., Linux audit data for laboratory servers, and OS-neutral intermediate format for red team engagement data. The third column shows the reduction in size achieved by our CSR representation[7], before any reductions are applied.

[7]Recall that CSR is uncompressed, so there is room for significant additional reduction in size, if the purpose is archival storage.

Dataset	Size on Disk	CSR
Linux Desktop	12.9GB	5.6×
Windows Desktop	2.1GB	2.4×
SSH/File server	6.7GB	15.1×
Web server	1.3GB	13.3×
Mail server	1.2GB	11.9×
Average (Geometric mean)		**8**×

Table 3.2: Log size on disk. The second column reports the log size of original audit data. The third column reports the factor of decrease in CSR log size on disk.

3.4.3 Dependence Graph Size

Table 3.3 illustrates the effect of different optimizations on memory use. On the largest dataset (Linux desktop), our memory use with is remarkably low: less than two bytes per event in the original data. On the other two larger data sets (Windows desktop and SSH/file server), it increases to 3.3 to 6.8 bytes per event. The arithmetic and geometric means (across all the data sets) are both less than 5 bytes/event.

Each event results in a forward and backward edge, each taking 6 bytes in our implementation (cf. Section 3.3). Subtracting this 4.7M*12B = 56.4MB from the 111MB, we see that the 1.1M nodes occupy about 55MB, or about 50 bytes per node. Recall that each node takes 32 bytes in our implementation, plus some additional space for storing file names, command lines, etc. A similar analysis of Windows data shows that about 2M events are stored occupying about 24MB, and that the 781K nodes take up about 53B/node.

Dataset	Total No. of Nodes	Total Events	Memory Usage(MB)
Linux Desktop	1.1M	72.6M	111
Windows Desktop	781K	10.3M	67
SSH/File Server	430K	14.4M	45
Web Server	141K	2.8M	16
Mail Server	189K	3M	21
Total	**2.64M**	**103.1M**	**260**

Table 3.3: Memory usage. The second column gives the total number of nodes in the dependence graph before any versioning. The third column gives the total number of events. The fourth column give the total memory usages after using the optimizations. Average memory use across these data sets is less than 5 bytes/event.

Dataset	Versions per node	
	Naive	Optimized
Linux Desktop	68.65	1.05
Windows Desktop	13.9	1.37
SSH/File Server	34.36	1.31
Web Server	20.62	1.29
Mail Server	16.20	1.32
Average	**25.58**	**1.26**

Table 3.4: Impact of naive and optimized versioning. Geometric means are reported on the last row of the table.

Effectiveness of Version Reduction Optimizations

Table 3.4 shows the number of node versions created with the naive versioning algorithm and our optimized algorithm. The second column shows that naive versioning leads to a version explosion, with about 26 versions per node. However, optimization versioning reduce the number versions by creating just about 1.3 versions per node, on average.

Table 3.5 breaks out the effects of optimizations individually. Since some optimizations require other optimizations, we show the four most meaningful combinations: (a) no optimizations, (b) all optimizations except redundant node (RNO), (c) all optimizations except Cycle-Collapsing (CCO), and (d) all optimizations. When all optimizations other than RNO are enabled, the number of versions falls to about 3.6× from 25.6× (unoptimized). Enabling all optimizations except CCO leads to about 3 versions on average per node. Comparing these with the last column, we can conclude that RNO contributes about a 3× reduction and CCO a 2.4× reduction in the number of versions, with the remaining 2.8× coming from REO. It should be noted that REO and CCO both remove versions as well as edges, whereas RNO removes only nodes.

Dataset	Versions per node			
	None	No RNO	No CCO	All
Linux Desktop	68.65	4.56	17.75	1.05
Windows Desktop	13.9	2.60	1.38	1.37
SSH/File Server	34.36	4.32	2.21	1.31
Web Server	20.62	3.46	2.15	1.29
Mail Server	16.20	3.57	2.12	1.32
Average	**25.58**	**3.63**	**3.01**	**1.26**

Table 3.5: Effectiveness of different versioning optimizations. Geometric means are reported on the last row of the table.

Chapter 4

Real-Time Attack Scenario Reconstruction From COTS Audit Data

In this chapter we present an approach and system for real-time reconstruction of attack scenarios on an enterprise host called SLEUTH. To meet the scalability and real-time needs of the problem, we develop a platform-neutral, main-memory based, dependency graph abstraction of audit-log data using the techniques discussed in the previous chapter. We present efficient, tag-based techniques for attack detection and reconstruction, including source identification and impact analysis. We also develop methods to reveal the big picture of attacks by construction of compact, visual graphs of attack steps.

4.1 Approach Overview and Contributions

SLEUTH is OS-neutral, and currently supports Microsoft Windows, Linux and FreeBSD. Audit data from these OSes is processed into a platform-neutral graph representation, where vertices represent subjects (processes) and objects (files, sockets), and edges denote audit events (e.g., operations such as read, write, execute, and connect). This graph serves as the basis for attack detection as well as causality analysis and scenario reconstruction. Using the techniques discussed in chapter 3 we developed a compact main-memory dependence graph representation. As mentioned earlier, graph algorithms on main memory representation can be orders of magnitude faster than on-disk representations, an important factor in achieving real-time analysis capabilities. In our experiments, we were able to process 79 hours worth of audit data from a FreeBSD system in 14 seconds, with a main memory usage of 84MB. This performance represents an analysis rate that is 20K times faster than the rate at which the data was generated.

We introduce a tag-based approach for identifying subjects, objects and events

that are most likely involved in attacks. Tags enable us to prioritize and focus our analysis, thereby addressing the second challenge mentioned above. Tags encode an assessment of *trustworthiness*[1] and *sensitivity* of data (i.e., objects) as well as processes (subjects). This assessment is based on data provenance derived from audit logs. In this sense, tags derived from audit data are similar to coarse-grain information flow labels. Our analysis can naturally support finer-granularity tags as well, e.g., fine-grained taint tags [113, 160], if they are available. Tags are described in more detail in Section 4.2, together with their application to attack detection.

Next we have developed a novel algorithm that leverage tags for root-cause identification and impact analysis (Section 4.4). Starting from alerts produced by the attack detection component, our backward analysis algorithm follows the dependencies in the graph to identify the sources of the attack. Starting from the sources, we perform a full impact analysis of the actions of the adversary using a forward search. We present several criteria for pruning these searches in order to produce a compact graph. We also present a number of transformations that further simplify this graph and produce a graph that visually captures the attack in a succinct and semantically meaningful way, e.g., the graph in Fig. 4.1. Experiments show that our tag-based approach is very effective: for instance, SLEUTH can analyze 38.5M events and produce an attack scenario graph with just 130 events, representing five orders of magnitude reduction in event volume.

We also introduce a customizable policy framework (Section 4.3) for tag initialization and propagation. Our framework comes with sensible defaults, but they can be overridden to accommodate behaviors specific to an OS or application. This enables tuning of our detection and analysis techniques to avoid false positives in cases where benign applications exhibit behaviors that resemble attacks. (See Section 4.5.6 for details.) Policies also enable an analyst to test out "alternate hypotheses" of attacks, by reclassifying what is considered trustworthy or sensitive and re-running the analysis. If an analyst suspects that some behavior is the result of an attack, they can also use policies to capture these behaviors, and rerun the analysis to discover its cause and impact. Since we can process and analyze audit data tens of thousands of times faster than the rate at which it is generated, efficient, parallel, real-time testing of alternate hypotheses is possible.

Our experimental evaluation (Section 4.5) is based mainly on a red team evaluation organized by DARPA as part of its Transparent Computing program. In this evaluation, attack campaigns resembling modern APTs were carried out on Windows, FreeBSD and Linux hosts over a two week period. In this evaluation, SLEUTH was able to:

- process, in a matter of seconds, audit logs containing tens of millions of events

[1]In this chapter, we use the term *trustworthiness tag* in order to highlight its relationship to whether we trust the sources influencing a node. Throughout the rest of this thesis, we have used the term *integrity* because the term goes hand-in-hand with *confidentiality.* The terms *integrity tag* and *trustworthiness tag* are interchangeable in this thesis.

generated during the engagement;

- successfully detect and reconstruct the details of these attacks, including their entry points, activities in the system, and exfiltration points;
- filter away extraneous events, achieving very high reductions rates in the data (up to 100K times), thus providing a clear semantic representation of these attacks containing almost no noise from other activities in the system; and
- achieve low false positive and false negative rates.

Our evaluation is not intended to show that we detected the most sophisticated adversary; instead, our point is that, given several unknown possibilities, the prioritized results from our system can be right on spot in real-time, without any human assistance. Thus, it really fills a gap that exists today, where forensic analysis seems to be primarily initiated manually.

4.2 Tags and Attack Detection

We use tags to summarize our assessment of the trustworthiness and sensitivity of objects and subjects. This assessment can be based on three main factors:

- *Provenance:* the tags on the immediate predecessors of an object or subject in the dependence graph,
- *Prior system knowledge:* our knowledge about the behavior of important applications, such as remote access servers and software installers, and important files such as */etc/passwd* and */dev/audio*, and
- *Behavior:* observed behavior of subjects, and how they compare to their expected behavior.

We have developed a policy framework, described in Section 4.3, for initializing and propagating tags based on these factors. In the absence of specific policies, a default policy is used that propagates tags from inputs to outputs. The default policy assigns to an output the lowest among the trustworthiness tags of the inputs, and the highest among the confidentiality tags. This policy is conservative: it can err on the side of over-tainting, but will not cause attacks to go undetected, or cause a forward (or backward) analysis to miss objects, subjects or events.

Tags play a central role in Sleuth. They provide important context for attack detection. Each audited event is interpreted in the context of these tags to determine its likelihood of contributing to an attack. In addition, tags are instrumental for the speed of our forward and backward analysis. Finally, tags play a central role in scenario reconstruction by eliminating vast amounts of audit data that satisfy the technical definition of dependence but do not meaningfully contribute to our understanding of an attack.

4.2.1 Tag Design

We define the following *trustworthiness tags* (*t-tags*):

- *Benign authentic* tag is assigned to data/code received from sources trusted to be benign, and whose authenticity can be verified.
- *Benign* tag reflects a reduced level of trust than benign authentic: while the data/code is still believed to be benign, adequate authentication hasn't been performed to verify the source.
- *Unknown* tag is given to data/code from sources about which we have no information on trustworthiness. Such data *can sometimes be* malicious.

Policies define what sources are benign and what forms of authentication are sufficient. In the simplest case, these policies take the form of whitelists, but we support more complex policies as well. If no policy is applicable to a source, then its t-tag is set to *unknown.*

We define the following *confidentiality tags* (*c-tags*), to reason about information stealing attacks:

- *Secret:* Highly sensitive information, such as login credentials and private keys.
- *Sensitive:* Data whose disclosure can have a significant security impact, e.g., reveal vulnerabilities in the system, but does not provide a direct way for an attacker to gain access to the system.
- *Private:* Data whose disclosure is a privacy concern, but does not necessarily pose a security threat.
- *Public:* Data that can be widely available, e.g., on public web sites.

An important aspect of our design is the separation between t-tags for code and data. Specifically, a subject (i.e., a process) is given two t-tags: one that captures its *code trustworthiness* (code t-tag) and another for its *data trustworthiness* (data t-tag). This separation significantly improves attack detection. More importantly, it can significantly speed up forensic analysis by focusing it on fewer suspicious events, while substantially reducing the size of the reconstructed scenario. Note that confidentiality tags are associated only with data (and not code).

Pre-existing objects and subjects are assigned initial tags using *tag initialization policies.* Objects representing external entities, such as a remote network connection, also need to be assigned initial tags. The rest of the objects and subjects are created during system execution, and their tags are determined using *tag propagation policies.* Finally, attacks are detected using behavior-based policies called *detection policies.*

As mentioned before, if no specific policy is provided, then sources are tagged with *unknown* trustworthiness. Similarly, in the absence of specific propagation policies, the default conservative propagation policy is used.

4.2.2 Tag-based Attack Detection

An important constraint in SLEUTH is that we are limited to information available in audit data. This suggests the use of provenance reflected in audit data as a possible basis for detection. Since tags are a function of provenance, we use them for attack detection. Note that in our threat model, audit data is trustworthy, so tags provide a sound basis for detection.

A second constraint in SLEUTH is that detection methods should not require detailed application-specific knowledge. In contrast, most existing intrusion detection and sandboxing techniques interpret each security-sensitive operation in the context of a specific application to determine whether it could be malicious. This requires expert knowledge about the application, or in-the-field training in a dynamic environment, where applications may be frequently updated.

Instead of focusing on application behaviors that tend to be variable, we focus our detection techniques on the high-level objectives of most attackers, such as backdoor insertion and data exfiltration. Specifically, we combine reasoning about an attacker's *motive* and *means.* If an event in the audit data can help the attacker achieve his/her key high-level objectives, that would provide the motivation and justification for using that event in an attack. But this is not enough: the attacker also needs the means to cause this event, or more broadly, influence it. Note that our tags are designed to capture means: if a piece of data or code bears the *unknown* t-tag, then it was derived from (and hence influenced by) untrusted sources.

As for the high-level objectives of an attacker, several reports and white papers have identified that the following steps are typical in most advanced attack campaigns [2, 8, 93]:

1. Deploy and run attacker's code on victim system.
2. Replace or modify important files, e.g., `/etc/passwd` or ssh keys.
3. Exfiltrate sensitive data.

Attacks with a transient effect may be able to avoid the first two steps, but most sophisticated attacks, such as those used in APT campaigns, require the establishment of a more permanent footprint on the victim system. In those cases, there does not seem to be a way to avoid one or both of the first two steps. Even in those cases where the attacker's goal could be achieved without establishing a permanent base, the third step usually represents an essential attacker goal.

Based on the above reasoning, we define the following policies for attack detection that incorporate the attacker's objectives and means:

- *Untrusted code execution:* This policy triggers an alarm when a subject with a higher code t-tag executes (or loads) an object with a lower t-tag[2].

[2]Customized policies can be defined for interpreters such as `bash` so that reads are treated the same as loads.

- *Modification by subjects with lower code t-tag:* This policy raises an alarm when a subject with a lower code t-tag modifies an object with a higher t-tag. Modification may pertain to the file content or other attributes such as name, permissions, etc.

- *Confidential data leak:* An alarm is raised when untrusted subjects exfiltrate sensitive data. Specifically, this policy is triggered on network writes by subjects with a *sensitive* c-tag and a code t-tag of *unknown.*

- *Preparation of untrusted data for execution:* This policy is triggered by an operation by a subject with a code t-tag of *unknown,* provided this operation makes an object executable. Such operations include `chmod` and `mprotect`[3,4].

It is important to note that "means" is not diluted just because data or code passes through multiple intermediaries. For instance, the untrusted code policy does not require a direct load of data from an unknown web site; instead, the data could be downloaded, extracted, uncompressed, and possibly compiled, and then loaded. Regardless of the number of intermediate steps, this policy will be triggered when the resulting file is loaded or executed. This is one of the most important reasons for the effectiveness of our attack detection.

Today's vulnerability exploits typically do not involve untrusted code in their first step, and hence won't be detected by the untrusted code execution policy. However, the eventual goal of an attacker is to execute his/her code, either by downloading and executing a file, or by adding execute permissions to a memory page containing untrusted data. In either case, one of the above policies can detect the attack. A subsequent backward analysis can help identify the first step of the exploit.

Additional detector inputs can be easily integrated into Sleuth. For instance, if an external detector flags a subject as a suspect, this can be incorporated by setting the code t-tag of the subject to *unknown.* As a result, the remaining detection policies mentioned above can all benefit from the information provided by the external detector. Moreover, setting of *unknown* t-tag at suspect nodes preserves the dependency structure between the graph vertices that cause alarms, a fact that we exploit in our forensic analysis.

The fact that many of our policies are triggered by untrusted code execution should not be interpreted to mean that they work in a static environment, where no new code is permitted in the system. Indeed, we expect software updates and upgrades to be happening constantly, but in an enterprise setting, we don't expect end users to be downloading unknown code from random sites. Accordingly, we subsequently

[3]Binary code injection attacks on today's OSes ultimately involve a call to change the permission of a writable memory page so that it becomes executable. To the extent that such memory permission change operations are included in the audit data, this policy can spot them.

[4]Our implementation can identify `mprotect` operations that occur in conjunction with library loading operations. This policy is not triggered on those `mprotect`'s.

describe how to support standardized software updating mechanisms such as those used on contemporary OSes.

4.3 Policy Framework

We have developed a flexible policy framework for tag assignment, propagation, and attack detection. We express policies using a simple rule-based notation, e.g.,

$$exec(s, o)\text{: } o.ttag < benign \rightarrow alert(\texttt{"UntrustedExec"})$$

This rule is triggered when the subject s executes a (file) object o with a t-tag less than *benign*. Its effect is to raise an alert named `UntrustedExec`. As illustrated by this example, rules are generally associated with events, and include conditions on the attributes of objects and/or subjects involved in the event. The effect of a policy depends on its type. The effect of a detection policy is to raise an alarm. For tag initialization and propagation policies, the effect is to modify tag(s) associated with the object or subject involved in the event.

Our current detection policies are informally described in the previous section. We therefore focus in this section on our current tag initialization and propagation policies.

Tag Initialization Policies

These policies are invoked to initialize tags for new objects, or preexisting objects when they are first mentioned in the audit data. Recall that when a subject creates a new object, the object inherits the subject's tags by default; however, in case of a preexisting object tag initialization policies are triggered. For example,

$$init(o)\text{: } o.type == \texttt{FILE} \rightarrow$$
$$o.ttag = \texttt{BENIGN\_AUTH}, o.ctag = \texttt{PUBLIC}$$

Tag Propagation Policies

These policies can be used to override default tag propagation semantics. Different tag propagation policies can be defined for different groups of related event types.Tag propagation policies can be used to prevent "over-tainting" that can result from files such as `.bash_history` that are repeatedly read and written by an application each time it is invoked. For example, the following policy skips taint propagation for this specific file:

$propRd(s, o)\text{: } match(o.name, \texttt{"\textbackslash.bash\_history\$"}) \rightarrow skip$[5]

[5]Here, "skip" means do nothing, i.e., leave tags unchanged.

4.4 Tag-Based Bi-Directional Analysis

4.4.1 Backward Analysis

The goal of backward analysis is to identify the entry points of an attack campaign. Entry points are the nodes in the graph with an in-degree of zero and are marked untrusted. Typically they represent network connections, but they can also be of other types, e.g., a file on a USB stick that was plugged into the victim host.

The starting points for the backward analysis are the alarms generated by the detection policies. In particular, each alarm is related to one or more entities, which are marked as suspect nodes in the graph. Backward search involves a backward traversal of the graph to identify paths that connect the suspect nodes to entry nodes. We note that the direction of the dependency edges is reversed in such a traversal and in the following discussions. Backward search poses several significant challenges:

- *Performance:* The dependence graph can easily contain hundreds of millions of edges. Alarms can easily number in thousands. Running backward searches on such a large graph is computationally expensive.

- *Multiple paths:* Typically numerous entry points are backward reachable from a suspect node. However, in APT-style attacks, there is often just one real entry point. Thus, a naive backward search can lead to a large number of false positives.

The key insight behind our approach is that tags can be used to address both challenges. In fact, tag computation and propagation is already an implicit path computation, which can be reused. Furthermore, a tag value of *unknown* on a node provides an important clue about the likelihood of that node being a potential part of an attack. In particular, if an *unknown* tag exists for some node A, that means that there exists at least a path from an untrusted entry node to node A, therefore node A is more likely to be part of an attack than other neighbors with *benign* tags. Utilizing tags for the backward search greatly reduces the search space by eliminating many irrelevant nodes and sets SLEUTH apart from other scenario reconstruction approaches such as [72, 86].

Based on this insight, we formulate backward analysis as an instance of shortest path problem, where tags are used to define edge costs. In effect, tags are able to "guide" the search along relevant paths, and away from unlikely paths. This factor enables the search to be completed without necessarily traversing the entire graph, thus addressing the performance challenge. In addition, our shortest path formulation addresses the multiple paths challenge by preferring the entry point closest (as measured by path cost) to a suspect node.

Dataset	Duration (hh-mm-ss)	Open	Connect + Accept	Read	Write	Clone + Exec	Close + Exit	Mmap / Loadlib	Others	Total # of Events	Scenario Graph
W-1	06:22:42	N/A	22.14%	44.70%	5.12%	3.73%	3.88%	17.40%	3.02%	100K	Fig. 4.6
W-2	19:43:46	N/A	17.40%	47.63%	8.03%	3.28%	3.26%	15.22%	5.17%	401K	Fig. 4.2
L-1	07:59:26	37%	0.11%	18.01%	1.15%	0.92%	38.76%	3.97%	0.07%	2.68M	Fig. 4.3
L-2	79:06:39	39.58%	0.08%	12.19%	2%	0.83%	41.28%	3.79%	0.25%	38.5M	-
L-3	79:05:13	38.88%	0.04%	11.81%	2.35%	0.95%	40.98%	4.14%	0.84%	19.3M	Fig. 4.7
F-1	08:17:30	9.46%	0.40%	24.65%	40.86%	2.10%	12.55%	9.08%	0.89%	701K	Fig. 4.4
F-2	78:56:48	11.78%	0.42%	16.60%	44.52%	2.10%	15.04%	8.54%	1.01%	5.86M	Fig. 4.5
F-3	79:04:54	11.31%	0.40%	19.46%	45.71%	1.64%	14.30%	6.16%	1.03%	5.68M	Fig. 4.1
Benign	329:11:40	11.68%	0.71%	26.22%	30.03%	0.63%	15.42%	14.32%	0.99%	32.83M	N/A

Table 4.1: Dataset for each campaign with duration, distribution of different system calls and total number of events.

For shortest path, we use Dijkstra's algorithm, as it discovers paths in increasing order of cost. In particular, each step of this algorithm adds a node to the shortest path tree, which consists of the shortest paths computed so far. This enables the search to stop as soon as an entry point node is added to this tree.

Cost function design. Our design assigns low costs to edges representing dependencies on nodes with *unknown* tags, and higher costs to other edges. Specifically, the costs are as follows:

- Edges that introduce a dependency from a node with *unknown* code or data t-tag to a node with *benign* code or data t-tag are assigned a cost of 0.
- Edges introducing a dependency from a node with *benign* code and data t-tags are assigned a high cost.
- Edges introducing dependencies between nodes already having an *unknown* tag are assigned a cost of 1.

The intuition behind this design is as follows. A benign subject or object immediately related to an `unknown` subject/object represents the boundary between the malicious and benign portions of the graph. Therefore, they must be included in the search, thus the cost of these edges is 0. Information flows among benign entities are not part of the attack, therefore we set their cost to very high so that they are excluded from the search. Information flows among untrusted nodes are likely part of an attack, so we set their cost to a low value. They will be included in the search result unless alternative paths consisting of fewer edges are available.

