
# SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data

**Md Nahid Hossain,** *Stony Brook University;* **Sadegh M. Milajerdi,** *University of Illinois at Chicago;* **Junao Wang,** *Stony Brook University;* **Birhanu Eshete and Rigel Gjomemo,** *University of Illinois at Chicago;* **R. Sekar and Scott Stoller,** *Stony Brook University;* **V.N. Venkatakrishnan,** *University of Illinois at Chicago*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hossain

**This paper is included in the Proceedings of the 26th USENIX Security Symposium**

**August 16–18, 2017 • Vancouver, BC, Canada**

ISBN 978-1-931971-40-9

**Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX**

# SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data*

Md Nahid Hossain[1], Sadegh M. Milajerdi[2], Junao Wang[1], Birhanu Eshete[2], Rigel Gjomemo[2], R. Sekar[1], Scott D. Stoller[1], and V.N. Venkatakrishnan[2]

[1]Stony Brook University
[2]University of Illinois at Chicago

## Abstract

We present an approach and system for real-time reconstruction of attack scenarios on an enterprise host. 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. We then 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. Our system participated in a red team evaluation organized by DARPA and was able to successfully detect and reconstruct the details of the red team's attacks on hosts running Windows, FreeBSD and Linux.

## 1 Introduction

We are witnessing a rapid escalation in targeted cyber-attacks ("Enterprise Advanced and Persistent Threats (APTs)") [1] conducted by skilled adversaries. By combining social engineering techniques (e.g., spear-phishing) with advanced exploit techniques, these adversaries routinely bypass widely-deployed software protections such as ASLR, DEP and sandboxes. As a result, enterprises have come to rely increasingly on second-line defenses, e.g., intrusion detection systems (IDS), security information and event management (SIEM) tools, identity and access management tools, and application firewalls. While these tools are generally useful, they typically generate a vast amount of information, making it difficult for a security analyst to distinguish truly significant attacks — the proverbial "needle-in-a-haystack" — from background noise. Moreover, analysts lack the tools to "connect the dots," i.e., piece together fragments of an attack campaign that span multiple applications or hosts and extend over a long time period. Instead, significant manual effort and expertise are needed to piece together numerous alarms emitted by multiple security tools. Consequently, many attack campaigns are missed for weeks or even months [7, 40].

In order to effectively contain advanced attack campaigns, analysts need a new generation of tools that not only assist with detection but also produce a compact summary of the causal chains that summarize an attack. Such a summary would enable an analyst to quickly ascertain whether there is a significant intrusion, understand how the attacker initially breached security, and determine the impact of the attack.

The problem of piecing together the causal chain of events leading to an attack was first explored in Backtracker [25, 26]. Subsequent research [31, 37] improved on the precision of the dependency chains constructed by Backtracker. However, these works operate in a purely forensic setting and therefore do not deal with the challenge of performing the analysis in real-time. In contrast, this paper presents SLEUTH,[1] a system that can alert analysts in real-time about an ongoing campaign, and provide them with a compact, visual summary of the activity in seconds or minutes after the attack. This would enable a timely response before enormous damage is inflicted on the victim enterprise.

Real-time attack detection and scenario reconstruction poses the following additional challenges over a purely forensic analysis:

1. *Event storage and analysis:* How can we store the millions of records from event streams efficiently and have algorithms sift through this data in a matter of seconds?

*This work was primarily supported by DARPA (contract FA8650-15-C-7561) and in part by NSF (CNS-1319137, CNS-1421893, CNS-1514472 and DGE-1069311) and ONR (N00014-15-1-2208 and N00014-15-1-2378). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

[1]SLEUTH stands for (attack) Scenario LinkagE Using provenance Tracking of Host audit data.