4.4.2 Forward Analysis

The purpose of forward analysis is to assess the impact of a campaign, by starting from an entry point and discovering all the possible effects dependent on the entry

point. Similar to backward analysis, the main challenge is the size of the graph. A naive approach would identify and flag all subjects and objects reachable from the entry point(s) identified by backward analysis. Unfortunately, such an approach will result in an impact graph that is too large to be useful to an analyst. For instance, in our experiments, a naive analysis produced impact graphs with millions of edges, whereas our refined algorithm reduces this number by 100x to 500x.

A natural approach for reducing the size is to use a distance threshold d_{th} to exclude nodes that are "too far" from the suspect nodes. Threshold d_{th} can be interactively tuned by an analyst. We use the same cost metric that was used for backward analysis, but modified to consider confidentiality[6]. In particular, edges between nodes with high confidentiality tags (e.g., *secret*) and nodes with low code integrity tags (e.g., *unknown* process) or low data integrity tags (e.g., *unknown* socket) are assigned a cost of 0, while edges to nodes with *benign* tags are assigned a high cost.

4.4.3 Reconstruction and Presentation

We apply the following simplifications to the output of forward analysis, in order to provide a more succinct view of the attack:

- *Pruning uninteresting nodes.* The result of forward analysis may include many dependencies that are not relevant for the attack, e.g., subjects writing to cache and log files, or writing to a temporary file and then removing it. These nodes may appear in the results of the forward analysis but no suspect nodes depend on them, so they can be pruned.

- *Merging entities with the same name.* This simplification merges subjects that have the same name, disregarding their process ids and command-line arguments.

- *Repeated event filtering.* This simplification merges into one those events that happen multiple times (e.g., multiple writes, multiple reads) between the same entities. If there are interleaving events, then we show two events representing the first and the last occurrence of an event between the two entities.

4.5 Experimental Evaluation

4.5.1 Implementation

Most components of SLEUTH, including the graph model, policy engine, attack detection and some parts of the forensic analysis are implemented in C++, and consist

[6]Recall that some alarms are related to exfiltration of confidential data, so we need to decide which edges representing the flow of confidential information should be included in the scenario.

of about 9.5KLoC.

4.5.2 Data Sets

Table 4.1 summarizes the dataset used in our evaluation. The first eight rows of the table correspond to attack campaigns carried out by a red team as part of the DARPA Transparent Computing (TC) program. This set spans a period of 358 hours, and contains about 73 million events. The last row corresponds to benign data collected over a period of 3 to 5 days across four Linux servers in our research laboratory.

Attack data sets were collected on Windows (W-1 and W-2), Linux (L-1 through L-3) and FreeBSD (F-1 through F-3) by three research teams that are also part of the DARPA TC program. The goal of these research teams is to provide fine-grained provenance information that goes far beyond what is found in typical audit data. However, at the time of the evaluation, these advanced features had not been implemented in the Windows and FreeBSD data sets. Linux data set did incorporate finer-granularity provenance (using the unit abstraction developed in [86]), but the implementation was not mature enough to provide consistent results in our tests. For this reason, we omitted any fine-grained provenance included in their dataset, falling back to the data they collected from the built-in auditing system of Linux. The FreeBSD team built their capabilities over DTrace. Their data also corresponded to roughly the same level as Linux audit logs. The Windows team's data was roughly at the level of Windows event logs. All of the teams converted their data into a common representation to facilitate analysis.

The "duration" column in Table 4.1 refers to the length of time for which audit data was emitted from a host. Note that this period covers both benign activities and attack related activities on a host. The next several columns provide a break down of audit log events into different types of operations. File open and close operations were not included in W-1 and W-2 data sets. Note that "read" and "write" columns include not only file reads/writes, but also network reads and writes on Linux. However, on Windows, only file reads and writes were reported. Operations to load libraries were reported on Windows, but memory mapping operations weren't. On Linux and FreeBSD, there are no load operations, but most of the mmap calls are related to loading. So, the mmap count is a loose approximation of the number of loads on these two OSes. The "Others" column includes all the remaining audit operations, including `rename`, `link`, `rm`, `unlink`, `chmod`, `setuid`, and so on. The last column in the table identifies the scenario graph constructed by SLEUTH for each campaign. Due to space limitations, we have omitted scenario graphs for campaign L-2.

4.5.3 Engagement Setup

The attack scenarios in our evaluation are setup as follows. Five of the campaigns (i.e., W-2, L-2, L3, F-2, and F3) ran in parallel for 4 days, while the remaining three

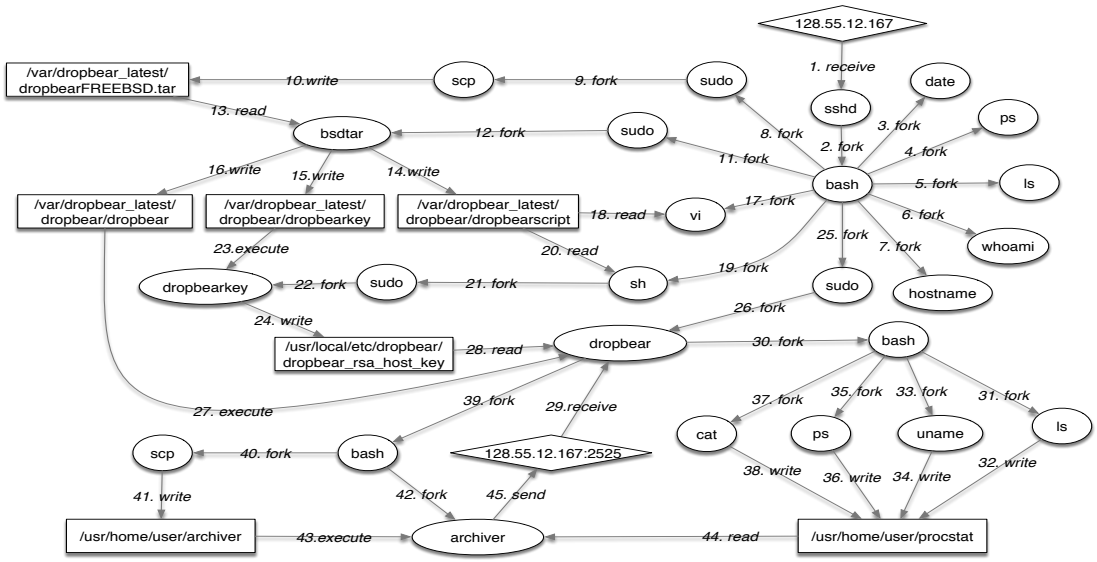

Figure 4.1: Scenario graph reconstructed from campaign F-3.

(W-1, L-1, and F-1) were run in parallel for 2 days. During each campaign, the red team carried out a series of attacks on the target hosts. The campaigns are aimed at achieving varying adversarial objectives, which include dropping and execution of an executable, gathering intelligence about a target host, backdoor injection, privilege escalation, and data exfiltration.

Being an adversarial engagement, we had no prior knowledge of the attacks planned by the red team. We were only told the broad range of attacker objectives described in the previous paragraph. It is worth noting that, while the red team was carrying out attacks on the target hosts, benign background activities were also being carried out on the hosts. These include activities such as browsing and downloading files, reading and writing emails, document processing, and so on. On average, *more than 99.9% of the events corresponded to benign activity.* Hence, SLEUTH had to automatically detect and reconstruct the attacks from a set of events including both benign and malicious activities.

We present our results in comparison with the ground truth data released by the red team. Before the release of ground truth data, we had to provide a report of our findings to the red team. The findings we report in this thesis match the findings we submitted to the red team. A summary of our detection and reconstruction results is provided in a tabular form in Table 4.3. Below, we first present reconstructed scenarios for selected datasets before proceeding to a discussion of these summary results.

4.5.4 Selected Reconstruction Results

Of the 8 attack scenarios successfully reconstructed by SLEUTH, we discuss campaigns W-2 (Windows) and F-3 (FreeBSD) in this section, while deferring the rest to Section 4.5.10. To make it easier to follow the scenario graph, we provide a narrative that explains how the attack unfolded. This narrative requires manual interpretation of the graph, but the graph generation itself is automated. In these graphs, edge labels include the event name and a sequence number that indicates the global order in which that event was performed. Ovals, diamonds and rectangles represent processes, sockets and files, respectively.

Campaign W-2. Figure 4.2 shows the graph reconstructed by SLEUTH from Windows audit data. Although the actual attack campaign lasted half an hour, the host was running benign background activities for 20 hours. These background activities corresponded to more than 99.8% of the events in the corresponding audit log.
Entry: The initial entry point for the attack is `Firefox`, which is compromised on visiting the web server `129.55.12.167`.
Backdoor insertion: Once Firefox is compromised, a malicious program called `dropper` is downloaded and executed. Dropper seems to provide a remote interactive shell, connecting to ports 443 and then 4430 on the attack host, and executing received commands using `cmd.exe`.
Intelligence gathering: Dropper then invokes `cmd.exe` multiple times, using it to perform various data gathering tasks. The programs `whoami`, `hostname` and `netstat` are being used as stand-ins for these data gathering applications. The collected data is written to `C:\Users\User1\Documents\Thumbs\thumbit\test\thumbs.db`.
Data exfiltration: Then the collected intelligence is exfiltrated to 129.55.12.51:9418 using `git`.
Clean-up: Dropper downloads a batch file called `burnout.bat`. This file contains commands to clean up the attack footprint, which are executed by `cmd.exe` (see edges 11,12, 31-33).

Campaign F-3. (Figure 4.1). Under the command of an attacker who uses stolen ssh credentials, `sshd` forks a `bash` process. Note that though there is no direct evidence from the audit data about the stolen ssh credentials, because of the subsequent events (`scp`) from this shell, we conclude this as a sign of an attacker that uses stolen ssh credentials.

Next the attacker, invokes `scp`, which downloads a file into location `/var/dropbear_latest/dropbearFREEBSD.tar`, which is then uncompressed. The file `dropbearscript` is next read and interpreted by `sh`. This action creates the process `dropbearkey`, which writes to `/usr/ local/etc/dropbear/dropbear_ecdsa_host_key` and `/usr/local/etc/dropbear/dropbear_rsa_host_key`. Next, another `sudo` process

created by `bash` starts another `dropbear` process which reads these two keys for future use (presumably to assist in connecting to a remote host).

Dropbear next starts a shell process, which executes a series of commands `ls`, `bash`, `uname`, `ps`, all of which write to a file `/usr/home/user/procstats`.

Finally, `dropbear` starts a `bash` process, which uses `scp` to download a file called `/usr/home/user/archiver`, and executes that file. The resulting process, called `archiver`, reads the file `/usr/home/user/procstats`, which contains the data output earlier, and exfiltrates this information to `128.55.12.167:2525`.

Summary. The above two graphs were constructed automatically by SLEUTH from audit data. They demonstrate how SLEUTH enables an analyst to obtain compact yet complete attack scenarios from hours of audit data. SLEUTH is able to hone in on the attack activity, even when it is hidden among benign data that is at least three orders of magnitude larger.

4.5.5 Overall Effectiveness

Dataset	Drop & Load	Intelligence Gathering	Backdoor Insertion	Privilege Escalation	Data Exfiltration	Cleanup
W-1	✓	✓			✓	✓
W-2	✓	✓	✓		✓	✓
L-1	✓	✓	✓		✓	✓
L-2	✓	✓	✓	✓	✓	✓
L-3	✓	✓	✓	✓	✓	✓
F-1			✓		✓	
F-2	✓	✓	✓		✓	
F-3	✓	✓			✓	

Table 4.2: SLEUTH results with respect to a typical APT campaign.

To assess the effectiveness of SLEUTH in capturing essential stages of an APT, in Table 4.2, we correlate pieces of attack scenarios constructed by SLEUTH with APT stages documented in postmortem reports of notable APT campaigns (e.g., the MANDIANT [8] report). In 7 of the 8 attack scenarios, SLEUTH uncovered the drop&load activity. In all the scenarios, SLEUTH captured concrete evidence of data exfiltration, a key stage in an APT campaign. In 7 of the scenarios, commands used by the attacker to gather information about the target host were captured by SLEUTH.

Another distinctive aspect of an APT is the injection of backdoors to targets and their use for C&C and data exfiltration. In this regard, 6 of the 8 scenarios reconstructed by SLEUTH involve backdoor injection. Cleaning the attack footprint is a common element of an APT campaign. In our experiments, in 5 of the 8 scenarios,

SLEUTH uncovered attack cleanup activities, e.g., removing dropped executables and data files created during the attack.

Dataset	Entry Entities	Programs Executed	Key Files	Exit Points	Correctly Identified Entities	Incorrectly Identified Entities	Missed Entities
W-1	2	8	7	3	20	0	0
W-2	2	8	4	4	18	0	0
L-1	2	10	7	2	20	0	1
L-2	2	20	11	4	37	0	0
L-3	1	6	6	5	18	0	0
F-1	4	13	9	2	13	0	1
F-2	2	10	7	3	22	0	0
F-3	4	14	7	1	26	0	0
Total	**19**	**89**	**58**	**24**	**174**	**0**	**2**

Table 4.3: Attack scenario reconstruction summary.

Table 4.3 shows another way of breaking down the attack scenario reconstruction results, counting the number of key files, network connections, and programs involved in the attack. Specifically, we count the number of attack entry entities (including the entry points and the processes that communicate with those entry points), attack-related program executions, key files that were generated and used during the campaign, and the number of exit points used for exfiltration (e.g., network sockets). This data was compared with the ground truth, which was made available to us after we obtained the results. The last two columns show the incorrectly reported and missed entities, respectively.

The two missed entities were the result of the fact that we had not spent any effort in cataloging sensitive data files and device files. As a result, these entities were filtered out during the forward analysis and simplification steps. Once we marked the two files correctly, they were no longer filtered out, and we were able to identify all of the key entities.

In addition to the missed entities shown in Table 4.3, the red team reported that we missed a few other attacks and entities. Some of these were in data sets we did not examine. In particular, campaign W-2 was run multiple times, and we examined the data set from only one instance of it. Also, there was a third attack campaign W-3 on Windows, but the team producing Windows data sets had difficulties during W-3 that caused the attack activities not to be recorded, so that data set is omitted from the results in Table 4.3. Similarly, the team responsible for producing Linux data sets had some issues during campaign L-3 that caused some attack activities not to be recorded. To account for this, Table 4.3 counts only the subset of key entities whose names are present in the L-3 data set given to us.

According to the ground truth provided by the red team, we incorrectly identified 21 entities in F-1 that were not part of an attack. Subsequent investigation showed

that the auditing system had not been shutdown at the end of the F-1 campaign, and all of these false positives correspond to testing/administration steps carried out after the end of the engagement, when the auditing system should not have been running.

Dataset	Log Size on Disk	# of Events	Duration hh:mm:ss	Packages Updated	Binary Files Written
Server 1	1.1G	2.17M	00:13:06	110	1.8K
Server 2	2.7G	4.67M	105:08:22	4	4.2K
Server 3	12G	20.9M	104:36:43	4	4.3K
Server 4	3.2G	5.09M	119:13:29	4	4.3K

Table 4.4: False alarms in a benign environment with software upgrades and updates. No alerts were triggered during this period.

4.5.6 False Alarms in a Benign Environment

In order to study SLEUTH's performance in a benign environment, we collected audit data from four Ubuntu Linux servers over a period of 3 to 5 days. One of these is a mail server, another is a web server, and a third is an NFS/SSH/SVN server. Our focus was on software updates and upgrades during this period, since these updates can download code from the network, thereby raising the possibility of untrusted code execution alarms. There were four security updates (including kernel updates) performed over this period. In addition, on a fourth server, we collected data when a software upgrade was performed, resulting in changes to 110 packages. Several thousand binary and script files were updated during this period, and the audit logs contained over 30M events. All of this information is summarized in Table 4.4.

As noted before, policies should be configured to permit software updates and upgrades using standard means approved in an enterprise. For Ubuntu Linux, we had one policy rule for this: when *dpkg* was executed by *apt*-commands, or by *unattended-upgrades*, the process is not downgraded even when reading from files with untrusted labels. This is because both *apt* and *unattended-upgrades* verify and authenticate the hash on the downloaded packages, and only after these verifications do they invoke *dpkg* to extract the contents and write to various directories containing binaries and libraries. Because of this policy, all of the 10K+ files downloaded were marked benign. As a result of this, no alarms were generated from their execution by SLEUTH.

Dataset	Duration (hh:mm:ss)	Memory Usage	Runtime	
			Time	Speed-up
W-1	06:22:42	3 MB	1.19 s	19.3 K
W-2	19:43:46	10 MB	2.13 s	33.3 K
W-Mean		6.5 MB		**26.3 K**
L-1	07:59:26	26 MB	8.71 s	3.3 K
L-2	79:06:39	329 MB	114.14s	2.5 K
L-3	79:05:13	175 MB	74.14 s	3.9 K
L-Mean		177 MB		**3.2 K**
F-1	08:17:30	8 MB	1.86 s	16 K
F-2	78:56:48	84 MB	14.02 s	20.2 K
F-3	79:04:54	95 MB	15.75 s	18.1 K
F-Mean		62.3 MB		**18.1 K**

Table 4.5: Memory use and runtime for scenario reconstruction.

4.5.7 Runtime and Memory Use

Table 4.5 shows the runtime and memory used by SLEUTH for analyzing various scenarios. The measurements were made on a Ubuntu 16.04 server with 2.8GHz AMD Opteron 62xx processor and 48GB main memory. Only a single core of a single processor was used. The first column shows the campaign name, while the second shows the total duration of the data set.

The third column shows the memory used for the dependence graph. The compact representation using the techniques described in Chapter 3 enables SLEUTH to store data spanning very long periods of time. As an example, consider campaign L-2, whose data were the most dense. SLEUTH used approximately 329MB to store 38.5M events spanning about 3.5 days. Across all data sets, SLEUTH needed about 8 bytes of memory per event on the larger data sets, and about 20 bytes per event on the smaller data sets.

The fourth column shows the total run time, including the times for consuming the dataset, constructing the dependence graph, detecting attacks, and reconstructing the scenario. We note that this time was measured after the engagement when all the data sets were available. During the engagement, SLEUTH was consuming these data as they were being produced. Although the data typically covers a duration of several hours to a few days, the analysis itself is very fast, taking just seconds to a couple of minutes. Because of our use of tags, most information needed for the analysis is locally available. This is the principal reason for the performance we achieve.

The "speed-up" column illustrates the performance benefits of SLEUTH. It can be thought of as the number of simultaneous data streams that can be handled by

Dataset	Untrusted execution		Modification by low code t-tag subject		Preparation of untrusted data for execution		Confidential data leak	
	Single t-tag	Split t-tags	Single t-tag	Split t-tags	Single t-tags	Split t-tags	Single t-tag	Split t-tags
W-1	21	3	1.2 K	3	0	0	6.1 K	11
W-2	44	2	3.7 K	108	0	0	20.2 K	18
L-1	60	2	53	5	1	1	19	6
L-2	1.5 K	5	19.5 K	1	280	8	122 K	159
L-3	695	5	26.1 K	2	270	0	62.1 K	5.3 K
Average Reduction		**45.39x**		**517x**		**6.24x**		**112x**

Table 4.6: Reduction in (false) alarms by maintaining separate code and data trustworthiness tags. The average reduction shows the average factor of reduction we get for alarms generation when using split trustworthiness tag over single trustworthiness tag.

Sleuth, if CPU use was the only constraint.

In summary, Sleuth is able to consume and analyze audit COTS data from several OSes in real time while having a small memory footprint.

4.5.8 Benefit of split tags for code and data

As described earlier, we maintain two trustworthiness tags for each subject, one corresponding to its code, and another corresponding to its data. By prioritizing detection and forward analysis on code trustworthiness, we cut down vast numbers of alarms, while greatly decreasing the size of forward analysis output.

Table 4.6 shows the difference between the number of alarms generated by our four detection policies with single trustworthiness tag and with the split trustworthiness (code and integrity) tags. Note that the split reduces the alarms by a factor of 100 to over 1000 in some cases.

Table 4.7 shows the improvement achieved in forward analysis as a result of this split. In particular, the increased selectivity reported in column 5 of this table comes from splitting the tag. Note that often, there is a 100x to 1000x reduction in the size of the graph.

4.5.9 Analysis Selectivity

Table 4.7 shows the data reduction pipeline of the analyses in Sleuth. The second column shows the number of original events in each campaign. These events include all the events in the system (benign and malicious) over several days with an overwhelming majority having a benign nature, unrelated to the attack.

The third column shows the final number of events that go into the attack scenario graph.

The fourth column shows the reduction factor when a naive forward analysis with single trustworthiness tag (single t-tag) is used from the entry points identified by

our backward analysis. Note that the graph size is very large in most cases. The fifth column shows the reduction factor using the forward analysis of SLEUTH— which is based on split (code and data) trustworthiness tags. As can be seen from the table, SLEUTH achieved two to three orders of magnitude reduction with respect to single t-tag based analysis.

The output of forward analysis is then fed into the simplification engine. The sixth column shows the reduction factor achieved by the simplifications over the output of our forward analysis. The last column shows the overall reduction we get over original events using split (code and data) trustworthiness tags and performing the simplification.

Overall, the combined effect of all of these steps is very substantial: data sets consisting of tens of millions of edges are reduced into graphs with perhaps a hundred edges, representing five orders of magnitude reduction in the case of L-2 and L-3 data sets, and four orders of magnitude reduction on other data.

4.5.10 Discussion of Additional Attacks

In this section, we provide graphs that reconstruct attack campaigns that weren't discussed in Section 4.5.4. Specifically, we discuss attacks L-1, F-1, F-2, W-1, and L-3.
Attack L-1. In this attack (Figure 4.3), `firefox` is exploited to drop and execute via a shell the file `mozillanightly`. The process `mozillanightly` first downloads and executes `mozillaautoup`, then starts a shell, which spawns several other processes. Next, the information gathered in file `netrecon.log` is exfiltrated and the file removed.

Attack F-1. In this attack (Figure 4.4), the `nginx` server is exploited to drop and execute via shell the file `dropper`. Upon execution, the `dropper` process forks a shell that spawns several processes, which write to a file and reads and writes to sensitive files. In addition, `dropper` communicates with the IP of the attacker. We report in the figure the graph related to the restoration and administration carried out after the engagement, as discussed in Section 4.5.5.

Attack F-2. The start of this attack (Figure 4.5) is similar to F-1. However, upon execution, the `dropper` process downloads three files named `recon, sysman`, and `mailman`. Later, these files are executed and used which are used to exfiltrate data gathered from the system.

Attack W-1. In this attack (Figure 4.6), `firefox` is exploited twice to drop and execute a file `mozillanightly`. The first `mozillanightly` process downloads and executes the file `photosnap.exe`, which takes a screenshot of the victim's screen and saves it to a png file. Subsequently, the jpeg file is exfiltrated by `mozillanightly`. The second `mozillanightly` process downloads and executes two files: 1) `burnout.bat`, which is read, and later used to issue commands to `cmd.exe` to gather data about the system; 2) `mnsend.exe`, which is executed by `cmd.exe` to exfiltrate the data gathered previously.

Attack L-3. In this attack (Figure 4.7), the file `dropbearLINUX.tar` is downloaded and extracted. Next, the program `dropbearkey` is executed to create three keys, which are read by a program `dropbear`, which subsequently performs exfiltration.

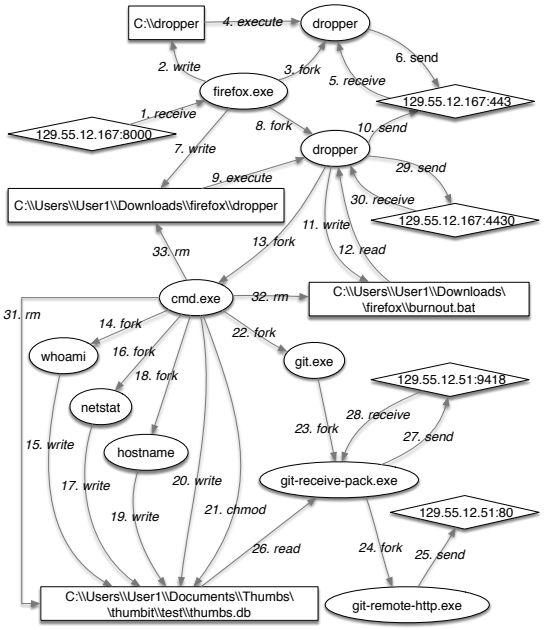

Figure 4.2: Scenario graph reconstructed from campaign W-2.

Dataset	Initial # of Events	Final # of Events	Reduction Factor			
			Single t-tag	Split t-tag	SLEUTH Simplif.	Total
W-1	100 K	51	4.4x	1394x	1.4x	1951x
W-2	401 K	28	3.6x	552x	26x	14352x
L-1	2.68 M	36	8.9x	15931x	4.7x	74875x
L-2	38.5 M	130	7.3x	2971x	100x	297100x
L-3	19.3 M	45	7.6x	1208x	356x	430048x
F-1	701 K	45	2.3x	376x	41x	15416x
F-2	5.86 M	39	1.9x	689x	218x	150202x
F-3	5.68 M	45	6.7x	740x	170x	125800x
Average Reduction			**4.68x**	**1305x**	**41.8x**	**54517x**

Table 4.7: Comparison of selectivity achieved using forward analysis with single trustworthiness tags, forward analysis with split code and data trustworthiness tags, and finally simplifications.

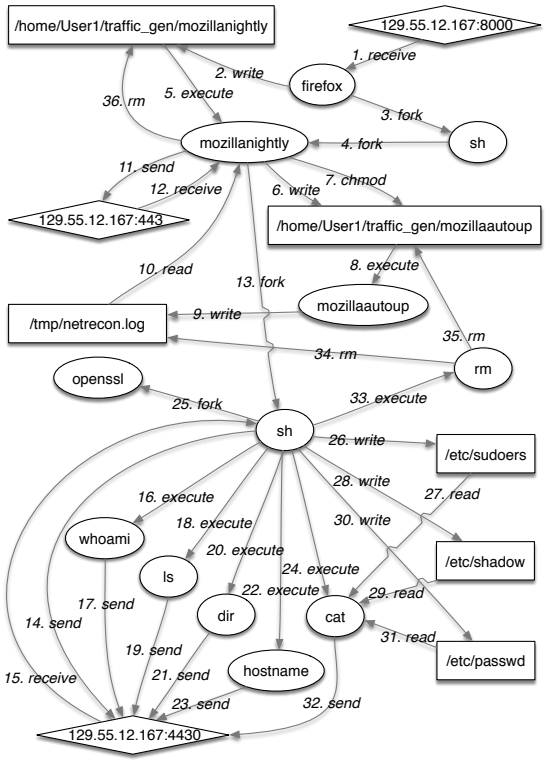

Figure 4.3: Scenario graph reconstructed from campaign L-1.

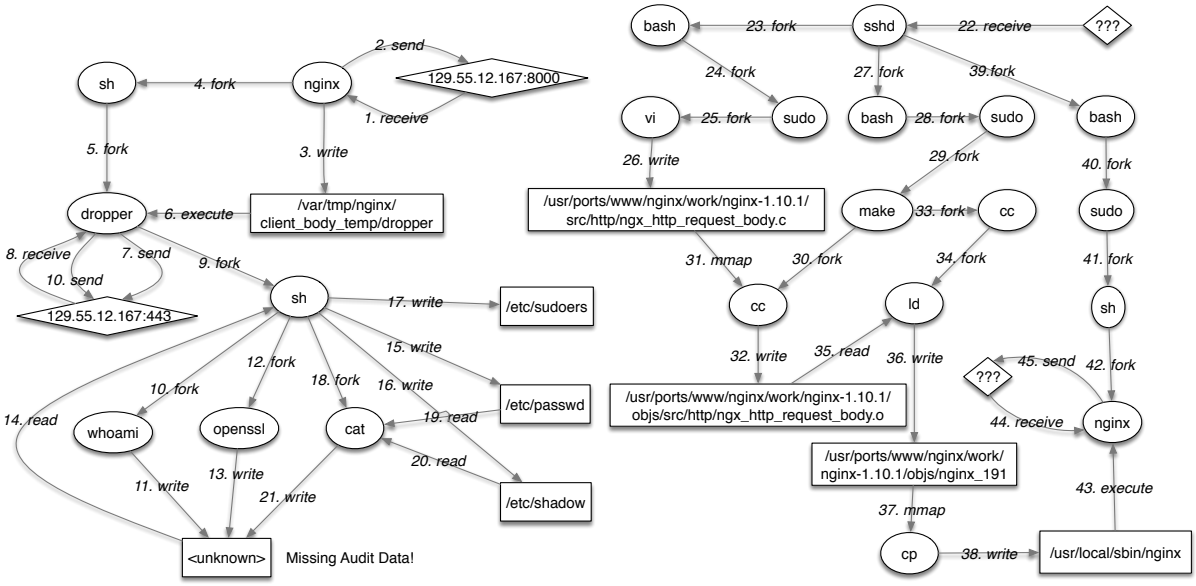

Figure 4.4: Scenario graph reconstructed from campaign F-1.

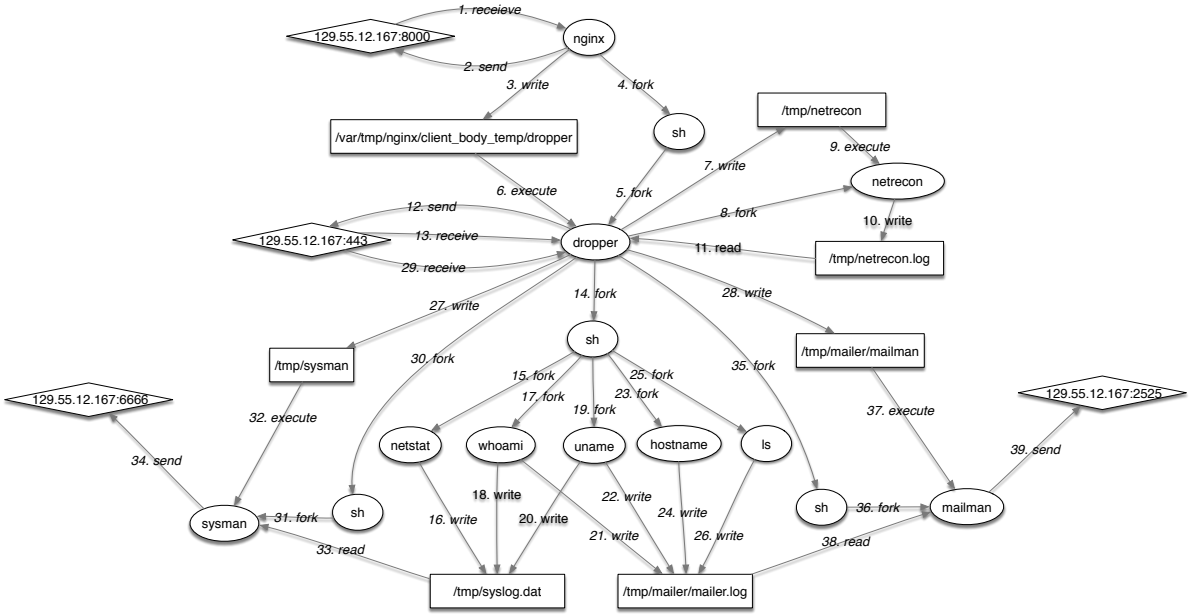

Figure 4.5: Scenario graph reconstructed from campaign F-2.

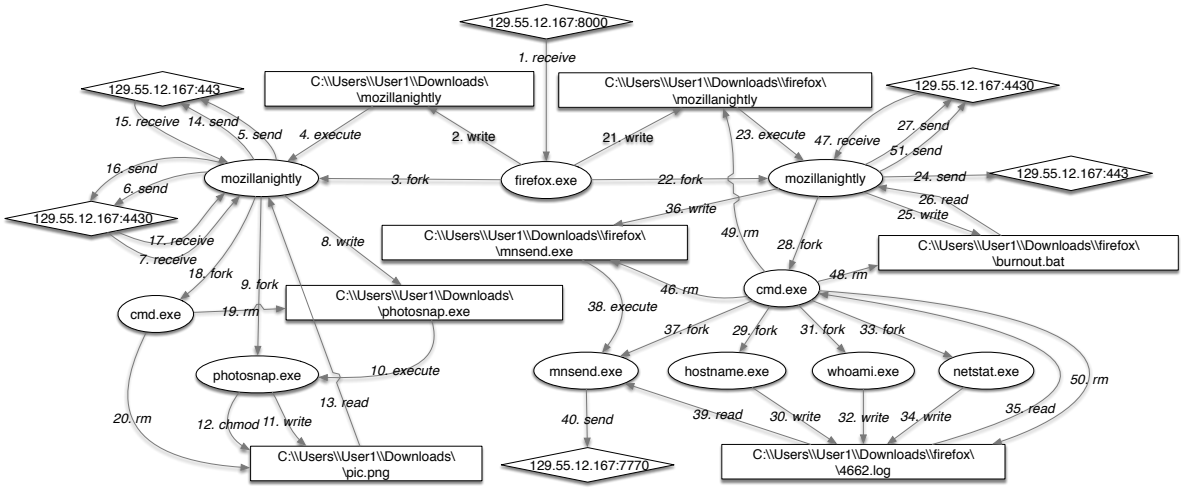

Figure 4.6: Scenario graph reconstructed from campaign W-1.

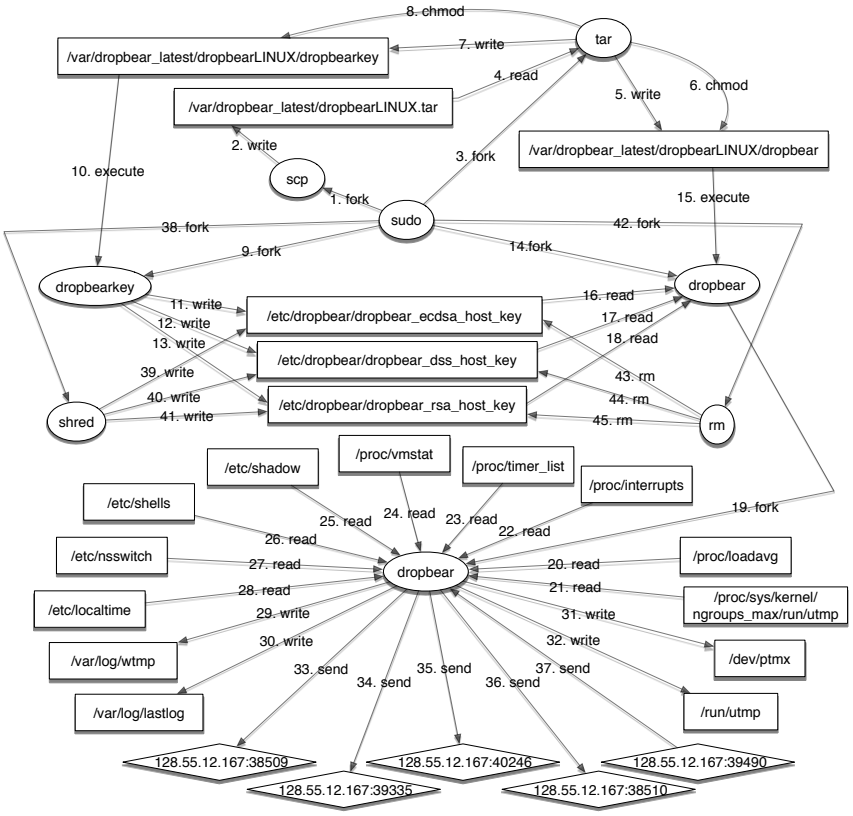

Figure 4.7: Scenario graph reconstructed from campaign L-3.

Chapter 5

Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics

The challenges faced in scenario reconstruction are formidable due to the intermixing of benign and attack activities. This causes forensic analysis to suffers from the *dependence explosion* problem, resulting in a vast number of benign events to be flagged as part of the attack. Existing forensic analysis techniques that rely on coarse-grained provenance are ill-equipped to deal with this explosion. Although SLEUTH's approach of cost-based pruning of forward paths is helpful, it is effective only on fast-moving attacks. For long-running attacks, it produces graphs with numerous benign nodes. In this chapter, we propose two novel techniques, *tag attenuation* and *tag decay,* to mitigate dependence explosion. Our techniques take advantage of common behaviors of benign processes, while providing a conservative treatment of processes and data with suspicious provenance. Our system, called MORSE, is able to construct a compact scenario graph that summarizes attacker activity by sifting through millions of system events in a matter of seconds. Our experimental evaluation carried out using data from two government-agency sponsored red team exercises and demonstrates that our techniques are (a) effective in identifying stealthy attack campaigns, (b) reduce the false alarm rates by more than an order of magnitude, and (c) yield compact scenario graphs that capture the vast majority of the attacks, while leaving out benign background activity.

5.1 Approach Overview and Summary of Contributions

We begin with a motivating example in Section 5.2 that illustrates the challenges in detecting and summarizing stealthy attack campaigns. We then introduce our techniques for mitigating dependence explosion in Section 5.3. Similar as SLEUTH, we associate *tags* to subjects and objects and propagate these tags along the direction of the information play. In MORSE we mainly identify two types of tags:

- *data tags* that capture the integrity and confidentiality aspects of a data item for both subjects and objects, and
- *subject tags* that are associated only with subjects, and indicate our level of suspicion that a particular subject is malicious. Although this tag is similar to code trustworthiness tag mentioned in Chapter 4 but there are some key differences which is discussed in Section 5.3

The core idea behind our approach is to *modulate tag propagation using subject tags.* In particular, our tag propagation rules are lenient on benign subjects, and take advantage of their typical behaviors in order to reduce dependence explosion. At the same time, we use conservative tag propagation rules for suspicious subjects that may be under the direct control of attackers.

We introduce two key concepts, *tag attenuation* and *tag decay* that mitigate dependence explosion through benign processes. Tag decay captures the intuition that a benign subject, if it is subverted and becomes malicious, will do so soon after consuming suspicious input that contains an exploit. For this reason, we allow the data tags of benign subjects to decay gradually and become benign over time, unless they exhibit suspicious behavior. This feature breaks the dependency between suspicious inputs and outputs of a benign subject after a certain threshold of time.

Tag attenuation captures the intuition that objects serve as imperfect intermediaries for propagating malicious behavior through benign subjects. In particular, each such propagation requires the intermediary object to contain an exploit that compromises the subject that consumes it. To capture the difficulty of creating a series of such exploits, we *attenuate* data tags of a benign subject before propagating it to the object that it writes into.

In Section 5.4, we present a policy-based attack detection approach similar to SLEUTH but also takes advantage of tag attenuation and decay to significantly reduce false positives. Our techniques for attack campaign reconstruction are described in Section 5.5.

We use the motivating example from Section 5.2 to illustrate its operation in Section 5.6. Our experimental evaluation (Section 5.7) shows that MORSE is effective in detecting a range of stealthy APT-style campaigns, where some of the critical steps

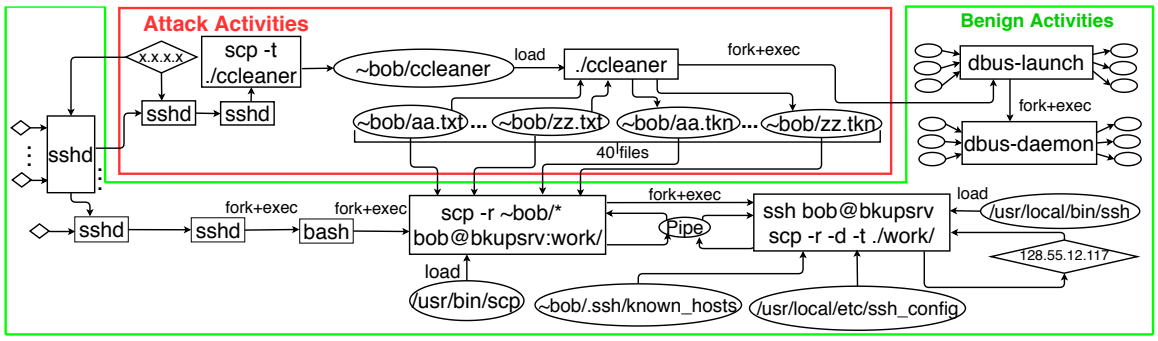

Figure 5.1: Motivating example: CCleaner ransomware. Rectangles denote subjects (processes), while oval-shaped nodes denote files and diamonds denote network objects. Edges denote events, and are oriented in the direction of information flow. Edges without a specific event label denote reads and writes.

are invisible in the data. For instance, our system was able to detect campaigns that relied on:

- previously stolen credentials,
- in-memory (rather than file-based) malware, and
- preexisting malware on the target system, including instances of rootkits, Trojan ssh servers and kernel malware.

Our tag attenuation and decay techniques *decreased false positives by an order of magnitude, while reducing scenario graph sizes by 35x,* all without missing any significant attacker activity. Evasion attacks and lateral movement are also discussed in Section 5.7.

5.2 Motivating Attack Scenario

In this section, we illustrate the problem of *dependency explosion* using an attack scenario from a recent red team engagement that was carried out as part of a research program organized by a government agency. The red team's goal was to organize a highly stealthy ransomware attack, with the following stealth and evasive elements:

- *Stolen credentials.* The red team assumed that the login credentials of the victim user had already been stolen by the attacker. This enabled the attacker to gain access to the victim machine without raising any suspicion.
- *Use of malware matching a benign application.* The red team made use of malware named *ccleaner* that was crafted to evade virus signatures. Note that

ccleaner [156] is a widely used application that analyzes all files and removes unneeded and unwanted files. This behavior blends in perfectly with that of ransomware that reads/encrypts/removes many files.

- *Extensive interaction with benign background activity.* A benign backup task periodically copied over all user files to a backup server, including files created by the attacker's malware. Moreover, malware execution made use of benign supporting processes such as `dbus-launch`.

Fig. 6.1 shows a fragment of the *dependence graph* (also known as *provenance graph*) relating to this attack, constructed from the audit log produced by the Linux `auditd` daemon. Rectangles in this graph are *subjects* (processes), while *objects* are depicted as ovals (files) and diamonds (network connections). Edges in the graph correspond to system events such as `read`, `write`, `load`, `fork`, `execve`, and so on. Edges are oriented in the direction of information flow, and annotated with event names. To reduce clutter, we omit annotations on read and write edges.

Logically, the attack begins with the theft of login credentials for the user Bob, but this step is assumed to have taken place "out-of-band" and is not visible in the audit data. Using these credentials, the attacker Trudy logs into Bob's machine B using ssh. There are numerous logins into B, as shown at the left end of Fig. 6.1. Some of these are by Bob and others may be by Trudy. To reduce clutter, we elide the details of these ssh sessions, except the one corresponding to the attack.

In the ssh session corresponding to the attack, Trudy downloads `ccleaner` malware using `scp`, and then executes it. This malware analyzes the files in Bob's home directory. It selects several files to hold as ransom. Each of these files are replaced with their encrypted versions, which are indicated with a `.tkn` suffix in the figure. When `ccleaner` starts up, it also executes `dbus-launch`, a behavior associated with some of the libraries and toolkits used by it.

In a parallel ssh session, Bob logs into B and initiates a backup of the files in his home directory to a second host A. This activity is shown at the left bottom of Fig. 6.1, and happens to take place immediately after the ransomware attack.

Note that benign activities surrounding the attack far exceed the attack activity. To reduce clutter, we have elided many of these benign activities, including: many ssh sessions for Bob, the details of all the files that were backed up, subsequent activities of `dbus-launch`, and so on. If those details were included, then the picture will be at least 10 times larger.

Challenges This attack poses many challenges for detection and forensic analysis tools. By using stolen credentials, Trudy enters the system without triggering any alarms related to typical break-in activities, e.g., scanning for and exploiting known vulnerabilities, or clicking on email attachments. By using novel malware with a disguised name, Trudy avoids triggering most code signature and file-name based mal-

Event	Tag to update	New tag value for different subject types		
		benign	suspect	suspect environment
create(s, x)	$x.dtag$	$s.dtag$		
read(s, x)	$s.dtag$	$min(s.dtag, x.dtag)$		
write(s, x)	$x.dtag$	$min(s.dtag + a_b, x.dtag)$	$min(s.dtag, x.dtag)$	$min(s.dtag + a_e, x.dtag)$
periodically:	$s.dtag$	$max(s.dtag,\ d_b{*}s.dtag{+}(1-d_b){*}T_{qb})$	no change	$max(s.dtag,\ d_e{*}s.dtag{+}(1-d_e){*}T_{qe})$

Table 5.1: Propagation rules for operations on data. Here, *dtag* refers to the data tag. T_{qb} and T_{qe} stand for quiescent tag values for benign and suspect environment processes, set respectively to $\langle 0.75, 0.75\rangle$ and $\langle 0.45, 0.45\rangle$ in our implementation. Attenuation (a_b and a_e) and decay rate (d_b and d_e) settings are discussed in Section 5.7.

ware detectors. By blending with the behavior of legitimate `ccleaner` program, Trudy can also evade behavior-based and anomaly-based attack detectors. In contrast, we present effective detection techniques using provenance tags, and in particular, a prioritized approach for tag propagation. Our technique is not only accurate on `ccleaner`, but also other stealthy attacks, including those that use in-memory payloads, browser extensions, and in one instance, kernel-resident malware.

The challenges faced in scenario reconstruction become extremely formidable, given all the intermixing with benign activities in this attack. SLEUTH's approach of cost-based pruning of forward paths is somewhat helpful, but the resulting graph still contains over 3000 nodes. This is at least an order of magnitude larger than what can be visualized and understood by an analyst. In contrast, we present a tag prioritization method that generates a far smaller and compact attack graph that only contains around 40 nodes.