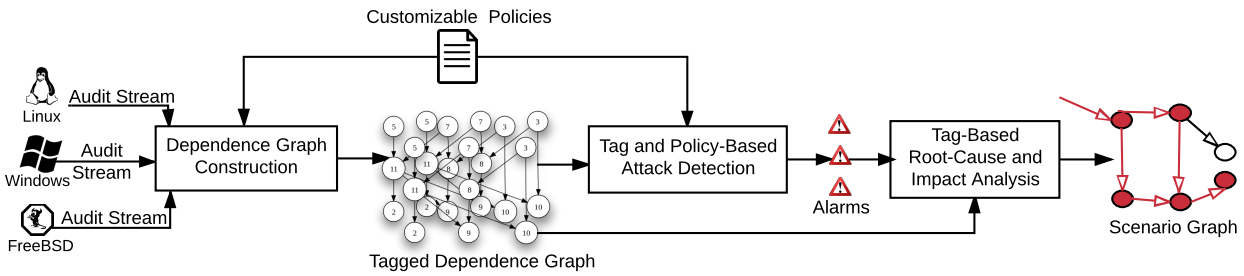


**Fig. 1:** SLEUTH System Overview

2. *Prioritizing entities for analysis:* How can we assist the analyst, who is overwhelmed with the volume of data, prioritize and quickly "zoom in" on the most likely attack scenario?

3. *Scenario reconstruction:* How do we succinctly summarize the attack scenario, starting from the attacker's entry point and identifying the impact of the entire campaign on the system?

4. *Dealing with common usage scenarios:* How does one cope with normal, benign activities that may resemble activities commonly observed during attacks, e.g., software downloads?

5. *Fast, interactive reasoning:* How can we provide the analyst with the ability to efficiently reason through the data, say, with an alternate hypothesis?

Below, we provide a brief overview of SLEUTH, and summarize our contributions. SLEUTH assumes that attacks initially come from outside the enterprise. For example, an adversary could start the attack by hijacking a web browser through externally supplied malicious input, by plugging in an infected USB memory stick, or by supplying a zero-day exploit to a network server running within the enterprise. We assume that the adversary *has not* implanted persistent malware on the host *before* SLEUTH started monitoring the system. We also assume that the OS kernel and audit systems are trustworthy.

### 1.1 Approach Overview and Contributions

Figure 1 provides an overview of our approach. 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.

The first contribution of this paper, which addresses the challenge of efficient event storage and analysis, is the development of a compact main-memory dependence graph representation (Section 2). 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.

The second major contribution of this paper is the development of 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* 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 [42, 58], if they are available. Tags are described in more detail in Section 3, together with their application to attack detection.

A third contribution of this paper is the development of novel algorithms that leverage tags for root-cause identification and impact analysis (Section 5). Starting from alerts produced by the attack detection component shown in Fig. 1, 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. 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.

The fourth contribution of this paper, aimed at tackling the last two challenges mentioned above, is a customizable policy framework (Section 4) 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 6.6 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.

The final contribution of this paper is an experimental evaluation (Section 6), 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 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.

## 2 Main Memory Dependency Graph

To support fast detection and real-time analysis, we store dependencies in a graph data structure. One possible option for storing this graph is a graph database. However, the performance [39] of popular databases such as Neo4J [4] or Titan [6] is limited for many graph algorithms unless main memory is large enough to hold most of data. Moreover, the memory use of general graph databases is too high for our problem. Even STINGER [16] and NetworkX [5], two graph databases optimized for main-memory performance, use about 250 bytes and 3KB, respectively, per graph edge [39]. The number of audit events reported on enterprise networks can easily range in billions to tens of billions per day, which will require main memory in the range of several terabytes. In contrast, we present a much more space-efficient dependence graph design that uses only about 10 bytes per edge. In one experiment, we were able to store 38M events in just 329MB of main memory.

The dependency graph is a per-host data structure. It can reference entities on other hosts but is optimized for the common case of intra-host reference. The graph represents two types of entities: *subjects*, which represent processes, and *objects*, which represent entities such as files, pipes, and network connections. Subject attributes include process id (pid), command line, owner, and tags for code and data. Objects attributes include name, type (file, pipe, socket, etc.), owner, and tags.

Events reported in the audit log are captured using labeled edges between subjects and objects or between two subjects. For brevity, we use UNIX names such as `read`, `connect`, and `execve` for events.

We have developed a number of techniques to reduce storage requirements for the dependence graph. Wherever possible, we use 32-bit identifiers instead of 64-bit pointers. This allows a single host's dependence graph to contain 4 billion objects and subjects. The number of objects/subjects in our largest data set was a few orders of magnitude smaller than this number.