5.3 Tags and Propagation

Provenance graphs faithfully capture all *possible* dependencies, and hence do nothing to address dependence explosion. The core of our approach is to develop *a system of tags* and *propagation rules* that *prioritize a subset of dependencies* for attack investigation. Our prioritization takes advantage of behaviors common to benign applications in order to prune away dependency chains unlikely to play a role in attacks. At the same time, it is conservative (i.e., assumes the worst-case behavior) in its reasoning about malicious subjects, thus making it evasion-resistant. Key to this approach is our method for tagging subjects as benign or malicious, a topic covered in Table 5.2, Sections 5.5.2 and 5.6. Here, we begin by defining the *subject tags* used to differentiate these groups:

- *suspicious code:* This value indicates that the subject's code is suspect, i.e., it could be malware.

- *suspicious environment:* This value, abbreviated as *susp_env*, indicates that the subject's code is benign, but its execution was started by a suspicious subject, which controlled the command-line parameters and environment variables.

- *benign:* This tag value indicates that both the code and running environment of a subject are benign. Benign subjects may contain exploitable vulnerabilities, so they may be compromised by malicious inputs.

- *trusted:* This tag indicates that the subject is capable of protecting itself from malicious inputs.

Unlike subject tags that are associated only with subjects, *data tags* are associated with objects as well as subjects, as both contain stored data. A data tag is a tuple $\langle c, i \rangle$, where:

- c is the *confidentiality tag* that captures data sensitivity, and

- i the *integrity tag* that captures data trustworthiness.

Highly confidential data needs protection from unauthorized disclosure. Logically, we distinguish between *high* and *low* values for confidentiality. However, since it is easier to express tag propagation rules as real-valued functions, we use real values for data tags. Note that by convention, *lower* numerical values correspond to *higher* levels of confidentiality. Thus, secret data such as private keys and password files should be assigned a confidentiality tag of zero, while public information should be assigned a tag of 1.0. Values in the range $[0.0, 0.5)$ are considered *high* confidentiality, while the range $[0.5, 1.0]$ corresponds to *low* confidentiality.

High integrity data is safe to consume, i.e., *it won't compromise the subject, or otherwise enable an attacker to control* its behavior. In contrast, low integrity data may compromise a subject that executes it. For this reason, we refer to high integrity data (specifically, the range $[0.5, 1.0]$) as *benign* and low integrity data (specifically, the range $[0.0, 0.5)$) as *suspicious.* Note the convention that *higher* numerical values correspond to *higher* levels of integrity. Thus, highly trusted data is given an integrity tag of 1.0, while highly suspicious data will have an integrity tag close to zero. In some contexts, it is helpful to define *suspiciousness tag*, which is obtained by subtracting the integrity tag value from 1.0.

The flow of *data tags* within the dependence graph is *modulated* by subject tags in our framework. To express these modulation rules concisely, we extend standard arithmetic operations to data tags as follows, where *op* is one of $+, -, *$ or / operators. Operations such as *min* and *average* can be extended similarly.

$$\begin{aligned} \langle c_1,\ i_1 \rangle\ op\ k &= \langle c_1\ op\ k,\ i_1\ op\ k \rangle \\ \langle c_1,\ i_1 \rangle\ op\ \langle c_2, i_2 \rangle &= \langle c_1\ op\ c_2,\ i_1\ op\ i_2 \rangle \end{aligned}$$

Event	Tag to update	New tag value for different subject types		
		benign	suspect	suspect environment
load(s, x)	$s.stag$	$min(s.stag, x.itag)$		
	$s.dtag$	$min(s.dtag, x.dtag)$		
exec(s, x)	$s.stag$	$x.itag$	$min(x.itag, susp\_env)$	$x.itag$
	$s.dtag$	$\langle 1.0, 1.0 \rangle$	$min(s.dtag, x.dtag)$	$min(s.dtag, x.dtag)$
inject(s, s')	$s'.stag$	$min(s'.stag, s.itag)$		
	$s'.dtag$	$min(s.dtag, s'.dtag)$		

Table 5.2: Propagation rules for code operations. Here, *stag* and *dtag* denote subject and data tags. The integrity component of *dtag* is referenced using *itag*.

Tag Propagation Rules

Events cause data tags to propagate in the direction of information flow. Unchecked propagation leads to a dependence explosion, so our core idea is to use *subject tags* to *modulate* data tags flowing through a subject. The guiding principles behind our design are:

- tag propagation should be *conservative for suspect subjects,* but can be *lenient for benign subjects.*
- tag propagation should *prioritize data flows that an attacker can control,* while de-emphasizing other data flows.
- only benign subjects can have benign data integrity; for other subjects, data integrity is forced to be *low,* say, 0.45.[1]

Tables 5.1 and 5.2 consider the main operations that propagate tags. Note that `fork` implicitly copies the parent's tags to the child. Other system calls such as `chmod`, `unlink`, and `mprotect` are security-relevant but do not change provenance. As a result, we are left with just the operations listed in the first column of Tables 5.1 and 5.2. These operations typically take two arguments s and x that represent the subject performing the operation and the object being operated on.

The second column in the table identifies the tag that will need to be updated as a result of the operation in the first column. The next three columns specify, respectively, the new tag values of this tag for benign, suspicious and suspect environment processes.

[1]Suspect subjects may be malicious and hence can generate low-integrity output even if they only consume benign input. Suspect environment subjects are spawned by suspect subjects, so they may already hold low integrity data in their memory (as command-line arguments, environment variables, etc.) and can output this data even before consuming input from low-integrity objects.

Propagation Rules for Operations on Data

The first row in Table 5.1 corresponds to object creation. The object simply inherits the subject's data tag in all cases. Note, however that an empty file contains no confidential or malicious content. Hence, for benign subjects, we delay this propagation of subject's tag until the first write operation. We avoid this lenient treatment for suspicious and suspect environment subjects, so that objects created by them will have low integrity from the very beginning.

The second row concerns a read operation. Note that if a process reads highly confidential (or low integrity) data, this immediately leads to the process memory holding highly confidential (or low integrity) data. For this reason, we update the subject's data tag to be the minimum of its current value and the tag of the data just read.

The next row concerns the write operation, which propagates the subject's tag to the object being written. For suspicious processes, this propagation is immediate, i.e., we assume that (a) the most confidential data within process memory may be output at this point, and (b) lowest integrity data within the process memory may be written. This conservative treatment ensures that all outputs of a malicious process will be treated with suspicion.

Tag attenuation for benign subjects. Note that even if a benign subject previously read highly confidential (or low integrity) data, an attacker cannot control whether a write operation will output such data. To factor this, we *attenuate* the confidentiality and integrity tags of a benign subject before propagating them to the object. Recall that smaller confidentiality (or suspiciousness) corresponds to larger tag value, so we can achieve attenuation by multiplying by a factor $f > 1$. However, a multiplicative factor will have no effect if the original tag value is zero. So, we prefer an additive factor. We use different additive factors a_b and a_e for benign and suspect environment subjects. Since an attacker is likely to have more control over suspect environment subjects, $a_e < a_b$.

For updating the data tags of objects being written, we take the *min* operation, so that the object's tag indicates the most confidential (and the lowest integrity) data contained within.

Tag decay for benign subjects. If a benign process is compromised by suspicious input, then this compromise will happen soon after input consumption. Otherwise, it is likely that the input, even though it was deemed suspicious at first, is really benign. So capture this intuition, we gradually lift the integrity tag to its *quiescent* value by applying a *decay* operator. Decay is *not* applied to higher tag values, thus leaving them untouched.

Tag decay is meaningful for confidentiality as well. Long-running benign applications that use highly sensitive data, e.g., passwords or keys, are designed to use them quickly, and then erase them from memory, or at least prevent them from being emitted in their output. For simplicity, we have used the same decay rate and quiescent

Name	Description	Operation(s)	Data integrity condition	Other conditions
$MemExec$	Prepare binary code for execution	$mmap(s,p), mprotect(s,p)$	$s.itag < 0.5$	$incl\_exec(p)$
$FileExec$	Execute file-based malware	$exec(s,o), load(s,o)$	$o.itag < 0.5$	$s.stag$ is benign
$Inject$	Process injection	$inject(s,s')$	$s.itag < 0.5$	$s'.stag$ is benign
$ChPerm$	Prepare malware file for execution	$chmod(s,o,p)$	$o.itag < 0.5$	$incl\_exec(p)$
$Corrupt$	Corrupt files	$write(s,o), mv(s,o), rm(s,o)$	$s.itag < 0.5 \leq o.itag$	
$Escalate$	Privilege escalation	$any(s)$	$s.itag < 0.5$	changed userid
$DataLeak$	Confidential data leak	$write(s,o)$	$s.itag < 0.5$	$s.ctag < 0.5 \leq o.ctag,\ socket(o)$

Table 5.3: Provenance-based policies for attack detection. Here, $socket(o)$ holds when o refers to a socket, while $incl\_exec(p)$ holds if p includes the execute permission.

value for both confidentiality and integrity tags.

As is common in modeling decays, we have used an exponential decay function. If the decay operation is applied once for each period t, then a tag with an initial value $v_0 < T_{qb}$ will change to v_n after n periods, as given by the following equation. Since $d_b < 1.0$, v_n converges to T_{qb} for large n.

$$v_n = v_0 * d_b^n + (1 - d_b^n) * T_{qb}$$

This rationale for decay does not apply to suspicious processes, so no decay operator is applied to them. For benign processes running within a suspect environment, a decay operator can be applied, but the rate parameter d_e should be larger than d_b, reflecting a greater level of skepticism about their behavior in comparison with benign processes. For the same reason, T_{qe} should be smaller than T_{qb}. In our implementation, we have used $T_{qe} = \langle 0.45, 0.45 \rangle$.

Propagation Rules for Operations on Code

Table 5.2 specifies propagation rules for code-related operations. In general, loading causes the integrity tag of loaded object to propagate to the subject. (This is the primary means of determining subject tags, a topic further discussed in Section 5.6.) For this propagation, we treat data integrity in the range of [0.5.1.0] as *benign*, while the range $[0.0, 0.5)$ is treated as *suspicious.* In addition, recall that the maximum data integrity of a subject is bounded by its subject tag. For this reason, all operations that load code into a subject s propagate the data integrity of the code object to the data integrity of s. In particular, we take the *min* of the data integrity of s and the code object.

Consider the `load` operation that is typically used to load a library into a subject's memory. When a benign process loads an object, its subject tag is downgraded to *suspicious* if the object has a low integrity tag; otherwise, the subject tag is left unchanged. This behavior is captured by the *min* operation used to update the subject tag of benign subjects on a `load` operation. The same logic applies to suspicious as well as suspect environment subjects.

Although `exec` is similar to load in terms of loading new code for execution, there are several important differences as well. In particular, `exec` causes the code memory to be cleared, so we simply overwrite the subject tag for benign code with the integrity of the new code. Moreover, since `exec` causes data memory to be cleared, we set the data tag to $\langle 1.0, 1.0 \rangle$ to indicate the absence of confidential data, and to reset its data integrity tag to be high. (Recall the condition that data integrity tags can never exceed the subject tag, so, the value of the integrity tag will automatically be reduced to that of the object just loaded.)

The above logic for updating subject tag on `exec` operations applies to subjects with a suspect environment as well. In addition, we no longer consider the process to be running in a suspect environment since the process performing the `exec` isn't *suspicious*. But we do not reset the subject's data tags, as our level of trust on these processes are strictly less than that of benign subjects.

For `exec`'s by suspicious processes, the above argument for replacing their subject tag with that of the executable continues to hold. However, note that since the process is starting out to be suspicious, the process after `exec` must be considered, at a minimum, to be in a suspect environment, and hence we take a *min* with *susp_env*. For data tag value, we apply the *min* operator as in the case of `load`.

Finally, we turn our attention to the `inject` operation, which loosely corresponds to one subject modifying the code of another. There may be no single system event that corresponds to `inject`, so it may be necessary to piece together a set of related operations. For instance, on Windows, an `inject` may correspond to a combination of operations made by a process s to open the memory of another process s', write to it, and then create a remote thread. On Linux, it may correspond to a combination of `ptrace` system call made by s to attach to another process s', followed by operations to modify the memory of s'. Regardless of when an `inject` is recognized, its behavior is similar to code loading. So, the rules for updating the tag are similar to those for the `load` operation.

5.4 Provenance-Based Attack Detection

Our key contribution here is to show that naive tag propagation can lead to a large number of false positives, while our tag prioritization achieves a dramatic reduction in this number. Secondly, our policies are more refined, enabling them to detect stealthy attacks based on in-memory malware. According to a recent report [6], a majority of threat actors (57%) avoided file-resident malware in 2018, choosing to go with in-memory malware, as it can evade most existing threat detectors (which are based on the presence or execution of file-resident malware). As further evidence of novelty in these policies, our approach was able to detect attacks that made use of preexisting rootkits and kernel-resident malware.

Table 5.3 summarizes the attack detection policies used in our system. These policies have the same general structure: they all concern a system call (e.g., writing

an object), with conditions imposed on (a) the data integrity tags of the subject and/or objects involved, and (b) other information associated with the call, such as permissions and userids. The policies in Table 5.3 abstract some of the essential steps of APT attacks [2, 8, 105], including the initial exploit, foothold establishment, privilege escalation, and exfiltration of sensitive data.

The first row of Table 5.3 aims to capture the execution of in-memory malware. This may either represent a memory corruption exploit used in the initial exploit stage, or an advanced in-memory payload used for attacker's foothold establishment or expansion. In order to trigger this policy, a subject's data must have suspicious provenance (signified by an integrity tag less than 0.5), and some of this data should be readied for execution, which requires the use of `mmap` or `mprotect` system calls with execute permission enabled. (Note that `mmap` and `mprotect` also occur during library loading operations. Our system maps these operations into a `load`, thus preventing this policy from being triggered by file loads.)

The second row is aimed at file-based malware execution. It is triggered by the load or execution of a file with suspicious provenance. The third row is similar, except that instead of a subject voluntarily loading suspicious code, malware is injected into its address space by another subject.

The fourth row detects a step in preparing file-based malware for execution by making the file executable. It requires the object's data to have suspicious provenance.

The fifth row detects overwriting of important system files (or registry entries), a step that is typically used to establish a (more permanent) foothold on a host. It is triggered by an attempt by a subject with suspicious provenance to overwrite a higher integrity object.

The sixth row recognizes a privilege escalation attack. This policy is triggered by any system call by a subject with suspicious provenance, provided the userid before and after the call are different.

Finally, the last row captures data exfiltration: an alarm is triggered when a subject with suspicious provenance writes sensitive data to a network socket that is not authorized for confidential data.

5.5 Attack Scenario Reconstruction

The central goal is to connect various attack steps to provide a high-level summary of an ongoing attack campaign. To achieve this, we first develop a dependence-based analysis to identify the initial attack step, also called the entry point. We then perform a tag-based forward dependency analysis to construct a graph that summarizes the campaign. We describe these two steps below.

5.5.1 Entry Point Identification

Attack campaigns consist of many steps. Some of these steps lead to numerous alerts, e.g., file corruption and data leak policies can easily raise thousands of alerts. It is infeasible for an analyst to track down each alert individually, so we have developed an alert aggregation and prioritization technique further described below.

Given an alarm, we first associate it with a subject or object originating it. For alarms raised on an input event, we consider the object to be the originating node. For all other events, the subject is considered the originator. We also assume that each alarm has an associated *weight*, which is a real number between 0 and 1 that reflects our confidence level in the alarm.

Given an alarm originating at node n, we perform a backward search in the dependence graph for the closest node n' that also triggered an alarm. If we don't find such an n', then we call this a *primary* alarm, and set $precursor(n)$ to $null$, and $weight(n)$ to be the weight of the alarm. Otherwise, the new alarm is classified as *secondary*; we set $precursor(n)$ to $precursor(n')$, and add the weight of the alarm to the weight of $precursor(n')$. Note that primary alarms have the combined weight of all the alarms ever raised.

For simplicity, our implementation follows only subject to subject edges while searching for n', and ignores edges between subjects and objects. (For alarms originating on objects, the first hop uses an object-to-subject edge, but the rest are subject-to-subject edges.)

An analyst can now pick the top few primary alarms with the highest weight, and investigate them further. We designate the least common ancestor of the selected primary alarm nodes as the entry point. In cases where the top-ranked primary alarms have a much higher weight than the rest, this entry point discovery does not require human assistance, and can be fully automated.

5.5.2 Forward Analysis

If the entry point or any of the primary alarm nodes are processes with *benign* subject tags, then their subject tag is modified to *suspicious*. Tag propagation rules are rerun on these processes, as well any descendants whose tag has changed as a result of this.

Next, a depth-first search of the dependence graph is initiated at the entry point node. This search does not visit nodes whose data integrity tag is above a set threshold (which defaults to 0.5, but may be changed by the analyst). This search identifies the nodes that will be included in the scenario graph. Next, we add all the edges incident on these nodes. We then add all the nodes attached to these newly added edges. As a final step, we combine multiple edges between two nodes if they have the same name, e.g., multiple reads.

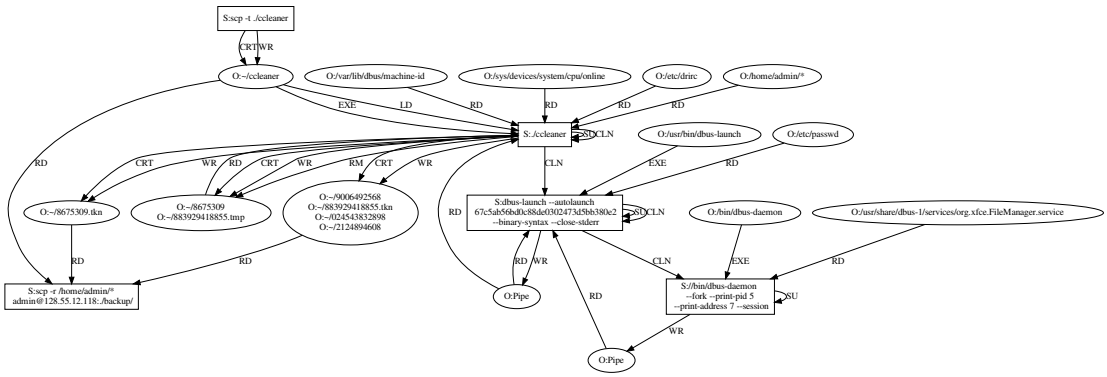

Figure 5.2: Scenario Graph constructed by MORSE for CCleaner Ransomware

5.6 Putting it All Together: Analysis of CCleaner

We now illustrate how the techniques described so far come together to analyze the `ccleaner` attack from Section 5.2. The resulting graph, as seen by the analyst, is shown in Fig. 5.2. Note that the graph generation is fully automated, and involves no manual post-processing.

Data Tag Initialization. Newly created objects and subjects inherit their tags from the subjects that create them, as described in Tables 5.1 and 5.2. But we need a separate mechanism for assigning tags to pre-existing entities such as (a) processes and files existing before the start of data collection, and (b) network endpoints.

Tag initialization can be based on an organization's host configuration practices and policies. Alternatively, they may be learned by observing the use of files during a training period. Neither of these options were available to us in our experiments. The dataset we used did not come with any documentation of host configuration practices. Moreover, although some training data was included, the behavior observed on the days of attacks differed significantly from the training data, thus ruling out the training option as well. For this reason, we relied on the following minimalist approach in our evaluation: we designated `/etc/passwd`, `/etc/shadow` and the `/var/log/` directory as confidential. All files originally present on the system were assigned high integrity. Finally, network addresses were assigned low integrity and confidentiality. This tag initialization is consistent with our threat model (Section 5.7) and sufficient for our evaluation. Our tag initialization code, used in the analysis of all the attacks in our evaluation, consists of 14 lines in E^*.

Subject Tag Initialization. Similar to our treatment of pre-existing files, all processes that were running at the start of data collection (e.g., servers such as `sshd`) were marked benign.

Subject tags of benign processes change to *suspicious* if they exhibit suspect behavior, e.g., loading or executing low-integrity code, or being injected by a lower integrity subject (Table 5.2). Additionally, when a number of alarms can be traced back to a subject, that subject is marked suspicious (Section 5.5.1).

Attack Detection. Note that the initial login by Trudy does not trigger any alarms. She is using stolen credentials, but our system has no information about this theft. Her IP address is unremarkable as well. When she downloads `ccleaner`, it is given a low integrity since it is being downloaded from an unknown internet site. When this file is executed, it triggers the *FileExec* alarm from Table 5.3. The `ccleaner` process is also marked as a suspect subject by the *exec* rule in Table 5.2. As a result, its file overwrite (or remove) operations trigger the *Corrupt* alarm as well.

While the policies shown in Table 5.3 have been sufficient in our experimental evaluation, note that additional attack detectors can easily be incorporated in our system, and used to (a) identify and tag suspicious subjects, and (b) trigger scenario graph generation.

Entry Point Identification. The entry-point identification technique described in Section 5.5 traces back the above *FileExec* and *Corrupt* alarms to Trudy's `scp` process. It is now given a subject tag of *suspicious,* and the tag propagation rules are rerun.

Forward Analysis. Since the `scp` and `ccleaner` processes have suspicious subject tags, no tag attenuation or decay is applicable to them. Hence, every file written by these subjects is assigned a low integrity tag, and their child processes continue to be suspicious.

When `ccleaner`'s child executes `dbus-launch`, a benign file, it is marked as suspect environment, as per the *execve* rule in Table 5.2 (middle column). As a suspect environment process, when it executes another benign file, `dbus-daemon`, this *execve* rule (see the right-most column) causes it to be marked benign. Note that `dbus-daemon` still has low data integrity, but due to attenuation and decay, its child processes end up having benign subject and data tags.

Recall that our forward analysis starts at the entry point node and traverses forward through all nodes (objects or subjects) with data integrity ≤ 0.5. The resulting graph is shown in Fig. 5.2.

Refinement and Rerun. Analysts can refine and rerun this analysis in order to convince themselves that some components of the attack haven't been missed. Since our forward analysis typically takes a small fraction of a second, analysts can explore refinements rapidly.

Some of the possible refinement actions include: (a) marking additional subjects as suspicious, (b) trying alternative attenuation and decay values, (c) changing the tag value threshold for including a node in the scenario graph, or (d) extending the graph forward at selected nodes. For this attack, there were no obvious candidates for (a). We tried (b) through (d), but found no more malicious activity.

5.7 Experimental Evaluation

Platform. The system under attack consisted of multiple hosts running recent

Data-set	Duration (hh:mm)	# of events	Short attack name	Attack name used in ground truth and short description of attack
L-3	263:05	714 M	Firefox backdoor	*Firefox backdoor w/ Drakon in-memory:* Firefox is exploited by a malicious web site to execute an in-memory payload. This provides a remote console for the attacker (Fig. 5.6).
			Browser extension	*Browser extension w/ Drakon dropper:* Exploit the victim system using a preexisting malicious Firefox browser extension, drop and execute a malicious file on disk (Fig. 5.11).
			Executable attachment	*Phishing e-mail w/ executable attachment:* A malicious executable file was sent as an email-attachment, which, after opening, established a connection to the attacker's machine.
F-3	263:28	21 M	Malicious HTTP request	*Nginx backdoor w/ Drakon in-memory (4 instances):* Attacker exploits Nginx server using a malicious HTTP request. Nginx then downloads and executes several malicious files (Fig. 5.7).
L-4	15:28	36.5 M	User-level rootkit	*Azazel:* Using a preexisting user-level rootkit, the attacker connected to the system using a remote shell and ran reconnaissance commands. (Fig. 5.12)
			CCleaner ransomware	*VNC attack:* Motivating example discussed in Section 5.2 (Fig. 5.2).
			Recon w/ Metasploit	*Metasploit:* Malware was downloaded and executed using Metasploit, giving the attacker remote access. Attacker ran various reconnaissance commands using this capability (Fig. 5.8).
			Kernel malware	*Firefox Drakon:* In-memory exploit works with a preexisting malicious kernel module for privilege escalation. This allowed the attacker to compromise the *sshd* server (Fig. 5.9).
F-4	11:53	37.2 M	Dropbear Trojan	*Dropbear SSH:* Using a pre-installed Trojan ssh server, the attacker logged into the victim, ran multiple reconnaissance commands and exfiltrated the results.
			Recon w/ Rootkit	*Micro APT:* The attacker uploaded two rootkits using *scp* to the target systems separately, executed them, gained root privilege and ran multiple recon commands (Fig. 5.10).

Table 5.4: Attacks contained in our datasets. L-3 and F-3 are from the 3^{rd} DARPA TC red team engagement, while L-4 and F-4 are from the 4^{th} engagement.

versions of Ubuntu Linux and FreeBSD. Our analysis was performed on an Ubuntu 18.04 Linux laptop with an Intel 2.7GHz i7-7500U CPU and 16GB memory.

Threat Model. Similar to previous research on attack reconstruction from audit logs [60, 61, 72, 92, 105], we assume that attackers cannot compromise audit record collection or the log itself. Best results are obtained if (a) victim systems start off in a benign state, i.e., without any pre-existing malicious software, and (b) all security-relevant system calls and arguments are included in the audit log. However, real-world systems may not always satisfy these conditions. Indeed, several of the attacks in our dataset relied on pre-existing malware. The logs were also incomplete due to missing system-call arguments and/or provenance in some cases. Despite these factors, MORSE was able to produce very good results.

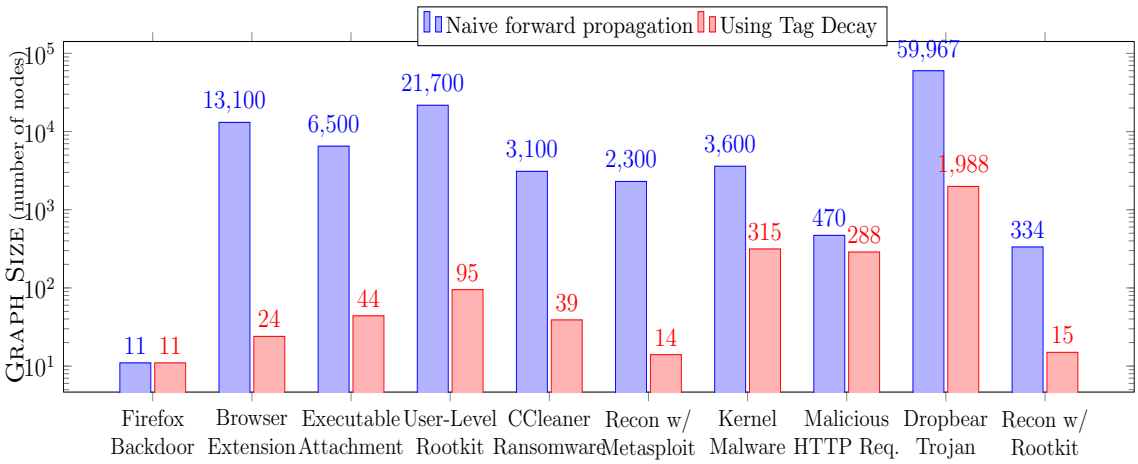

Figure 5.3: Reduction in scenario graph size achieved using tag attenuation and decay. The average size reduction is 35 times, and no relevant nodes were dropped.

5.7.1 Dataset

Many previous works [60, 72, 86, 92, 98] have based their evaluation on attack datasets created by the authors themselves. This choice is not optimal, as it can introduce a bias in attack selection that favors the authors. Yet, it is unavoidable in the absence of third-party datasets. We have therefore chosen to evaluate our system using attacks carried out by an independent red team, as part of the DARPA Transparent Computing (TC) program.

DARPA organized five red team engagements between 2016 and 2019. The scale and sophistication of these engagements increased significantly after the first two engagements, so we focused our evaluation on the third and fourth engagements. (The fifth engagement had not taken place by the time this work was carried out in early 2019.) Note that the third engagement data has already been publicly released [3] by DARPA, while the rest may be available on request.

In its choice of attacks, the red team was guided by what they considered were emerging stealthy APT techniques. But they were less concerned about data completeness. For instance, audit data collection typically began long after many background services had been started. As a result, they were unable to track provenance through such services. Moreover, some of the red team attacks relied on rootkits or malicious kernel modules that had been present on the victim system prior to audit data collection. We believe that similar gaps are unavoidable in real-world systems, and hence the red team data enables a realistic evaluation that wouldn't have been possible, had we created the data on our own.

Dataset	FileExec		MemExec		ChPerm		Corrupt		CDL		Escalate		**Total Alarms**	
	Base	Ours	Base	Ours	Base	Ours	Base	Ours	Base	Ours	Base	Ours	Base	Ours
L-3	479	1.31x	1.45M	13.96x	9	1.41x	184K	10.53x	13.4K	40.36x	959	1.54x	1.65M	11.54x
L-4	53	18.33x	337K	16.73x	66	22.45x	32K	13.68x	1.88K	15.95x	211	1.92x	371K	16.45x
F-3	19	1x	N/A	N/A	1.81K	1.86x	6.4K	1.91x	41.03K	94.19x	113	21.98x	49.4K	11.32x
F-4	38	9.5x	N/A	N/A	1.82K	2.65x	166K	16.85x	53.90K	4.84x	243	4.52x	222K	7.85x
Average		**3.89x**		**15.28x**		**3.53x**		**8.25x**		**23.28x**		**4.14x**		**11.40x**

Table 5.5: Alarm reduction due to tag attenuation and decay, with $a_b = 0.2, d_b = 0.25, a_e = 0.1, d_e = 0.5$. The last two columns show the reduction across all alarm types, while the others break it down by alarm type. "Base" columns show the alarms generated by SLEUTH, while "Ours" show the reduction achieved by MORSE.

Data from DARPA TC Engagement 3

In our evaluation, we used the datasets from the TRACE and CADETS teams in the DARPA TC program [3]. TRACE data, henceforth called L-3 dataset, is derived from Linux audit data. CADETS data, called F-3 dataset, is derived from FreeBSD DTrace [4] data. More details about these datasets is shown in Table 5.4.

According to the ground truth provided, there were four attacks that (mostly) succeeded in L-3, plus several failed attempts. There were five attacks in F-3, of which four were repetitions of the same attack. The last attack, which also appeared in L-3, is a web-site password stealing attack that lures the user to a phishing web site. There are no subsequent effects on the victim's machine or network. As a result, this attack is not visible in the system-call audit data, which just shows the user visiting a web site — a perfectly normal activity. So we have omitted this attack from our analysis, and show only the remaining attacks in Table 5.4.

Data from DARPA TC Engagement 4

The L-4 and F-4 datasets shown in Table 5.4 are from the 4^{th} red team engagements involving a pair of Ubuntu Linux and a pair of FreeBSD systems that interact with each other. While the attacks themselves were more stealthy than Engagement 3, and involved attacks that spanned multiple hosts, the adversarial team chose to work in a serial fashion, focusing on just a single operating system on each day of the engagement. As a result, the datasets were shorter.

5.7.2 Effectiveness of Tag Attenuation and Decay

Parameter Selection. Our method is characterized by the rates of attenuation and decay for benign and suspect environment subjects. Values of these four parameters (a_b, d_b, a_e and d_e) can be chosen based on a high-level understanding of how they affect alarms. For instance, consider a benign subject s_1 reads a file f_1 with integrity 0.0 and writes to file f_2, which is then read by another benign subject s_2 that then writes to f_3, which, in turn, is read by a benign s_3 that then writes to f_4. If we set $a_b = 0.2$,

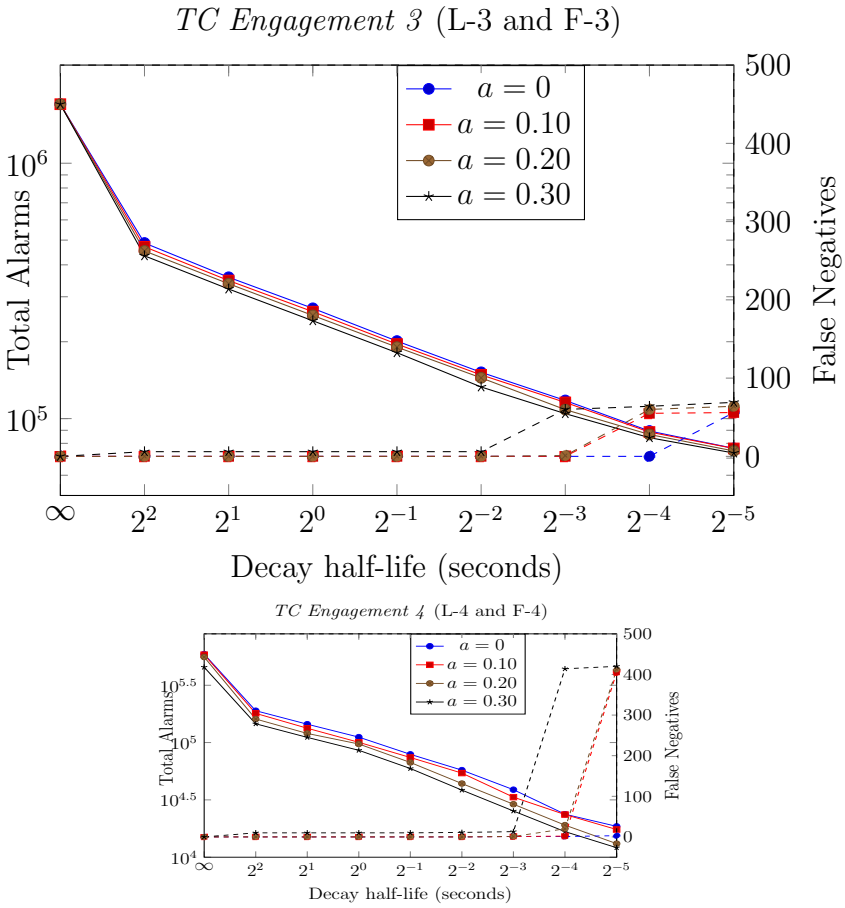

Figure 5.4: Total number of alarms and false negatives on *TC Engagement 3* and *Engagement 4* datasets using different attenuation and decay rates. The scale for total number of alarms is on the left, while the false negative scale is on the right. The total number of true positives are 126 and 425. The total number of alarms without attenuation and decay are 1.69 million and 0.59 million respectively, and they reduce by 10x with tag attenuation and decay.

then it is easy to see that f_2 and f_3 will have low integrity (specifically, integrity of 0.2 and 0.4 respectively), but f_4 will have a high integrity. In other words, this choice of a_b limits low integrity data from propagating beyond two subject-to-object hops. This seems like a sensible choice: it is extremely unlikely that one can craft malicious data f_1 that will first exploit a vulnerability in s_1 to compromise it, and cause it to produce another malicious file f_2, which, in turn, exploits a vulnerability in the second benign subject s_2, causing it to produce another malicious file f_3 that in turn contains an exploit for s_3. For suspect environment subjects, we set $a_e = 0.1$ to reflect the fact that attackers have more control over suspect environment subjects.

We can use a similar process for choosing the decay rate parameter d_b. When a benign subject consumes malicious input, it usually takes a very short time for the exploit to succeed or fail, say, 50 to 200 milliseconds. Accordingly, we could set the half-life of d_b to be a slightly above this threshold, at 0.25 seconds. Note that in this context, half-life is the duration in which the difference between the current data tag and its quiescent value will be halved. For instance, if a benign subject starts with an integrity of 0.15, in 0.25 seconds its integrity will increase to 0.45. (Recall that we use 0.75 as the quiescent data tag value for benign subjects.) For suspect environment subjects, we use double this value, i.e., $d_e = 0.5$ seconds.

We validate the above analysis-driven selection of decay and attenuation parameters using three sets of experiments below.

Scenario Graph Size Reduction

Figure 5.3 summarizes the reduction in scenario graph sizes achieved using tag attenuation and decay. These graphs were generated as described in Section 5.5.2: starting from the primary alarm, and retaining only nodes with data integrity below 0.5. The geometric mean of the reduction achieved across all the attacks in our dataset is about 35. No relevant nodes were missed.

Note that in some cases, the resulting graphs are still large, especially in the case of Dropbear, with about 2K nodes. This is because Dropbear is an SSH server that continues to be used for the duration of the dataset, and any of its actions during this period could actually be malicious. However, in realistic settings, the analyst would want to construct the scenario graph soon after an alarm is triggered. We observed that if the graph is generated within 10 minutes of the attack, our approach would indeed generate a compact graph consisting of just 20 nodes.

SLEUTH [61], our previous work, also achieves alarm reduction using two subject tags, called code- and data-trustworthiness tags. By triggering only on code-trustworthiness, it reduced false alarms by two orders of magnitude on TC Engagement 1 dataset. *However, this strategy causes it to miss half the attacks in Engagement 3 and 4,* including the Firefox backdoor, user-level rootkit, kernel malware, dropbear, and some of the malicious HTTP requests.

Alarm Reduction

To properly evaluate our approach of alarm reduction, we calculated the alarm reduction achieved on an hourly basis, and computed its geometric mean. This was done individually for each alarm type, as well as the total number of alarms. These results are shown in Table 5.5. Across all datasets and all alarm types, our approach achieved an average of 11.4x reduction in the number of alarms.

Note that MORSE's *FileExec*, *MemExec*, *ChPerm*, *Corrupt* and *DataLeak* policies match those of SLEUTH but for the use of tag attenuation and decay. Consequently, SLEUTH's alarm counts correspond to the "Base" column in Table 5.5. *Thus,* MORSE *generates* an order of magnitude *fewer alarms than* SLEUTH.

False Negatives

High values for tag attenuation and/or decay can lead to false negatives. To assess this potential, we plot the total alarm numbers and the false negatives (based on the ground truth) in Fig. 5.4. Alarm number curves are sloping down, with the y-scale shown on the left of each chart. False negative curves slope upward, and their scale is shown on the right side of the chart.

From the charts, it is clear that false negatives are generally absent at attenuation rates of 0.2 or lower, but they increase afterwards. At rates above 0.25, if low integrity data from the internet is written to a file after passing through a pipe, the file will have high data integrity (i.e., ≥ 0.5). This behavior, seen with some services such as `ssh`, causes attacks to be missed. These results support our initial choice of 0.2 for attenuation rate. If additional margin of safety is desired, it can be reduced to 0.1. While this increases alarms, we found that the scenario graph sizes are unchanged from Fig. 5.3.

At our chosen attenuation rate, false negatives due to decay don't increase significantly until we reach decay rates at least 4x faster than the 250ms we suggested earlier. These results hold for both datasets we have used in our evaluation. Although not shown here due to space limitations, this observation holds even if we separate the datasets further based on the OS.

Summary of Effectiveness

For the attenuation and decay rate selected at the beginning of this section, we achieve an 11.4x reduction in alarms without experiencing false negatives. We also achieve a 35x reduction in scenario graph size without false negatives. The decay rate could be increased by a further 4x before experiencing false negatives, while the attenuation rate could be decreased by 2x without changing scenario graph sizes, thus providing significant margins of safety.

Data set	Size on disk (GB)	Number of attacks	Graph generation time/attack (sec.)
L-3	23.79	3	0.043
L-4	2.27	4	0.053
F-3	1.18	4	0.030
F-4	1.26	2	0.220

Table 5.6: Runtime for scenario graph generation.

Data set	Total events	File size on disk (GB)	Memory Usage (GB)
L-3	714 M	23.79	0.49
L-4	36.5 M	2.27	0.11
F-3	21 M	1.18	0.19
F-4	37.2 M	1.26	0.11
Total	**808.7 M**	**28.5**	**0.90**

Table 5.7: Main memory size of dependence graphs.

5.7.3 Runtime Performance

Table 5.6 shows performance related to scenario graph reconstruction. The second column shows on-disk sizes of data sets in *compressed* Apache Avro binary format. The third column shows the number of attacks in each dataset, while the fourth shows the average time to generate the scenario graphs for these attacks. Even though the data set sizes range from a few to tens of GBs, scenario graph generation is very fast, taking on average ***69*** milliseconds per attack across the 13 attack instances in our dataset. The principal source of this speed is the compact in-memory dependence graph representation used in our implementation. Specifically, we have developed (a) a versioned graph representation that is acyclic, and (b) a notion of *full dependence preservation* [63] that eliminates the need to store the vast majority of events, while guaranteeing accurate forensic analysis results. Table 5.7 shows the resulting in-memory size of the dependence graph for each dataset. Memory usage varies between 0.7 and 9 bytes per event across these datasets, with the overall average of ***1.12 bytes*** of memory per event.

Construction of the dependence graph from Apache Avro format is fast, taking about a second per 100K events. This is 10x to 100x faster than the rate of data generation, enabling MORSE to operate in real-time. Consumption from our CSR format is even faster, operating at about ***1M*** events per second.

5.7.4 Analysis of Evasion Attacks

A natural question is whether attackers can evade detection by abusing our mechanisms for mitigating dependence explosion. Tag decay can be abused by artificially introducing delays between the time a subject reads input and the time it writes it. Tag attenuation can be abused by making many intermediate copies of data. Both abuses are easy if performed by an attacker-controlled process. However, our system is designed to tag such processes with a *suspicious* subject tag. Since tag decay and attenuation are not applied to suspect subjects, no evasion is possible for such subjects. To successfully abuse our tag explosion mitigation techniques, attackers need to control or co-opt processes with *benign* or *susp_env* subject tags.

Controlling benign processes. The primary means for attackers to control a process is by having it execute their code. This requires the use of a *load, exec* or *inject* operation shown in Table 5.2. Since these operations change the subject to be *suspicious*,, they don't serve the goal of controlling a process with benign subject tag.

Command interpreters such as `python` and `bash` can use read operations to load scripts, and this may provide an evasion path for an attacker. Our system treats read operations as loads for command interpreters, thereby closing off this option. (The list of command interpreters is specified in E^*.)

Another evasion strategy is to use in-memory code. By monitoring the *mmap/mprotect* operations required for this, our $MemExec$ policy can detect such attacks (and did so in our evaluation).

Finally, attackers may use stolen credentials to access an interactive command shell. We rely on additional suspicious activities to detect such attacks. In our experimental datasets, attackers downloaded and executed malware, overwrote library files outside of the normal software update/install mechanisms, or exfiltrated sensitive information. Our entry point identification traced these actions to the shell process. This process was assigned a suspect subject tag, stopping it from abusing our tag optimizations.

Naturally, it is possible for attacks to go undetected. But since we support additional external detectors, this possibility isn't specific to our system. Indeed, an analyst won't even initiate a forensic analysis without signs of an attack, so the tag values become moot.

Co-opt benign process. Attackers may try to have their data copied over many times by benign processes. The tag of the final copy can then surpass the low integrity (or high confidentiality) threshold due to tag attenuation. But this isn't as simple as using a benign `cp` program to copy data. In particular, the attacker would have to control command-line arguments to `cp`. This can be accomplished if the attacker's process created the `cp` process, but then, `cp` would be a *susp_env* subject (discussed further below) rather than a benign one. So, attackers have to rely on pre-existing file-copying workflows, e.g., the backup operation in the `ccleaner` example. We believe it is hard enough to find a string of such benign workflows, but if an attacker manages to do so, the mitigation measures described below provide a way to cope with them.

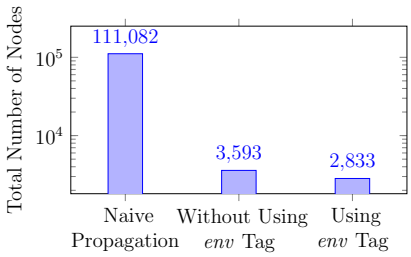

Figure 5.5: Size of the scenario graph without decay or attenuation, without using env tag (i.e., no decay or attenuation for suspect environment subjects) and using env tag.

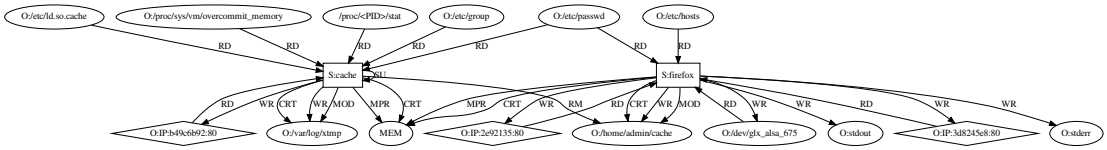

Figure 5.6: ***Firefox Backdoor.*** Firefox was first compromised by a malicious ad server, resulting in an in-memory payload. This generated multiple $MemExec$ alarms. Next, an *Escalate* alarm was triggered, as the attacker escalated privilege using a kernel implant. Installed prior to the engagement, this implant was accessed using the device */dev/glxalsa675*. Subsequently, *DataLeak* alarms were raised when Firefox read and exfiltrated */etc/passwd*. In the second part of the attack, a *cache* process displayed many of the same behaviors (and raised the associated alarms) as the compromised Firefox, but the provenance of this process was missing in the data. As a result, two distinct entry points were identified, namely, the Firefox and *cache* processes. A forward analysis from these entry points resulted in the above graph. Note that *cache* removes a file (*/home/admin/cache*) downloaded by Firefox, indicating that the two attacks are related.

Control susp_env process. Techniques to induce *susp_env* processes to execute attacker's code are the same as those for benign processes. Thus, the detection/mitigation measures mentioned above for benign processes will pose challenges for attacking *susp_env* processes,[2] forcing them to look for other avenues, e.g., by providing malicious arguments, or manipulating their input/output channels. Reflecting the added opportunities provided by this richer interface, we use a quiescent value of $\langle 0.45, 0.45 \rangle$ for these processes, i.e., their data integrity will never rise above 0.5, so they will always be present in the scenario graph seen by the analyst.