While our design emphasizes compact data structures for objects and subjects, compactness of events is far more important: events outnumber objects and subjects by about two orders of magnitude in our largest data set. Moreover, the ratio of events to objects+subjects increases with time. For this reason, we have developed an ultra-compact representation for events that can use as little as 6 bytes of storage for many events.

Events are stored inside subjects, thereby eliminating a need for subject-to-event pointers, or the need for event identifiers. Their representation uses variable-length encoding, so that in the typical case, they can use just 4 bytes of storage, but when needed, they can use 8, 12, or 16 bytes. Most events operate on an object and have a timestamp. Since a per-subject order of events is maintained, we dispense with microsecond granularity for timestamps, instead opting for millisecond resolution. In addition, we store only relative time since the last event on the same subject, which allows us to do with 16-bits for the timestamp in the typical case[2]. Objects are referenced within events using an index into a per-subject table of object identifiers. These indices can be thought of like file descriptors — they tend to have small val-

[2]Longer intervals are supported by recording a special "timegap" event that can represent millions of years.

ues, since most subjects use a relatively small number of objects. This enables object references to be represented using 8 bits or less. We encode event names for frequently occurring events (e.g., open, close, read and write) using 3 bits or less. This leaves us with several bits for storing a summary of event argument information, while still being within 32 bits.

We can navigate from subjects to objects using the event data stored within subjects. However, forensic analysis also requires us to navigate from objects to subjects. For this purpose, we need to maintain event information within objects using object-event records. Object event records are maintained only for a subset of events: specifically, events such as `read` and `write` that result in a dataflow. Other events (e.g., `open`) are not stored within objects. Object-event records are further shrunk by storing a reference to the corresponding subject-event record, instead of duplicating information.

As with subject-event records, we use a variable-length encoding for object-event records that enables them to be stored in just 16 bits in the most common case. To see how this is possible, note that objects tend to be operated on by a single subject at a time. Typically, this subject performs a sequence of operations on the object, e.g., an open followed by a few reads or writes, and then a close. By allowing object-event records to reuse the subject from their predecessor, we can avoid the need for storing subject identifiers in most records. Next, we allow object-event records to store a relative index for event records within subjects. Two successive event records within a subject that operate on the same object are likely to be relatively close to each other, say, with tens or hundreds of events in-between. This means that the relative index stored with object-event record can be 12 bits or less in most cases, thus allowing these records to be 16 bits or less in the typical case.

This design thus allows us to store bidirectional timestamped edges in as little as 6 bytes (4 bytes for a subject-event record and 2 bytes for an object-event record). In experiments with larger data sets, the total memory use of our system is within 10 bytes per event on average.

Our variable length encoding allows us to represent full information about important (but rare) events, such as rename, chmod, execve, and so on. So, compactness is achieved without losing any important information. Although such encoding slows down access, access times are still typically less than 100ns, which is many orders of magnitude faster than disk latencies that dominate random access on disk-resident data structures.

# 3 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, 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.

## 3.1 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.

### 3.2 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 [1, 2, 3]:

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[3].
- *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`[4,5].

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

[4] 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.

[5] Our implementation can identify `mprotect` operations that occur

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

## 4 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.,

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

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. Attributes of interest include:

in conjunction with library loading operations. This policy is not triggered on those `mprotect`'s.

| Event | Direction | Alarm trigger | Tag trigger |
|---|---|---|---|
| `define` | | | *init* |
| `read` | O→S | *read* | *propRd* |
| `load`, `execve` | O→S | *exec* | *propEx* |
| `write` | S→O | *write* | *propWr* |
| `rm`, `rename` | S→O | *write* | |
| `chmod`, `chown` | S→O | *write*, *modify* | |
| `setuid` | S→S | | *propSu* |

**Table 2:** Edges with policy trigger points. In the direction column, S indicates subject, and O indicates object. The next two columns indicate trigger points for detection policies and tag setting policies.

- *name:* regular expressions can be used to match object names and subject command lines. We use Perl syntax for regular expressions.
- *tags:* conditions can be placed on t-tags and c-tags of objects and/or subjects. For subjects, code and data t-tags can be independently accessed.
- *ownership and permission:* conditions can be placed on the ownership of objects and subjects, or permissions associated with the object or 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. While we use a rule-based notation to specify policies in this paper, in our implementation, each rule is encoded as a (C++) function.

To provide a finer degree of control over the order in which different types of policies are checked, we associate policies with *trigger points* instead of events. In addition, trigger points provide a level of indirection that enables sharing of policies across distinct events that have a similar purpose. Table 2 shows the trigger points currently defined in our policy framework. The first column identifies events, the second column specifies the direction of information flow, and the last two columns define the trigger points associated with these events.

Note that we use a special event called `define` to denote audit records that define a new object. This pseudo-event is assumed to have occurred when a new object is encountered for the first time, e.g., establishment of a new network connection, the first mention of a pre-existing file, creation of a new file, etc. The remaining events in the table are self-explanatory.

When an event occurs, all detection policies associated with its alarm trigger are executed. Unless specifically configured, detection policies are checked only when the tag of the target subject or object is about to change. ("Target" here refers to the destination of data flow in an operation.) Following this, policies associated with the event's tag triggers are tried in the order in which they are specified. As soon as a matching rule is found, the

tags specified by this rule are assigned to the target of the event, and the remaining tag policies are not evaluated.

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.

### 4.1 Tag Initialization Policies

These policies are invoked at the *init* trigger, and are used 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, this can be overridden using tag initialization policies.

Our current tag initialization policy is as follows. Note the use of regular expressions to conveniently define initial tags for groups of objects.

$init(o)$: $match(o.name, \texttt{"\^IP:(10\.0|127)"}) \rightarrow$
$o.ttag = \texttt{BENIGN\_AUTH}, o.ctag = \texttt{PRIVATE}$
$init(o)$: $match(o.name, \texttt{"\^IP:"}) \rightarrow$
$o.ttag = \texttt{UNKNOWN}, o.ctag = \texttt{PRIVATE}$
$init(o)$: $o.type == \texttt{FILE} \rightarrow$
$o.ttag = \texttt{BENIGN\_AUTH}, o.ctag = \texttt{PUBLIC}$

The first rule specifies tags for intranet connections, identified by address prefixes 10.0 and 127 for the remote host. It is useful in a context where SLEUTH isn't deployed on the remote host[6]. The second rule states that all other hosts are untrusted. All preexisting files are assigned the same tags by the third rule. Our implementation uses two additional policies that specify c-tags.

### 4.2 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, as indicated in the "Tag trigger" column in Table 2.

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. The following policy skips taint propagation for this specific file:

$propRd(s,o)$: $match(o.name, \texttt{"\textbackslash.bash\_history\$"}) \rightarrow skip$[7]

Here is a policy that treats files read by `bash`, which is an interpreter, as a load, and hence updates the code t-tag.

$propRd(s,o)$: $match(s.cmdline, \texttt{"\^/bin/bash\$"}) \rightarrow$
$s.code\_ttag = s.data\_ttag = o.ttag, s.ctag = o.ctag$

Although trusted servers such as `sshd` interact with untrusted sites, they can be expected to protect themselves, and let only authorized users access the system. Such servers should not have their data trustworthiness downgraded. A similar comment applies to programs such as software updaters and installers that download code from untrusted sites, but verify the signature of a trusted software provider before the install.

$propRd(o,s)$: $match(s.cmdline, \texttt{"\^/sbin/sshd\$"}) \rightarrow skip$

Moreover, when the login phase is complete, typically identified by execution of a `setuid` operation, the process should be assigned appropriate tags.

$propSu(s)$: $match(s.cmdline, \texttt{"\^/usr/sbin/sshd\$"}) \rightarrow$
$s.code\_ttag = s.data\_ttag = \texttt{BENIGN}, s.ctag = \texttt{PRIVATE}$

## 5 Tag-Based Bi-Directional Analysis

### 5.1 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 ir-

[6] If SLEUTH is deployed on the remote host, there will be no `define` event associated with the establishment of a network connection, and hence this policy won't be triggered. Instead, we will already have computed a tag for the remote network endpoint, which will now propagate to any local subject that reads from the connection.

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

relevant nodes and sets SLEUTH apart from other scenario reconstruction approaches such as [25, 31].

Based on this insight, we formulate backward analyis 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 chalenge by by preferring the entry point closest (as measured by path cost) to a suspect node.

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.

### 5.2 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[8]. 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.

### 5.3 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.

## 6 Experimental Evaluation

### 6.1 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 of about 9.5KLoC. The remaining components, including that for reconstruction and presentation, are implemented in Python, and consist of 1.6KLoC.