Tag attenuation, however, can cause some outputs of *susp_env* processes to have data integrity above 0.5. To avoid missing attack elements due to this, an analyst can disable the use of *susp_env* tag altogether, replacing it with *suspicious* tag. We found that this change had no effect on the scenario graphs for some attacks, but affected others significantly. But when we examine the total size of all scenario graphs in our dataset, it isn't substantially larger after this change. (See Fig. 5.5.) In our experience, we found that for some attacks such as the `ccleaner` and kernel malware, use of *susp_env* led to substantial simplification of the graph that made it easier to understand the attack initially. Starting with this understanding, it was much easier to ascertain that the nodes added by the elimination of *susp_env* tag were unimportant.

Mitigation. In the discussion above, we showed that many of the obvious approaches for abusing our tag optimization don't work. The remaining abuse mechanisms can be mitigated using the "refinement and rerun" process described in Section 5.6: analysts can retry scenario graph construction by varying (a) processes assigned suspect subject tags, (b) attenuation/decay rates, (c) tag threshold for inclusion in the scenario graph, etc. As our system is driven by a small set of rules, the implementation is very fast, enabling retrials to be completed in a fraction of a second (Table 5.6).

5.7.5 Detection Details and Scenario Graphs

For the attacks in our dataset, we discuss below their detection, entry-point identification, forensic analysis and scenario graph generation. Two attacks are omitted because the scenario graph was too large (Dropbear Trojan), or uninteresting (Executable Attachment).

Attacks Within Single Hosts

- *Firefox backdoor:* This attack uses an in-memory payload. The scenario graph for this attack is shown in Fig. 5.6.

[2]Just as command interpreters may use *read* operations for code loading, they may accept code arguments on their command-line. To account for this, we suppress the transition to *susp_env* if a suspect subject executes a command interpreter.

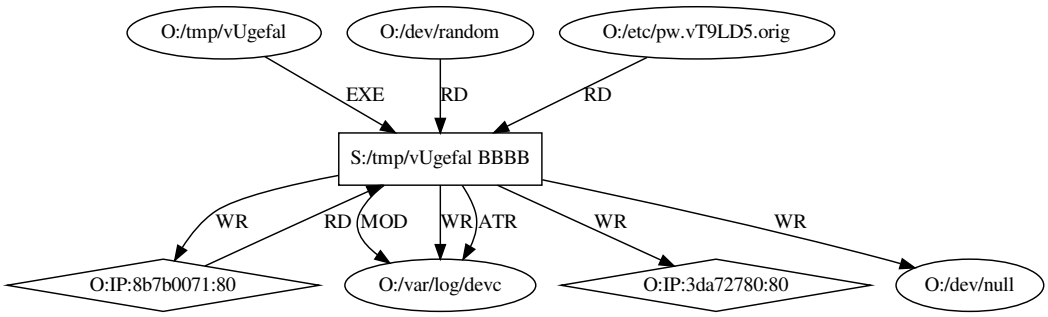

Figure 5.7: ***Malicious HTTP Request.*** This figure shows one of the more successful attempts of this attack, which began with an exploit of *nginx*. A malicious file */tmp/vUgefal* was then downloaded and executed, raising a $FileExec$ alarm. The attacker went on to write another file */var/log/devc*, which was intended to be injected into the *sshd* process, but this attempt failed. Our entry point identification identified *vUgefal* process. A forward analysis from this process yielded the above graph. We also performed a backward analysis to identify the network entry point and the *nginx* process that downloaded */tmp/vUgefal*, but these nodes are not shown above.

- *Browser extension:* This attack exploited a vulnerable Firefox extension. Its scenario graph is shown in Fig. 5.11.
- *Malicious HTTP request:* The attacker tried compromising the *sshd* process on the FreeBSD system but failed. The scenario graph shown in Fig. 5.7 captures one of the attack attempts that includes downloading and executing a malicious file.
- *CCleaner ransomware:* Detection of this attack was described in depth in Section 5.6.
- *Recon with Metasploit:* Similar to the `ccleaner` attack, the attacker uploaded a malicious file */usr/local/bin/hc* to the system using stolen credentials. The file was later executed and used for running recon as shown in Fig. 5.8.
- *Kernel malware:* This attack uses pre-installed kernel malware for privilege escalation, and compromising an existing *sshd* process, as described in Fig. 5.9.

Attacks With Lateral Movement

MORSE tracks lateral movement using cross-host tag propagation. Specifically, if host A reads from host B within the same enterprise, we propagate the data tags from the

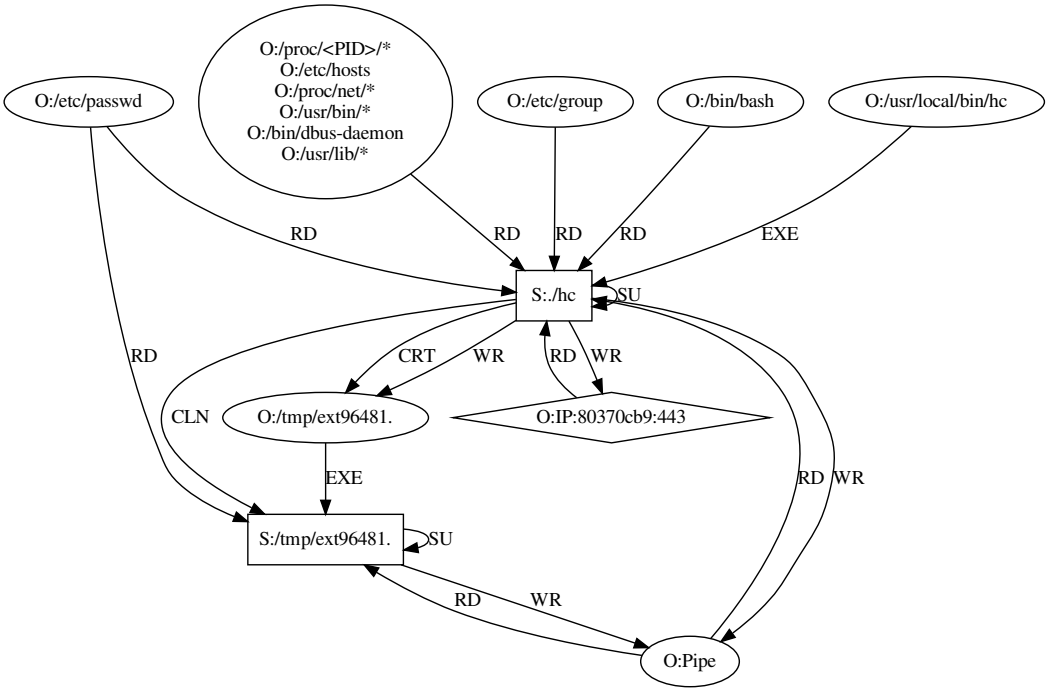

Figure 5.8: ***Recon with Metasploit.*** This attack began with a malicious file *hc* that was `scp`'d onto the victim host using previously stolen credentials. When this file was executed, a *FileExec* alarm was triggered. This process, together with another piece of downloaded malware */tmp/ext96481*, probed and exfiltrated sensitive data to a remote IP address. These actions raised *DataLeak* alarms. MORSE traced these alarms back to *hc*. A forward analysis from this node results in the above scenario graph. A backward analysis from *hc* revealed the `scp` process involved and the network entrypoint, but these are not shown above.

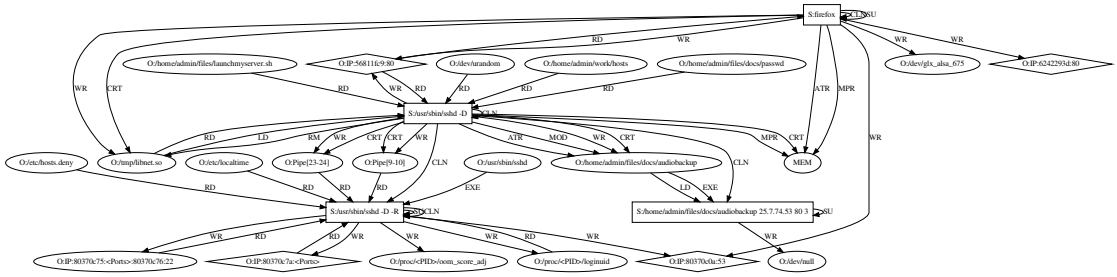

Figure 5.9: **Kernel Malware.** *Firefox,* compromised by a malicious website, executed an in-memory payload that triggered several $MemExec$ alarms. Next, an $Escalate$ alarm was triggered, as the attacker escalated privilege using a kernel implant installed prior to the engagement. *Firefox* then downloaded a malicious file */tmp/libnet.so*, which was meant to be injected into an existing *sshd* process. However, in the data, there is no injection, but *sshd* did raise several $MemExec$ alarms, as well as a $FileExec$ alarm due to loading */tmp/libnet.so*. Next, *sshd* downloaded */home/admin/file/docs/audiobackup* and made it executable, raising a $ChPerm$ alarm. It also performed some recon and exfiltrated the information, causing several $DataLeak$ alarms. In total, *more than 500 secondary alarms were raised,* all tracing back to *Firefox.* A forward analysis, performed about 10 minutes after the attack, yielded the above scenario graph.

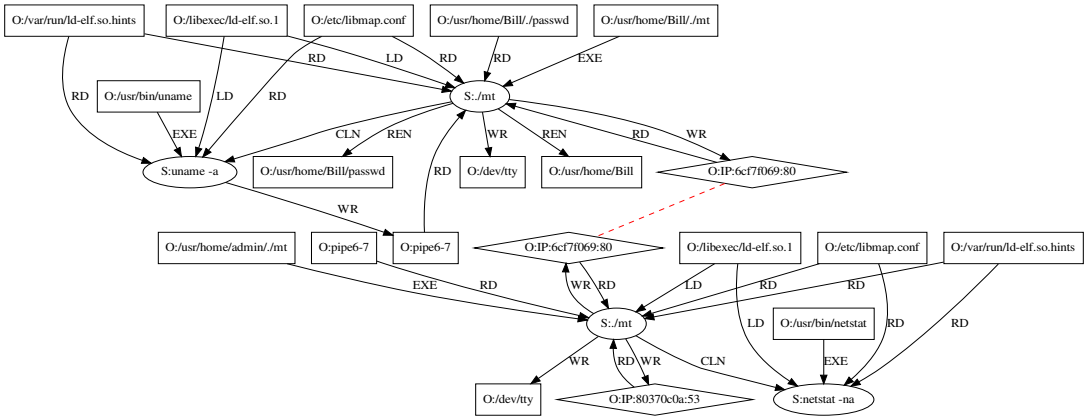

Figure 5.10: **Recon with Rootkit attack.** This attack began with uploads of *mt,* a rootkit, to two FreeBSD hosts. When *mt* was executed, a $FileExec$ alarm was triggered. As *mt* gathered and exfiltrated sensitive information to an external IP address, $DataLeak$ alarms were raised. These alarms were clustered independently on the two machines, tracing back to the *mt* process. A forward analysis from this process yielded the above graph. Note that the two graphs are disconnected, except for the dotted line showing the shared attacker site. A backward analysis from *mt* showed that the attacker logged in using *scp*, presumably using stolen credentials.

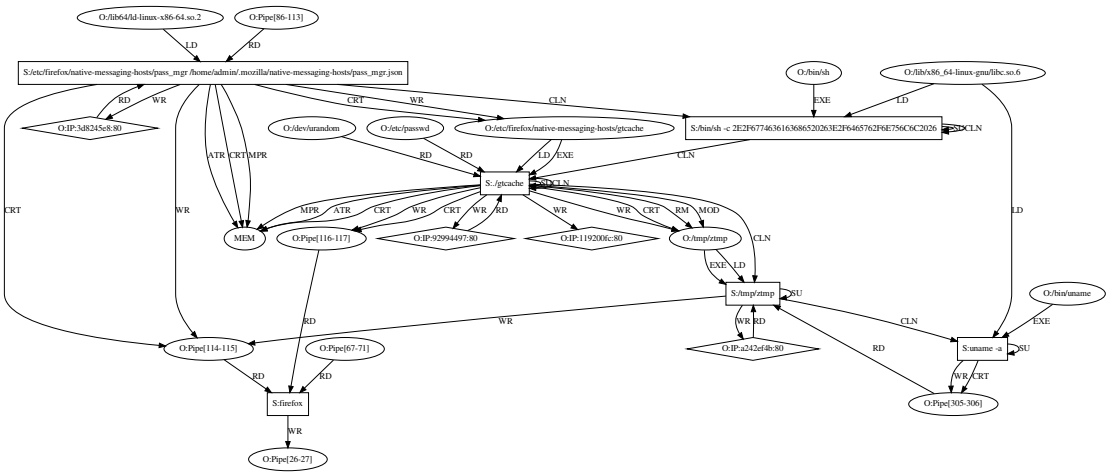

Figure 5.11: **Browser extension.** The attack started when a vulnerable browser-plugin *pass_mgr* got compromised while visiting a malicious website. This raised $MemExec$ alarms. Next, the compromised plug-in downloaded a program *gtcache* and executed it, resulting in a $FileExec$ alarm. In turn, *gtcache* downloaded and executed *ztmp*. Both programs performed recon to collect and exfiltrate sensitive information to the network, resulting in several $DataLeak$ alarms. Tracing back from these alarms, MORSE identified *pass_mgr* as the entry point. A forward analysis from this node yielded the above scenario graph.

sending subject on B to the receiving subject on A. Subject tags are also propagated in the case of remote access services. Hence, if a suspicious process on host B launches an ssh session on A, the `sshd` process on A will also be tagged suspicious. With this tracking, MORSE was able to detect both attacks in our dataset that involved lateral movement:

- *User-level rootkit:* The attacker utilizes a pre-existing user-level rootkit to log into a Linux host, and then moves laterally into a second host. See Fig. 5.12 for additional details.
- *Recon with rootkit:* The F-4 attack in Fig. 5.10 is simpler, consisting of two instances of the same attack on two machines.

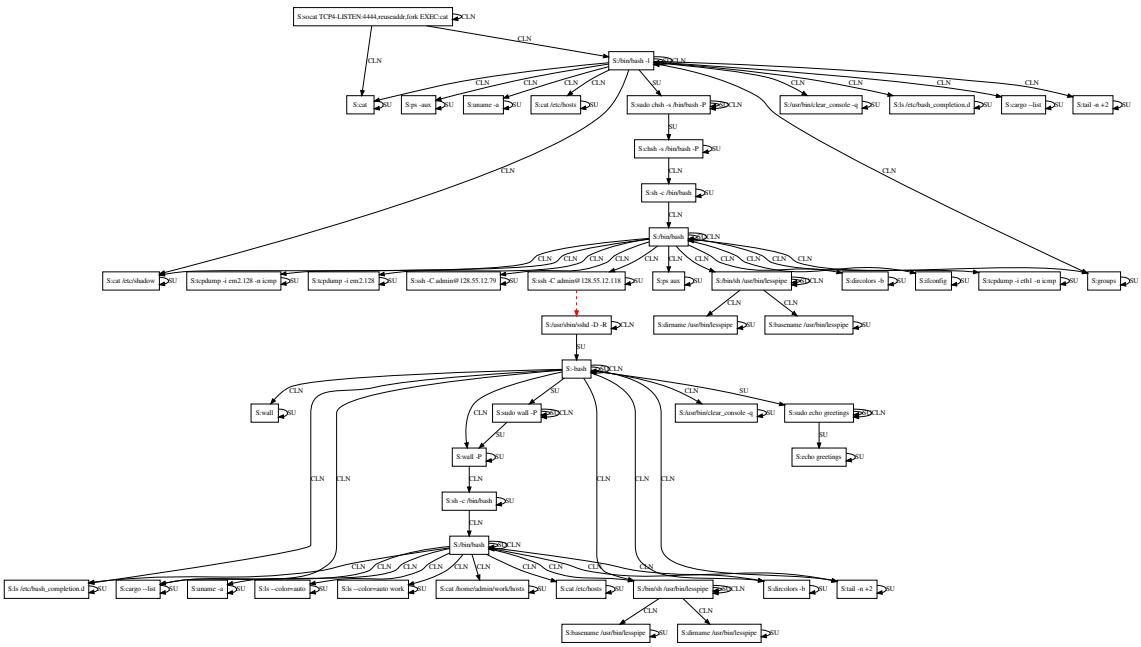

Figure 5.12: **User-level rootkit.** This attack takes advantage of a user-level rootkit, in the form of a shared library `libselinux.so`, which had been installed on the victim host prior to the start of the engagement. During the engagement, the attacker accessed this rootkit to exfiltrate */etc/shadow* to a remote IP address, raising a *DataLeak* alarm. This was the sole indication of unusual behavior in the audit data, thus making this the most stealthy attack in our dataset. The attacker, possibly after using password cracking on this shadow file, obtains access to a second machine via *ssh*. Since the sole alarm was generated by a *bash* process, we marked it suspicious, and performed a forward analysis from there. Since the resulting graph was large, we refined the forward analysis to follow only process creation and execution edges to yield the above graph. Note that the attacker ran several commands to collect sensitive data, such as `tcpdump`, `ifconfig`, and `ps`. Other notable commands include `clear_console` and `chsh`. On the second machine, since a suspect process from the first machine connected to it, the target process (*sshd*) was marked as a *suspect* subject by MORSE. The scenario graph originating from this *sshd* process has been shown together with the scenario graph generated on the first host, with the network connection indicated with a dashed line.

Chapter 6

Probabilistic Confidentiality Tag

The ultimate goal of most APTs is to exfiltrate sensitive data relevant to the attacker's interest. According to IBM's X-Force Threat Intelligence Index 2021[64], the top three attack types of 2020 are Ransomware, Data theft and Server access. 59% percent of ransomware attacks also blend in data theft so that the attacker can threatened to leak the sensitive data if a ransom is not paid. To gain access to sensitive information, the attackers need to gain control of the targeted network by first establishing a foothold and then moving laterally within the network to gain higher privilege. Although each APT campaign has its own degree of sophistication, however, the literature agrees that an attack can be decomposed into some general phases. The Tao of Network Security Monitoring subdivides the attacks in to five stages [23] and the Cyber Kill Chain (Lockheed Martin) into seven stages [93], whereas MITRE ATT&CK proposes a more fine-grained partitioning called *tactics* [106]. Despite the reference model, the first step always requires gathering information on the target and it is commonly defined as "reconnaissance".The goal of the reconnaissance phase is to identify weak points of the target network to find weaknesses in its defense. Attackers gathers information of the victim organization for identifying exploitable weaknesses which may lead to infiltration of the target network. In some cases the exfiltrated data is also used for reconnaissance for creating new routes to access other victim environments.

Tag-based or Provenance-based approaches provide a robust basis for detecting exfiltration because it does not depend on specific patterns of confidential data use. Pattern-based approaches are prone to false positives because of use by legitimate applications, and false negatives due to attacks that purposely evade known patterns. Instead, provenance-based techniques tracks the flow of confidential data to untrusted network endpoints, so it is unaffected by the exact manner in which confidential data is gathered. Moreover, legitimate use cases rarely involve untrusted network endpoints, so those can be largely avoided as well.

The robustness of provenance-based approaches does not address the orthogonal problem of cataloging all confidential data sources. Every file on every host has to be tagged, which is a daunting task, considering that a single host often contains hun-

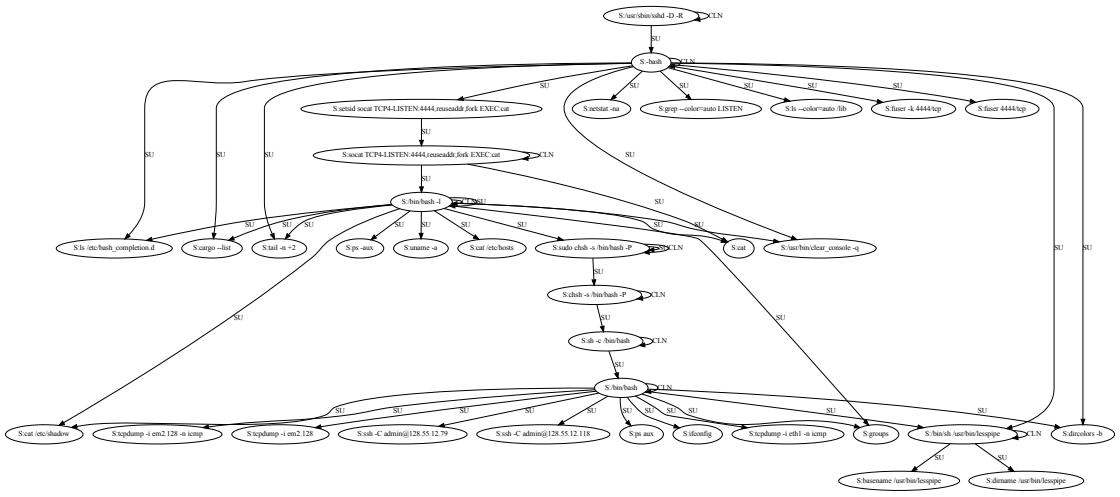

Figure 6.1: Motivating example: Kernel Rootkit.

dreds of thousands of files. Moreover, not every confidential file is equally sensitive: an *ssh* private key is a lot more valuable to an attacker than a typical data file, which in turn is more valuable than most binaries. Previous tag-based approaches rely on a few discrete tag values (e.g., "confidential" and "public"), and does not adequately capture the range of differences in sensitivities of files. Moreover, existing tag-based approaches are not sensitive to the quantity of information exfiltrated which is crucial for detecting if an attacker is carrying out reconnaissance on the system using system tools, e.g., network information, host information etc.

In this chapter, we have developed and implemented a new approach that assigns real-valued confidentiality tags to files. By interpreting a file's confidentiality tag as a probability that an attacker would target a file, we take into account the increased value of multiple confidential files. At the same time, our probability interpretation models the diminishing returns from each additional exfiltrated content. Finally, we have developed a novel approach for automating confidentiality tag assignment by observing the use of files during normal observation. In particular, files that are always read prior to privilege change, such as the shadow password file, are assigned high confidentiality.

6.1 Motivating Attack Scenario

In this section, we illustrate the importance of detecting *reconnaissance* and *data exfiltration* using an attack scenario from a red team engagement that was carried out as part of the DARPA Transparent Computing program. The red team's goal was to organize a highly stealthy cyber- attack, using the following stealthy maneuvers:

- *Supply Chain Attack.* The red team placed a malicious library containing a rootkit called *Azazel* before the data gathering started to simulate supply chain attack. This enabled the attacker to create a backdoor to later gain access to the system.

- *Stolen Credentials.* The red team assumed that the login credentials of the victim user had already been stolen by the attacker. This enabled the attacker to gain access to the victim machine without raising any suspicion.

- *Reconnaissance using system tools.* The red team performed reconnaissance on the victim machine and the network using system tools only and moved laterally to another machine using the gathered information.

Fig. 6.1 shows the process tree of the dependence graph (also known as provenance graph) relating to this attack, constructed from the audit log produced by the Linux auditd daemon. Ovals in this graph are subjects (processes). Edges in the graph correspond to system events such as read, write, load, fork, execve, and so on. Edges are oriented in the direction of information flow, and annotated with event names. To reduce clutter, we only display the process tree of the attack scenario.