### 6.2 Data Sets

Table 3 summarizes the dataset used in our evaluation. The first eight rows of the table correspond to attack cam-

[8] 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.

| 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. 15 |
| W-2 | 19:43:46 | N/A | 17.40% | 47.63% | 8.03% | 3.28% | 3.26% | 15.22% | 5.17% | 401K | Fig. 5 |
| L-1 | 07:59:26 | 37% | 0.11% | 18.01% | 1.15% | 0.92% | 38.76% | 3.97% | 0.07% | 2.68M | Fig. 12 |
| 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. 16 |
| F-1 | 08:17:30 | 9.46% | 0.40% | 24.65% | 40.86% | 2.10% | 12.55% | 9.08% | 0.89% | 701K | Fig. 13 |
| F-2 | 78:56:48 | 11.78% | 0.42% | 16.60% | 44.52% | 2.10% | 15.04% | 8.54% | 1.01% | 5.86M | Fig. 14 |
| 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 |
| 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 3:** Dataset for each campaign with duration, distribution of different system calls and total number of events.

paigns 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 [31]), 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 3 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.

### 6.3 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 (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 paper 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 7. Below, we first present reconstructed scenarios for selected datasets before proceeding to a discussion of these summary results.

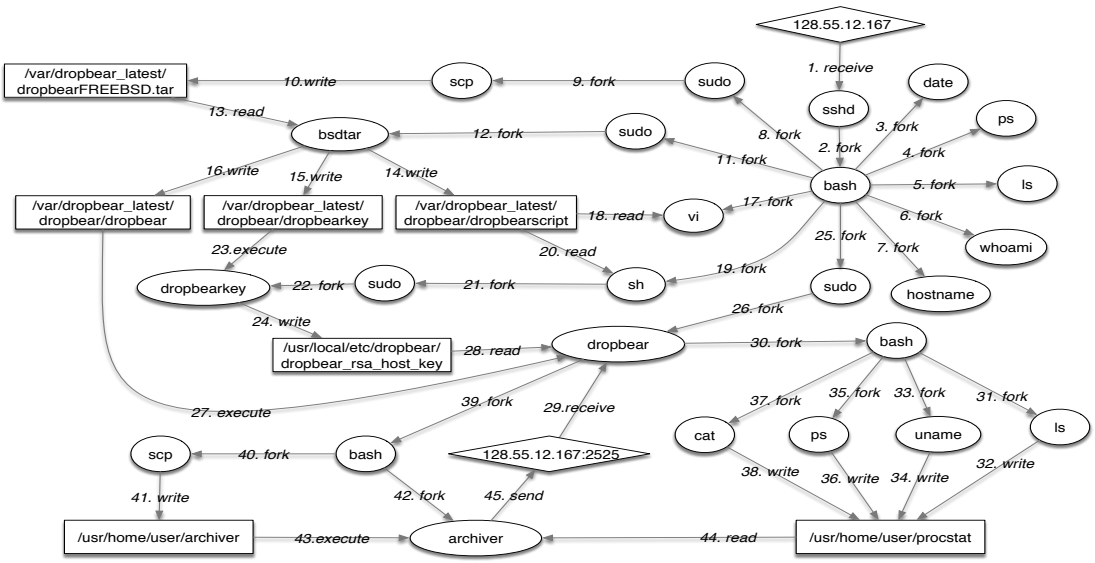


**Fig. 4:** Scenario graph reconstructed from campaign F-3.

## 6.4 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 6.10. 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.

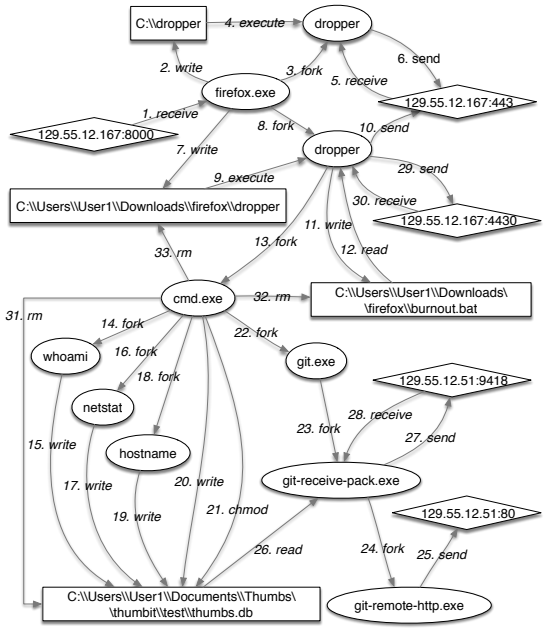


**Fig. 5:** Scenario graph reconstructed from campaign W-2.

**Campaign W-2.** Figure 5 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). 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.

## 6.5 Overall Effectiveness

To assess the effectiveness of SLEUTH in capturing essential stages of an APT, in Table 6, we correlate pieces of attack scenarios constructed by SLEUTH with APT stages documented in postmortem reports of notable APT campaigns (e.g., the MANDIANT [3] 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 | 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 6:** SLEUTH results with respect to a typical APT campaign.

| 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 7:** Attack scenario reconstruction summary.

Table 7 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 7, 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 7. 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 7 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

| 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 8:** False alarms in a benign environment with software upgrades and updates. No alerts were triggered during this period.

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.

## 6.6 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 8.

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.

## 6.7 Runtime and Memory Use

Table 9 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.

| 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 9:** Memory use and runtime for scenario reconstruction.

The third column shows the memory used for the dependence graph. As described in Section 2, we have designed a main memory representation that is very compact. This compact representation 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 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.

## 6.8 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

| 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 10:** 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.

the size of forward analysis output.

Table 10 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 11 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.

## 6.9 Analysis Selectivity

Table 11 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.

| 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 11:** Comparison of selectivity achieved using forward analysis with single trustworthiness tags, forward analysis with split code and data trustworthiness tags, and finally simplifications.

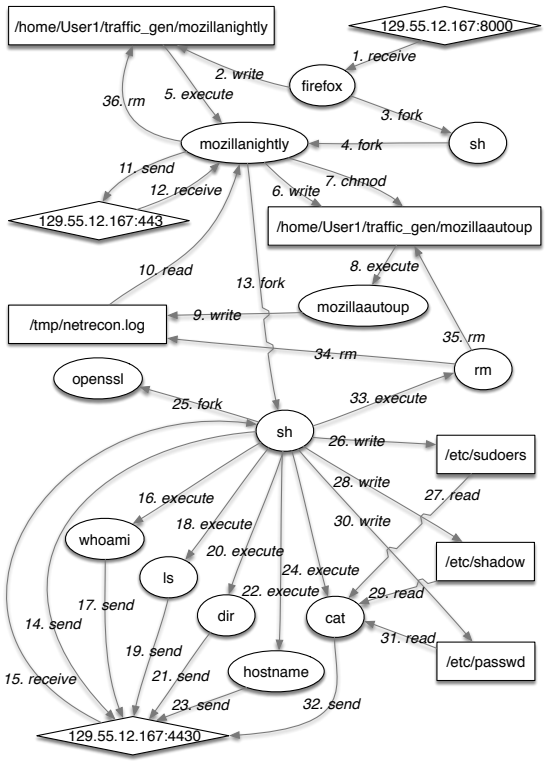


**Fig. 12:** Scenario graph reconstructed from campaign L-1.

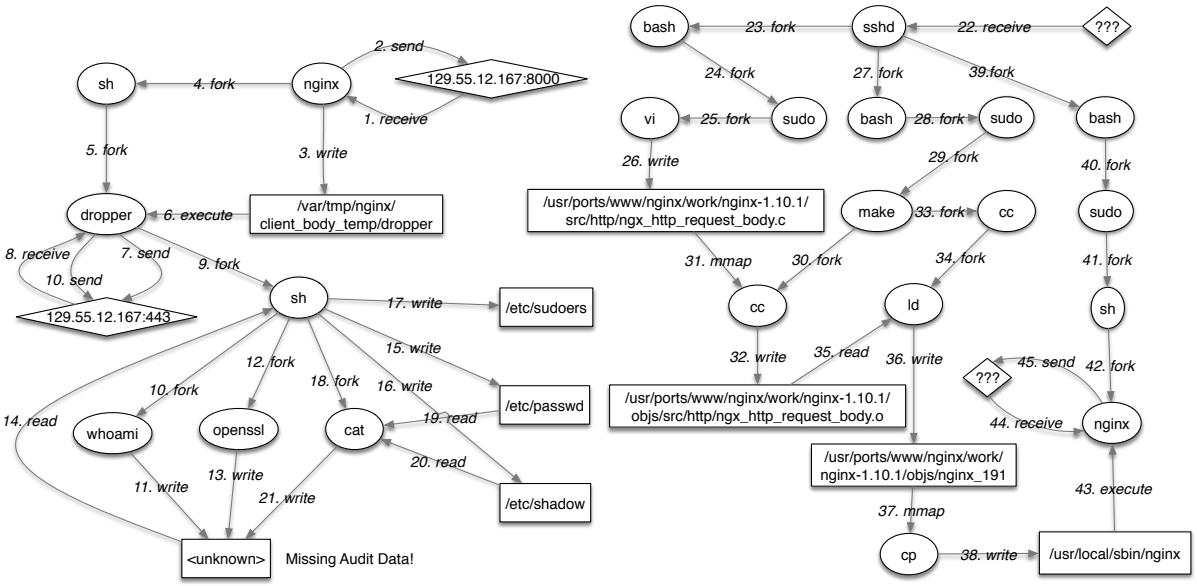


**Fig. 13:** Scenario graph reconstructed from campaign F-1.

### 6.10 Discussion of Additional Attacks

In this section, we provide graphs that reconstruct attack campaigns that weren't discussed in Section 6.4. Specifically, we discuss attacks L-1, F-1, F-2, W-1, and L-3. **Attack L-1.** In this attack (Figure 12), `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 13), 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 6.5.

**Attack F-2.** The start of this attack (Figure 14) 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 15), `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 16), 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.

## 7 Related Work

In this section, we compare SLEUTH with efforts from academia and open source industry tools. We omit comparison to proprietary products from the industry as there is scarce technical documentation available for an in-depth comparison.

**Provenance tracking and Forensics** Several logging and provenance tracking systems have been built to monitor the activities of a system [21, 41, 23, 22, 13, 45, 9] and build *provenance graphs*. Among these, *Backtracker* [25, 26] is one of the first works that used dependence graphs to trace back to the root causes of intrusions. These graphs are built by correlating events collected by a logging system and by determining the causality among system entities, to help in forensic analysis after an attack is detected.

SLEUTH improves on the techniques of Backtracker in two important ways. First, Backtracker was meant to operate in a forensic setting, whereas our analysis and data representation techniques are designed towards real-time detection. Setting aside hardware comparisons, we note that Bactracker took 3 hours for analyzing au-