Logically, the attack begins with the theft of login credentials for the user Bob, but this step is assumed to have taken place "out-of-band" and is not visible in the audit data. Using these credentials, the attacker Trudy logs into Bob's machine B using ssh. Also, a malicious library file called *libselinux.so* was also placed inside the */lib* directory which also assumed to have taken place "out-of-band" to simulate a *Supply Chain Attack.*

In one of the ssh sessions, Trudy logs into Bob's machine and modified the LD_PRELOAD variable to point to the malicious *libselinux.so* shared object and exits the system to reduce the attack footprint. Later, when Bob logs into the system and ran the *netcat* command, it allowed the rootkit to be loaded into it's memory and created the backdoor which would allow the attacker to gain root privilege to the system as well. Once the backdoor was created, Trudy would later enter the system using root privilege and perform some reconnaissance on the system to map the network, steal the shadow file, gather group information, etc. Using the gathered information Trudy would perform lateral movement within the network to access the systems that contains sensitive information to her interest.

Note that benign activities surrounding the attack far exceed the attack activity. To reduce clutter, we have elided many of these benign activities, including: many ssh sessions for Bob, the details of all the files involved during the sessions, subsequent activities of ssh logins, and so on. If those details were included, then the picture will be at least 10 times larger.

Challenges: This attack poses many challenges for detection and forensic analysis tools. By using stolen credentials, Trudy enters the system without triggering any

alarms and completely disassociates her connection of interfering with the malicious shared library. By using a supply chain attack no alarms would be triggered when loading the malicious library into a legitimate system tool by a legitimate user. The only malicious step that took place in this machine is the reconnaissance step taken to gather sensitive information about the network and the victim machine which by themselves are not suspicious.

The challenges faced in finding the entry point and generating the complete attack scenario is even more formidable. The only single event malicious step in this attack took place towards the end of the attack when Trudy accessed the *shadow* file. Generating a forward graph from that particular process would result in an incomplete attack graph. Also, the attacker cloned the bash process multiple times to hide the backdoor used using the *socat* process. Only by detecting the *socat* process as the entry point of the attack and running the forward search could give the actual and complete attack scenario. Otherwise an analyst would have to manually search in order to find every single part of the attack, which could take days to months.

6.2 Approach Description

Our attack detection and campaign reconstruction techniques operate on a *dependence graph.* The nodes in this graph represent *objects* and *subjects*, and edges represent events involving two nodes. Objects consist of files, network connections, and other types of inter-process communication mechanisms. Subjects are processes, while events are system calls.

Like most previous techniques on attack campaign investigation, our techniques are also based on dependence analysis. A node in the graph can have numerous ancestors, each of which could have potentially influenced its content or behavior. When analyzing a particular event on a particular node, it would be impractically inefficient to consider each of its ancestors individually, as there can easily be millions of such ancestors. A common technique for speeding up this analysis is to maintain a compact summary of information about the ancestors in the form of *information flow tags.*

We rely on a graph dependency based formulation introduced more recently in the context of attack campaign reconstruction [61, 62, 63, 105]. Two tags are associated with each object: a *confidentiality* tag that indicates if the object contains sensitive data, and an *integrity* tag that pertains to the trustworthiness of its data. Subjects also contain data, so they too have data confidentiality and integrity tags. In addition, we introduce an additional *sensitivity* tag for subjects. Unlike confidentiality and integrity tags that primarily capture provenance, sensitivity tag is primarily used during training for inferring confidentiality tag for objects for tag assignment.

Our method is characterized by the three tags mentioned above, the rules for computing and updating these tags, and tag-based policies for attack detection. We

use the same interpretation of integrity tag mentioned in previous works [61, 62]. That is why we omit the description for that. We discuss the subject sensitivity tag in detail in the training section.

6.2.1 Probabilistic Confidentiality Tags

The most common goal of stealthy attack campaigns is to access and exfiltrate confidential data, e.g., login credentials, financial information, intellectual property, etc. Confidentiality tags provide important contextual clues that enable accurate attack detection: data that is likely to be targeted can be assigned high confidentiality (i.e., tag values close to 1.0) while the less likely targets are given low confidentiality (i.e., tag values close to 0.0). Using these tags, detectors can better discriminate attack activities from benign background activity.

Accurate tag assignment is a central challenge in applying tag-based techniques. While it is easy enough to assign a confidentiality tag of 1.0 to highly sensitive information such as passwords and private keys, the choice of tags for other files (especially user files) is far from clear. The underlying problem is the lack of a constructive definition of confidentiality tags, one that will provide a basis to assign numerical values. This lack of definition not only impedes tag assignments for files, but also makes it difficult to reason about the combined value of multiple (moderately) sensitive files to an attacker. We overcome this problem by developing a probability-based definition of a confidentiality tag.

Definition 8 (Confidentiality Tag)**.** *A confidentiality tag of data item is intended to capture the probability that this data will be targeted for theft/exfiltration by attackers.*

It is not easy to predict every attacker goal and/or preferred attack methods, so the probability assignments won't be perfect. Nevertheless, the definition provides a basis for assigning numbers in the range $[0, 1]$ to data files. More importantly, the probability interpretation enables reasoning about combined value of multiple files. If files f_1 and f_2 are useful to an attacker with probabilities C_1 and C_2 respectively, then their combination would be useful with a probability C given by

$$C = 1 - (1 - C_1)(1 - C_2) = C_1 + C_2 - C_1C_2 \tag{6.1}$$

This formula is simply the product rule for probabilities, assuming that C_1 and C_2 are independent[1]

Note that this rule captures our intuition that the combined value of two files is more than a single file. If both C_1 and C_2 are very small, then their combined value is very close to $C_1 + C_2$; otherwise, the C_1C_2 term cannot be ignored, so the combined

[1]Specifically, $1 - C_1$ and $1 - C_2$ represent probabilities that f_1 and f_2 are *not* useful to the attacker. Thus, $(1 - C_1)(1 - C_2)$ represents the probability that neither file is useful to the attacker. The complement of this quantity represents the probability that at least one of the files is useful to the attacker.

value is somewhat less than the sum. This captures the intuition that as the attacker accesses more and more sensitive data files, the incremental value added by each file becomes smaller and smaller. As an example, if the attacker reads 10 files, each with a confidentiality of 0.01, the combined confidentiality is close to $10 * 0.01 = 0.1$. However, if he reads 100 such files, the combined confidentiality is 0.63, which is far less than $100 * 0.01 = 1.0$.

6.2.2 Confidentiality Tag Assignment

Our technique for inferring object confidentiality tags relies on a training phase. The inferring scheme is designed to facilitate human understanding using the subject *sensitivity* tags, so that an analyst can examine the learned confidentiality tags, and override them if she so desires. For example, organization specific sensitive files that would require manual tagging.

Subject Sensitivity Tag In a benign setting highly confidential files containing system related information are always accessed by subjects with high privilege, for example the */etc/shadow* files can be accessed only by a *root* process. On the other hand, processes that communicate with networks do not access confidential files or sensitive configuration files. For inferring files confidentiality we categorize subjects into multiple groups. We assign real values between 1.0 and 0 for sensitivity tags for inferring object confidentiality from these values. Our assumption here is that files that are only accessed by processes with higher privilege have high probability that an attacker would target that file. If any files are publicly available or that can be accessed by any process are unlikely to be an attacker's target. Another observation is that, a normal user rarely read system information related files. These files are mostly read by background processes in the system but can be viewed by anyone. Reading just a couple of these files may not be harmful but doing it for numerous files could indicate that someone is trying to gather information about the system. Files that are regularly accessed by normal users or processes that does not require any high privilege are mostly never a target of the attacker. For these reasons we infer confidentiality tags based on the type of subjects they are accessed in benign setting. Here, we begin by defining the sensitivity tags used to differentiate these groups:

- *inv. subject:* Only the *init* process is part of this group. The assigned value for this group is 1.0.
- *root subject:* All subjects with root privilege are put into this group. The assigned value for this group is 0.8.
- *non-root benign subject:* Any non-root processes with no network activity are put into this group. The moment the process access (read/write) a network connection, the tag is updated. The assigned value for this group is 0.4.

- *non-root unknown subject:* Any non-root processes after making a network connection are put into this group. The assigned value for this group is 0.0.

The number corresponds to how many of the files if accumulated will be considered to be confidential. Reading just a couple of files that are mainly accessed by the root subject could make a process instantly to be confidential. Where as files that are mainly accessed by benign subject requires a whole lot to be read to be confidential. On the other regular files are not given any confidential properties.

We further subdivided the sensitivity tags between *non-root benign subject* and *non-root untrusted subject* depending on the distance of their interaction. For example, if a non-root benign subject reads files that are updated by non-root untrusted subjects we assign the sensitivity tag value of 0.1 for that subject. If a non-root benign subject reads files that are updated by a non-root subject with sensitivity tag value of 0.1 we assign the sensitivity tag value of 0.2 for that subject and so on.

We don't create any group based on the network connection for *root subjects* because in benign settings we observed root processes with network activity interacting with sensitive files e.g., *sshd* processes.

Inferring Confidentiality Tags from Training Data. The training phase requires data that contains only benign activity. During the training phase for a particular file, we keep track of all the subjects and their sensitivity tags that read that file and calculate the geometric mean of the sensitivity of the readers. If file **A** was read by n subjects having sensitivity of $S_1, S_2, ..., S_n$. The geometric mean of the sensitivity of the readers of file **A** will be,

$$G(S_A) = \exp(\frac{1}{n}\sum_{i=1}^{n}\log(S_i + \delta)) - \delta$$

Here S_i can be zero and to handle that we add and later subtract a very small δ value.

Once we have calculated the mean sensitivity of the readers, the initial confidentiality of file **A** is inferred as,

$$C_A = G(S_A)$$

6.2.3 Update and Propagation of Tags

Once the tags are inferred from the training we assign them to the objects during the start of detection time and let them propagate. Note that we do not use the subject's sensitivity tag during detection. The sensitivity tag can be replaced by other tags during detection representing the subjects behavior as in [61, 62] and combined to get better results. But in this work we are mainly focusing on the data tags. We update the data tags using the following methods:

Propagation of Confidentiality Tags. When a process reads a file, any confidential data in the file moves into the address space of the process. For this reason, the confidentiality tag of a subject is updated using Equation 6.1 whenever it performs a read. Subsequently, when it writes to an object, confidential data can propagate from its memory to this object. So objects inherit their confidentiality tags from subjects. Specifically, when a subject creates (or completely overwrites) an object, the object's confidentiality tag is set to the confidentiality of the subject. For writes that partially overwrite the original content, the object accumulates the subject's confidentiality tags using Equation 6.1.

Exceptions from these default propagation rules can be specified for specific applications. For instance, an authentication server may read highly confidential data such as the user's password, but the application may be deemed trustworthy and not prone to leak this information. In such a case, it is appropriate to create a specialized rule that decreases the confidentiality tag's probability by a weighting factor before propagation. We use the integrity tag of the subject to control the propagation of confidentiality during write. If the subject's confidentiality is C_s and integrity is I_s then the object's confidentiality C_a, after the write operation will be,

$$C_a = 1 - (1 - C_a) * ((1 - I_s) * (1 - C_s) + I_s * 1.00)$$

This ensures that if a subject contains no exploit then the objects confidentiality will stay the same otherwise it will be decreased based on the probability the process containing exploit.

Propagation Rules for Operations on Code. During a clone operation we simple propagate the tags as it is from the parent to the child process. On load events we use the same propagation rules as read.

Although *exec* is similar to load in terms of loading new code for execution, there are several important differences as well. In particular, *exec* causes data memory to be cleared, that we set the confidentiality tag to be 0 and the integrity tag to be benign, to indicate the absence of confidential data, and to reset its data integrity tag to be high. Next, we update the confidentiality and the integrity tag to be the same as the object that was executed.

6.2.4 Attack Detection

We only generate two types of alarm in these work. Note that, we do not focus on alarms that are generated on code operations such as *File Execution* alarms which are generated extremely low in number and are detected quite accurately and demonstrated in previous work. But as APT attacks are getting sophisticated, attackers are moving further away from file based attacks. They are mostly carrying out their attacks by gaining access to the system through stolen credentials, supply chain at-

tacks and using system tools. That is why we only focus on alarms generated using the data tags. We mainly have two types of alarms:

- *ProcConf.* If a process with low integrity tag (high probability of being compromised) becomes confidential i.e., $(1 - C) < t$, where t is a threshold value, this alarm is raised.
- *DataLeak.* If a process with low integrity tag (high probability of being compromised) becomes confidential i.e., $(1 - C) < t$, where t is a threshold value, and starts writing to an unknown network address or to the terminal, this alarm is raised.

Although by definition *ProcConf* alarm will always be triggered before the *DataLeak* alarm but the *DataLeak* alarm can be used to track individual applications that are leaking sensitive information and also the network address it is leaking the information to. Also if a benign process already containing sensitive information is compromised later the *ProcConf* alarm will not be generated. In that case, the *DataLeak* alarm can help to detect the attack during the exfiltration.

6.2.5 Entry Point Identification and Attack Scenario Reconstruction

One of the core goal of our work is to identify the initial step of an attack campaign. Although reconnaissance is the initial steps of an APT attack, an attacker could perform it in multiple phase in a long time to hide this step. As a result, numerous alarms will be raised at different time of the attack campaign. It is infeasible for an analyst to track down each alert individually, so we have developed an alert aggregation and prioritization technique further described below.

At first we associate an alarm with a subject. Given an alarm originating at node n, we perform a backward search along the process tree in the dependence graph for the closest group of nodes N that also triggered an alarm. If we don't find such a node, then we place the current node n to a new group N and mark node n as the entry point of the alarm. Otherwise, the new alarm n is placed in group N and the marked node in that group is considered the entry point for alarm n.

The backward search is performed until it reaches the initial process or the remote login server process such as an *ssh* process.

Next, an analyst can now investigate each alarm group and make an informed decision. In cases of a group containing numerous alarms can be an indication of several reconnaissance attempts are being taken place and an analyst could investigate that group by generating the attack scenario using a forward analysis.

The forward analysis is carried out by running a depth-first search from the entry point subject. To not clutter the scenario graph the depth-first search only follow subject to subject edges and subject to object edges. Object to subject edges are

ignored at first. If numerous suspicious files were dropped into the system in that case the depth-first search criteria can to be modified to allowed to include one or more object to subject edge on each unique path from the entry point. This would prevent from cluttering the scenario graph and provide better summary of the attack.

6.3 Evaluation

Platform. The system under attack consisted of multiple hosts running recent versions of Ubuntu Linux. Our analysis was performed on an Ubuntu 20.04 Linux laptop with an Intel 2.7GHz i7-7500U CPU and 16GB memory.

Threat Model. We assume that attackers cannot compromise audit record collection or the log itself. Although bests results can be obtained if the log collection is performed on clean systems and every single system call events are captured but, in real-world systems it is hard to satisfy these criteria. The datasets we tested our techniques on contained pre-existing malware and even crucial system call events were dropped/missing. However, ConfTag was able to detect every single reconnaissance and dataleak steps despite these factors.

6.3.1 Dataset

Similar to previous research on attack reconstruction from audit logs [61, 62, 105], we valuate our system using attacks carried out by an independent red team, as part of the DARPA Transparent Computing (TC) program. However, the DARPA TC datasets had some drawbacks due to multiple failed attack attempts and reusing the same type of attacks throughout the engagements. Due to this factor, to better evaluate our system's capability we also collected our own attack logs and used them for attack detection which, we discuss in detail next. We also collected data logs from long running benign systems to perform a false positive analysis.

Dataset from DARPA TC Engagements

DARPA TC program carried out five engagements in total where engagement 3 and engagement 5 datasets are both publicly available [3]. We evaluated our system using the data collected by the Trace team on both of these datasets including data from engagement 4. Most of the attacks carried out in these engagements assumed the attacker already had access to *stolen credentials* which was carried out before the data collection started. Moreover, many of attacks depended on preexisting malwares and rootkits which were also installed in the system before the data collection started. Next, we further discuss the attack scenarios during each of the engagements:

Engagement 3. There were mainly 4 attacks that partially succeeded during this engagement. Among them one of the attacks is a website password stealing where the victim was lured to a malicious website and was scammed to provide sensitive information. These attack did not generate any relevant system call events and is not visible in the dataset. The other attack consisted of an attacker sending a malicious attachment. But the attack failed soon after the victim clicked and executed the malicious attachment. As a result, there were no data exfiltration or reconnaissance steps in the attack and is out of scope for our system.

One of the two successful attacks in the dataset was a *Firefox in-memory* attack where a Firefox process was compromised and a malware was dropped and executed which later gathered information about the system and exfiltrated it. The in-memory attack on Firefox read numerous files in the victim machine (probably to find a vulnerability). One interesting aspect of the attack is that there were multiple relevant events missing which could be a part of the attack. The other successful attack was a pre-installed malicious browser extension which also gathered information of the system and exfiltrated it.

Engagement 4. This dataset contained 4 attacks in total and took place within two Linux Ubuntu machines. Most of the attacks during this engagement assumed that the attackers had access to stolen credentials. The attacks consisted of a *ransomware*, attack carried out using *Metasploit* that performed a system wide search to check if any security sensitive applications were running, a compromised *Firefox* process injecting a *sshd* process and a stealthy preexisting *rootkit* attack through which an attacker entered the system and performed system wide reconnaissance and also lateral movement.

Engagement 5. This engagement generated the largest and longest dataset in the DARPA TC program, involving three Linux Ubuntu machines and generated almost 5 billion events in total. This dataset contained 3 attacks where, two of the attacks were stealthy *rootkit* attacks that failed midway. The other attack involved a *Firefox* process which was compromised due to an in-memory attack and later compromised an *sshd* process using in-memory injection technique. In all three of these attacks, the attacker was able to perform some form of reconnaissance on the system and some data exfiltration as well.

Dataset Generated and Collected in Our Lab

Due to a lot of failed attacks during the engagements the attacks did not contain proper reconnaissance and data exfiltration steps. For that reason, to further evaluate our system we generated a dataset containing 6 different attack scenarios. The attacks contained little to almost no file based attacks and were mainly carried out using system tools. Here are the description of the attacks that were carried out:

Scene A. In this scenario the attacker using the *cronjob* started a bash process with root privilege. Next using the root privilege the attacker ran system wide reconnaissance and gathered the information in a file. This file was later compressed and uploaded to the attacker's server for exfiltration.

Scene B. The attacker gained a root shell in the system by using the $LD\_PRELOAD$ environment variable. Once root access was gained, the attacker gathered sensitive information of the system and redirected that to a file. The file was later exfiltrated using an *scp* process which was initiated from the attacker's machine using stolen credentials.

Scene C. In this scenario the attacker compromised the *vim* process which resulted with a shell. This time the attacker only gathered very low sensitive information regarding the system such as the *bash_history* file and network related information and gathered that into a compressed file. Later this file was uploaded to the attacker's server. Using this information the attacker created a reverse shell on the victim's machine to use it as a backdoor and erased all previous steps.

Scene D. Same as before the attacker compromised the *vim* process which resulted in a root bash process this time. This time the attacker gather info about the network of the victim's system and then created a new user in the system. Attacker's ssh key was added to root *authorized_keys* for persistence.

Scene E. The attacker combined multiple steps from the previous scenarios in this attack. At first the attacker got a root shell by compromising the *vim* process. Next, a script file was created and scheduled to run by the *cronjob* with root privilege. The script creates a persistent backdoor with root privilege using *systemd*. To avoid detection by security tools, the script is padded with zeroes in the end to turn it into a large file. This step was mainly carried out to avoid generating hash for the file.

Scene F. In this scenario the attacker compromised the system using stolen credentials so that whenever the *find* command is run, it will provide the attacker with a root shell. Using the root shell the attacker gathers very low sensitive information about the system such as, *bash_history* and network information and redirected that information to a compressed file. Next the attacker inserted a backdoor on the victim's *php* web server and also uploaded the data to the web server for exfiltration. Later the attacker from their machine accessed the backdoor and also downloaded the data from the web server.

6.3.2 Analysis of the Confidentiality Tag Inferred during Training

We take a look at the confidentiality tags that were automatically assigned during training to evaluate our tag inference technique.

At first, we look at the confidentiality of the files that should have high probability of being target of an attacker. Files such as */etc/shadow, /etc/group* and *ssh keys* automatically received extremely high probability during training. Exfiltrating even a single one of these files will trigger an alarm in our system. We also notice every single file in the */root/* directory also received extremely high probability as well. We also looked at the */proc/pid/maps* file which received a 40% probability. Multiple process would access these files so a couple of access should not massively increase the probability of data gathering taking place. But if someone is iterating through all the map files in the */proc/* directory could be an indication that an attacker might be performing reconnaissance by mapping out the process memory space. Log files such as */var/log* also was marked having a high probability. Interestingly the */etc/passwd* file received a very low probability as many processes with network connection read this file.

Next we look at some interesting findings. We discover that the */etc/ssh* directory which contains the public and private keys of the host have also received high probability. Configuration files such as in the */etc/pam.d/* directory and */etc/modprobe.d/* also received higher probability of containing sensitive information. Although writing to these files may be considered malicious, but an attacker might be interested in knowing the configuration setting to perform an attack. All device files in the */sys/* and */dev/* directory also received high confidentiality as well.

We also did notice benign files receiving high probability. For example, files related to *dpkg* because only a root system administrator was allowed to perform package updates in our benign dataset. Also files in the */run/* directory received high confidentiality as well.

6.3.3 Threshold Selection

In this section, we tune the threshold parameter t for generating alarms in our system. As mentioned earlier, if a process contains confidentiality C both *ProcConf* and *DataLeak* alarm is triggered on $(1 - C) < t$. We use the precision and recall graph on Engagement 4 dataset to tune the threshold parameter t, as shown in Fig 6.2. We used this particular dataset because, this had 3 successful attacks which can be used to generate better precision and recall graphs.

As we can see from the graphs, on a threshold value of 10^{-3} we get the highest precision along with the highest recall. But because precision falls right after that point, we select the value just before that in our evaluation which is 10^{-4}. In our evaluation we use the same threshold on every other dataset including the data gathered

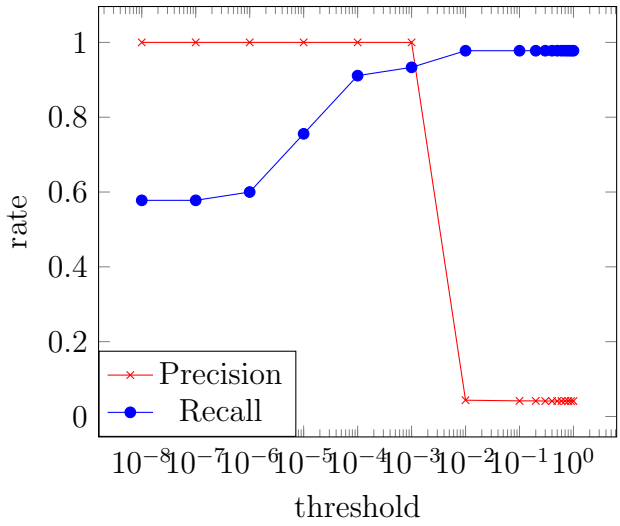

Figure 6.2: Precision and Recall for detecting Threshold value t.

in our lab.

6.3.4 Evaluation of Inferred Confidentiality Tag and Confidentiality Accumulation

Using the threshold selected in the previous section we evaluate on system on all the datasets which is shown in Figure 6.3. We evaluate our system by first generating the precision and recall percentage and then calculating their F1-score (harmonic mean) when both inferred tags are used along with accumulation. We also show how the system would perform without the accumulation. Finally we selected some files to be highly confidential which are mentioned in the MITRE ATT&CK framework [106] to create the Base initial tags. The files includes the *shadow* file, *ssh* keys and the */proc/pid/map* files. Although the */etc/passwd* file was considered to be a highly confidential file in the DARPA TC program, but marking it as highly confidential significantly reduces the performance of the *Base* system and it generates massive amount of false positives. That is why we did not consider that file to be confidential in the Base technique.

As we can observe from the results, in many of the cases just using the inferred confidentiality tag or the base tags most of the attack are not detected at all. However, even when some part is detected the coverage is significantly lower. As for using the inferred confidentiality tag along with accumulation most of the attacks are detected with high coverage and low false positives. Only in *Engagement 5* we do see a lower amount of precision but still high recall. The reason behind that is there was a backup *scp* process running in the system which was gathering a huge number of files in the system and sending it to a remote network location. In our system, all network

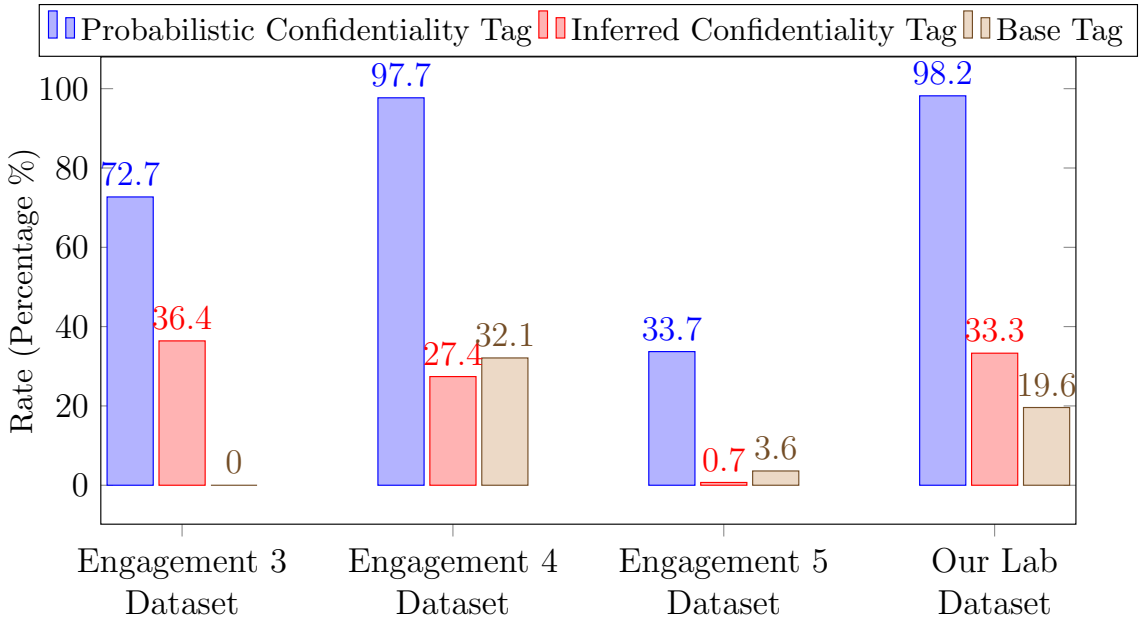

Figure 6.3: Comparison of F1-score (Harmonic mean of precision and recall value) on using the Inferenced confidentiality tags and accumulation, just the Inferenced confidentiality tags and using Base confidentiality tags.

Dataset	# of Events	Duration	False Alarms
Server 1	5.19 M	119h:13m:29s	0
Server 2	4.53 M	105h:08m:22s	1
Server 3	20.9 M	104h:36m:43s	0
Eng 5	1.43 B	248h:21m:34s	1

Table 6.1: False positive alarms generated on benign datasets.

connections are considered to be untrusted because no previous information about trusted network address were provided in the dataset. The precision could be raised higher if that information was provided. In *Engagement 5* the low precision happened due to how the attack was carried out. The firefox process that was part of the initial attack started running on the last day of the engagement. As the firefox process was previously compromised, it kept on iterating through the system and generated those false positives. But due to the entry point detection, these false alarms were also grouped with the initial compromised firefox process and was able to generate concise graph for the analyst. Whereas, the base technique failed to detect the compromised firefox and only generated an alarm when the *shadow* file was exfiltrated in the second attack.

6.3.5 False Positive Analysis

In this section we look at how our system performs on benign long running dataset shown in Table 6.1. We collected three dataset from three different lab servers which ran for multiple days with normal user usage. As we can see from the table, only a single *DataLeak* alarm was raise in total on one of the server. We also used one of the machines in *Engagement 5* dataset for the false positive analysis because that particular dataset did not contain any attack according to the ground truth but did contain benign activity. Even with huge traffic the dataset only generated a single Dataleak alarm. This analysis shows that our system does not generate much false positives when benign activities are carried out in the system.

Chapter 7

Conclusion and Future Work

APTs continue to be the biggest threat not only for large organizations but also for individuals. Attackers are constantly improving their techniques by making their attacks stealthier and harder to detect for current state-of-the-art SIEMs. The main challenges reside in handling the large amount of data required to correlate the attack events due to the long running nature of APTs. In this thesis, we demonstrated the effectiveness of tag-based techniques, not only in attack detection and forensic analysis but also on how it can be used to speed up the process.

At first, we presented different techniques that allows for compact in-memory dependence graph generation. These techniques are capable of reducing the number of events by a factor of 7 without compromising dependency information. We also demonstrated how to massively speed up the forensic analyses process by converting the timestamped provenance graph to a compact versioned graph.

Next, we developed three different tag-semantics for accurate real-time attack detection and forensic analyses. SLEUTH is capable of detecting and generating the attack graph of any file-based attacks in real-time using COTS audit logs. By simply dividing the trustworthiness tag into code and data trustworthiness, SLEUTH is capable of reducing the number of false positives and also the size of the attack graph by a factor of 100 to over 1000.

To deal with increasing stealthiness of APT attacks and control the dependence explosion problem, we introduce two techniques called *tag attenuation* and *tag decay* in MORSE. By modulating the tag propagation using the subject's code tag on benign subjects, we demonstrated that our approach is highly effective in *automatic detection* of stealthy APT-style campaigns in *real-time*. Our techniques cut down false alarms by over an order of magnitude, while yielding compact scenario graphs that were smaller by a factor of 35x on average. Due to the conservative nature of tag propagation for malicious process, attackers are unable to evade our detection system.

Lastly, we introduce the semantic of probabilistic tags on confidentiality. As attackers are moving away from malware based attacks and are mainly carrying out

their attack using stolen credentials and system tools, recognizing the *reconnaissance* and *data leak* steps have become crucial in detecting current APTs. By using our automated confidentiality inference system and accumulation technique our system is capable of detecting every single reconnaissance and data leak in multiple datasets with high precision and recall.

In the future, we plan on incorporating probabilistic tags to the subject behavior to detect anomalous processes. We call this tag *subject benignity tag* and is intended to capture the probability of a process being malicious. To update this tag based on the behavior of the subject, we employ a number of detectors of suspicious behavior. These detectors trigger alarms. Some alarms are very reliable indicators of compromise, so they warrant a significant decrease in the subject benignity. Others are more ambiguous and may result in small or negligible decreases.

To facilitate the specification of rules for updating benignity tags, we divide it into three components, a) parent tag (value inherited from parent) b) child tag (component accumulated from its children) and c) behavior tag (derived from the process behavior). We combine these components into the benignity tag to classify if the process is malicious. Our technique for updating behavior tags relies on a training phase. We perform frequency based training using different time-window to generate distributions of the alarms triggered in benign data sets. If during detection we see significant deviation from the benign distribution we lower the behavior tag of the process.

Probabilistic view of subject tags also allows for incorporating outside anomaly detectors in our system as well. Probabilistic subject benignity tags in combination with probabilistic confidentiality tags could massively improve the detection of long running and stealthy APTs.

Bibliography

[1] Actions Taken by Equifax and Federal Agencies in Response to the 2017 Breach. https://www.gao.gov/assets/700/694158.pdf.

[2] APT Notes. https://github.com/kbandla/APTnotes. Accessed: 2016-11-10.

[3] DARPA transparent computing engagement 3 and 5 data release. https://github.com/darpa-i2o/Transparent-Computing/. Accessed: 2021-12-30.

[4] FreeBSD DTrace. https://wiki.freebsd.org/DTrace/. Accessed: 2019-5-1.

[5] IBM QRadar SIEM. https://www.ibm.com/us-en/marketplace/ibm-qradar-siem.

[6] IBM X-Force Threat Intelligence Index. https://www.ibm.com/security/data-breach/threat-intelligence. Accessed: 2019-3-7.

[7] Logrhythm, the security intelligence company. https://logrhythm.com/.

[8] MANDIANT: Exposing One of China's Cyber Espionage Units. https://www.fireeye.com/content/dam/fireeye-www/services/pdfs/mandiant-apt1-report.pdf. Accessed: 2016-11-10.

[9] Micro Focus ArcSight ESM. https://www.microfocus.com/en-us/products/siem-security-information-event-management/overview.

[10] The opm data breach: How the government jeopardized our national security for more than a generation. https://oversight.house.gov/report/opm-data-breach-government-jeopardized-national-security-generation/.

[11] SIEM, AIOps, Application Management, Log Management, Mach ine Learning, and Compliance. https://www.splunk.com/.

[12] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[13] Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security*, 2010.

[14] Akritidis, Costa, Castro, and Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX security*, 2009.

[15] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. May 2008.

[16] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z Berkay Celik, Xiangyu Zhang, and Dongyan Xu. {ATLAS}: A sequence-based learning approach for attack investigation. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[17] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

[18] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 2014.

[19] Zhifeng Bao, Henning Köhler, Liwei Wang, Xiaofang Zhou, and Shazia Sadiq. Efficient provenance storage for relational queries. In *CIKM*, 2012.

[20] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. Washington, DC, October 2003.

[21] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security*, 2015.

[22] Adam Bates, Dave (Jing) Tian, Grant Hernandez, Thomas Moyer, Kevin R. B. Butler, and Trent Jaeger. Taming the costs of trustworthy provenance through policy reduction. *ACM Trans. Internet Technol.*, 2017.

[23] Y. Bejtlich. *The Tao of Network Security Monitoring Beyond Intrusion Detection*. Pearson Education, 2004.

[24] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[25] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. pages 158–168, Ottawa, Canada, June 2006.

[26] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *IEEE Security and Privacy*, 2006.

[27] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *USENIX Security Symposium*, 2003.

[28] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[29] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[30] K. J. Biba. Integrity Considerations for Secure Computer Systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.

[31] Prithvi Bisht and V.N. Venkatakrishnan. XSS-Guard: Precise Dynamic Detection of Cross-Site Scripting Attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, 2008.

[32] T Bowen, D Chee, M Segal, R Sekar, P Uppuluri, and T Shanbag. Building survivable systems: An integrated approach based on intrusion detection and confinement. In *Darpa Information Security Symposium*, 2000.

[33] Uri Braun, Simson Garfinkel, David A Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I Seltzer. Issues in automatic provenance collection. In *Provenance and Annotation Workshop*, 2006.

[34] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[35] L. Cavallaro and R. Sekar. Taint-enhanced anomaly detection. In *International Conference on Information Systems Security*, 2011.

[36] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. Technical report, Secure Systems Lab at Stony Brook University, 2007.

[37] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *ACM SIGMOD*, 2008.

[38] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Distributed provenance compression. In *ACM SIGMOD*, 2017.

[39] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing*, page 409, Phoenix, Arizona, April 2001.

[40] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[41] Hervé Debar and Andreas Wespi. Aggregation and correlation of intrusion-detection alerts. In *RAID*. Springer, 2001.

[42] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, 1987.

[43] Dhurjati and Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN*, 2006.

[44] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*. ACM, 2005.

[45] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.

[46] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *IEEE Security and Privacy*, 2003.

[47] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas Longstaff. A sense of self for unix processes. In *IEEE Security and Privacy*, 1996.

[48] Debin Gao, Michael K Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *ACM CCS*, 2004.

[49] Debin Gao, Michael K. Reiter, and Dawn Song. On gray-box program tracking for anomaly detection. pages 103–118, San Diego, CA, USA, August 2004.

[50] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. SAQL: A stream-based query system for real-time abnormal system behavior detection. In *USENIX Security Symposium*, 2018.

[51] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. AIQL: Enabling efficient attack investigation from system monitoring data. In *USENIX ATC*, 2018.

[52] Ashish Gehani and Dawood Tariq. Spade: support for provenance auditing in distributed environments. In *International Middleware Conference*, 2012.

[53] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.

[54] Ashvin Goel, W-C Feng, David Maier, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. In *25th IEEE International Conference on Distributed computing systems workshops*, 2005.

[55] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *SOSP*, 2005.

[56] Guofei Gu, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, and Wenke Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *USENIX Security Symposium*, 2007.

[57] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*, 2020.

[58] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Code Generation and Optimization*, 2012.

[59] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1172–1189. IEEE, 2020.

[60] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*, 2019.

[61] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott Stoller, and V.N. Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security*, 2017.

[62] Md Nahid Hossain, Sanaz Sheikhi, and R. Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[63] Md Nahid Hossain, Junao Wang, R Sekar, and Scott D Stoller. Dependence preserving data compaction for scalable forensic analysis. In *USENIX Security*, 2018.

[64] IBM X-Force threat intelligence index. `https://www.ibm.com/downloads/cas/M1X3B7QG`. Accessed: 2021-12-08.

[65] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Fazzini Mattia, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM CCS*, 2017.

[66] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security*, 2018.

[67] Jones and Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Workshop on Automated Debugging*, 1997.

[68] Klaus Julisch. Clustering intrusion detection alarms to support root cause analysis. *Transactions on Information and System Security (TISSEC)*, 2003.

[69] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. pages 272–280, Washington, DC, October 2003.

[70] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. *SIGPLAN Not.*, 2012.

[71] Kil, Jun, Bookholt, Xu, and Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *ACSAC*, 2006.

[72] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *SOSP*, 2003.

[73] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[74] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, 1994. IEEE Computer Society Press.

[75] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Security and Privacy*, 1997.

[76] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX Security*, 2009.

[77] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *SOSP*. ACM, 2007.

[78] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. Baltimore, MD, August 2005.

[79] Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna. *Intrusion detection and correlation: challenges and solutions*. Springer Science & Business Media, 2005.

[80] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *ACM CCS*, 2003.

[81] S. Kumar and E. Spafford. A pattern-matching model for intrusion detection. In *National Computer Security Conference*, 1994.

[82] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses, Morgan-Claypool and ACM Press*, 2018.

[83] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. *ASPLOS*, 2016.

[84] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. Mci: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

[85] Lap Chung Lam and Tzi-cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2004.

[86] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[87] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage collecting audit log. In *ACM CCS*, 2013.

[88] Wenke Lee, Salvatore J Stolfo, and Kui W Mok. A data mining framework for building intrusion detection models. In *IEEE Security and Privacy*, 1999.

[89] Lixin Li, Jim Just, and R. Sekar. Address-space randomization for windows systems. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[90] Ninghui Li, Ziqing Mao, and Hong Chen. Usable Mandatory Integrity Protection for Operating Systems . In *S&P*. IEEE, 2007.

[91] Zhenkai Liang, Weiqing Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. In *ACM TISSEC*, 2009.

[92] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[93] Lockheed Martin. The cyber kill chain. `https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html`. Accessed: 2021-12-08.

[94] Peter Loscocco and Stephen Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Ottawa Linux Symposium*, 2001.

[95] Teresa F Lunt, Ann Tamaru, and F Gillham. *A real-time intrusion-detection expert system (IDES)*. SRI International. Computer Science Laboratory, 1992.

[96] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *ACSAC*, 2015.

[97] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security*, 2017.

[98] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[99] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1035–1044, 2016.

[100] Ziqing Mao, Ninghui Li, Hong Chen, and Xuxian Jiang. Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. In *Transactions on Information and System Security (TISSEC)*. ACM, 2011.

[101] the PaX team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt, 2001.

[102] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. In *PPAA*, 2014.

[103] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Propatrol: Attack investigation via extracted high-level tasks. In *In International Conference on Information Systems Security, Springer*, 2018.

[104] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *ACM CCS*, 2019.

[105] Sadegh M. Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V.N. Venkatakrishnan. HOLMES: Real-time APT Detection through Correlation of Suspicious Information Flows. In *IEEE Security and Privacy*, 2019.

[106] MITRE Corporation. Adversary Tactics and Techniques Knowledge Base (ATT&CK). `https://attack.mitre.org/`. Accessed: 2019-03-04.

[107] Kiran-Kumar Muniswamy-Reddy and David A Holland. Causality-based versioning. *ACM Transactions on Storage (TOS)*, 2009.

[108] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *USENIX ATC*, 2006.

[109] Kiran-Kumar Muniswamy-Reddy, Charles P Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *USENIX FAST*, 2004.

[110] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 2009.

[111] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *ACM SIGMOD*, 2008.

[112] Necula, Condit, Harren, McPeak, and Weimer. CCured: type-safe retrofitting of legacy software. *ACM TOPLAS*, 2005.

[113] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[114] Peng Ning, Yun Cui, and Douglas S Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *ACM CCS*, 2002.

[115] Peng Ning and Dingbang Xu. Learning attack strategies from intrusion alerts. In *ACM CCS*, 2003.

[116] Ben Niu and Gang Tan. Modular control-flow integrity. In *PLDI*, 2014.

[117] Steven Noel, Eric Robertson, and Sushil Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *Annual Computer Security Applications Conference*, 2004.

[118] Chetan Parampalli, R Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 156–167. ACM, 2008.

[119] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *SoCC*, 2017.

[120] Patil and Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software — Practice & Experience*, 1997.

[121] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: Attack story reconstruction via community discovery on correlated log graph. In *ACSAC*, 2016.

[122] R. Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *ASIACCS*, 2012.

[123] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-Fi: Collecting high-fidelity whole-system provenance. In *ACSAC*, 2012.

[124] P. Porras and R. Kemmerer. Penetration state transition analysis: A rule based intrusion detection approach. In *Annual Computer Security Applications Conference*, 1992.

[125] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *NDSS*, 2015.

[126] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. Practical fine-grained binary code randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[127] Rui Qiao, Mingwei Zhang, and R Sekar. A principled approach for rop defense. In *Annual Computer Security Applications Conference*, 2015.

[128] Xinzhou Qin and Wenke Lee. Statistical causality analysis of infosec alert data. In *RAID*, 2003.

[129] Douglas S Santry, Michael J Feeley, Norman C Hutchinson, Alistair C Veitch, Ross W Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP*, 1999.

[130] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Code generation and optimization*, 2008.

[131] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Network and Distributed System Security Symposium*, 2009.

[132] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based approach for detecting anomalous program behaviors. In *IEEE Security and Privacy*, 2001.

[133] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *National Information Systems Security Conference*, 1998.

[134] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, 1999.

[135] Xiaokui Shu, Frederico Araujo, Douglas L Schales, Marc Ph Stoecklin, Jiyong Jang, Heqing Huang, and Josyula R Rao. Threat intelligence computing. In *ACM CCS*, 2018.

[136] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *ACM CCS*, 2015.

[137] Craig A Soules, Garth R Goodson, John D Strunk, and Gregory R Ganger. Metadata efficiency in a comprehensive versioning file system. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2002.

[138] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS*, 2004.

[139] Weiqing Sun, R. Sekar, Zhenkai Liang, and V. N. Venkatakrishnan. Expanding malware defense by securing software installations. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[140] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *IEEE Security and Privacy*, 2008.

[141] Wai Kit Sze, Bhuvan Mital, and R Sekar. Towards more usable information flow policies for contemporary operating systems. In *ACM SACMAT*, 2014.

[142] Wai-Kit Sze and R Sekar. A portable user-level approach for system-wide integrity protection. In *ACSAC*, 2013.

[143] Wai Kit Sze and R Sekar. Provenance-based integrity protection for windows. In *ACSAC*, 2015.

[144] Laszlo Szekeres, Mathias Payer, Tao Wei, and R Sekar. Eternal war in memory. *S&P Magazine*, 2014.

[145] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *ACM SIGMOD*, 2008.

[146] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security*, 2014.

[147] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS*, 2018.

[148] Prem Uppuluri and R Sekar. Experiences with specification based intrusion detection. In *Recent Advances in Intrusion Detection*, 2001.

[149] V. N. Venkatakrishnan, Peri Ram, and R. Sekar. Empowering mobile code using expressive security policies. In *New Security Paradigms Workshop*, 2002.

[150] G. Vigna and R. Kemmerer. Netstat: A network-based intrusion detection approach. In *Computer Security Applications Conference*, 1998.

[151] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.

[152] David Wagner and Drew Dean. Intrusion detection via static analysis. In *IEEE Security and Privacy*, 2001.

[153] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM CCS*, 2002.

[154] Wei Wang and Thomas E Daniels. A graph based approach toward network forensics analysis. *ACM Transactions on Information and System Security (TISSEC)*, 2008.

[155] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Security and Privacy*, 1999.

[156] Wikipedia. Ccleaner. `https://en.wikipedia.org/wiki/CCleaner`. Accessed: 2019-03-28.

[157] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell D.E. Long. A hybrid approach for efficient provenance storage. In *CIKM*, 2012.

[158] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. Florence, Italy, October 2003.

[159] Wei Xu, Sandeep Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, May 2005.

[160] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[161] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Foundations of software engineering*, 2004.

[162] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM CCS*, 2016.

[163] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: an efficient pointer arithmetic checker for C programs. In *ASIACCS*, 2010.

[164] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *OSDI*. USENIX, 2006.

[165] Yan Zhai, Peng Ning, and Jun Xu. Integrating ids alert correlation and os-level dependency tracking. In *International Conference on Intelligence and Security Informatics*, 2006.

[166] Chao Zhang, Chengyu Song, Z. Kevin Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting virtual function tables' integrity. In *NDSS*, 2015.

[167] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Security and Privacy*, 2013.

[168] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security*, 2013.

[169] Mingwei Zhang and R Sekar. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *ACSAC*, 2015.

[170] Ningning Zhu and Tzi-cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Dependable Systems and Networks*, 2003.