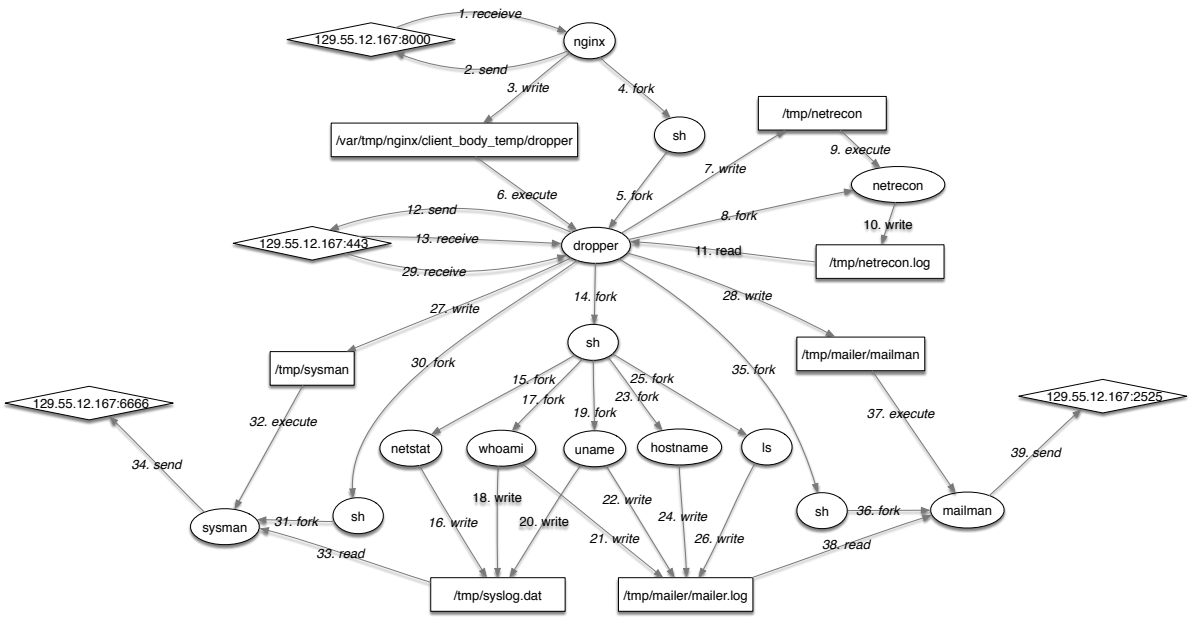


**Fig. 14:** Scenario graph reconstructed from campaign F-2.

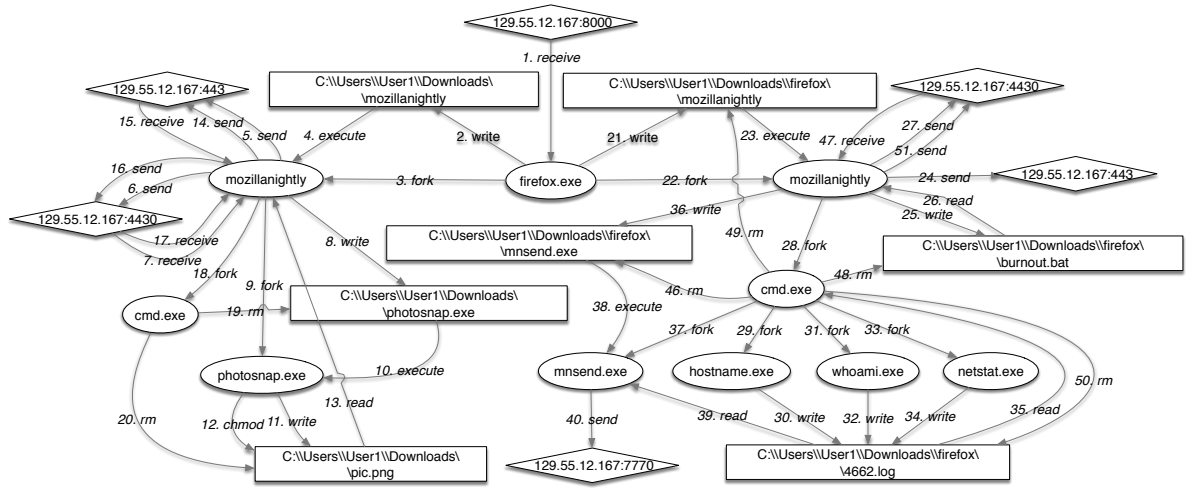


**Fig. 15:** Scenario graph reconstructed from campaign W-1.

dit data from a 24-hour period, whereas SLEUTH was able to process 358 hours of logs in a little less than 3 minutes. Secondly, Backtracker relies on alarms generated by external tools, therefore its forensic search and pruning cannot leverage the reasons that generated those alarms. In contrast, our analysis procedures leverage the results from our principled tag-based detection methods and therefore are inherently more precise. For example, if an attack deliberately writes into a well-known log file, Backtracker's search heuristics may remove the log file from the final graph, whereas our tag-based analysis will prevent that node from being pruned away.

In a similar spirit, *BEEP* [31] and its evolution *ProTracer* [37] build dependence graphs that are used for forensic analysis. In contrast, SLEUTH builds dependence graphs for real-time detection from which scenario subgraphs are extracted during a forensic analysis. The forensic analysis of [31, 37] ensures more precision than Backtracker [25] by heuristically dividing the execution of the program into execution units, where each unit represents one iteration of the main loop in the program. The instrumentation required to produce units is not always automated, making the scalability of their approach a challenge. SLEUTH can make use of the additional precision afforded by [31] in real-time detection, when such information is available.

While the majority of the aforementioned systems operate at the system call level, several other systems track information flows at finer granularities [24, 8, 31]. They typically instrument applications (e.g., using Pin [35]) to

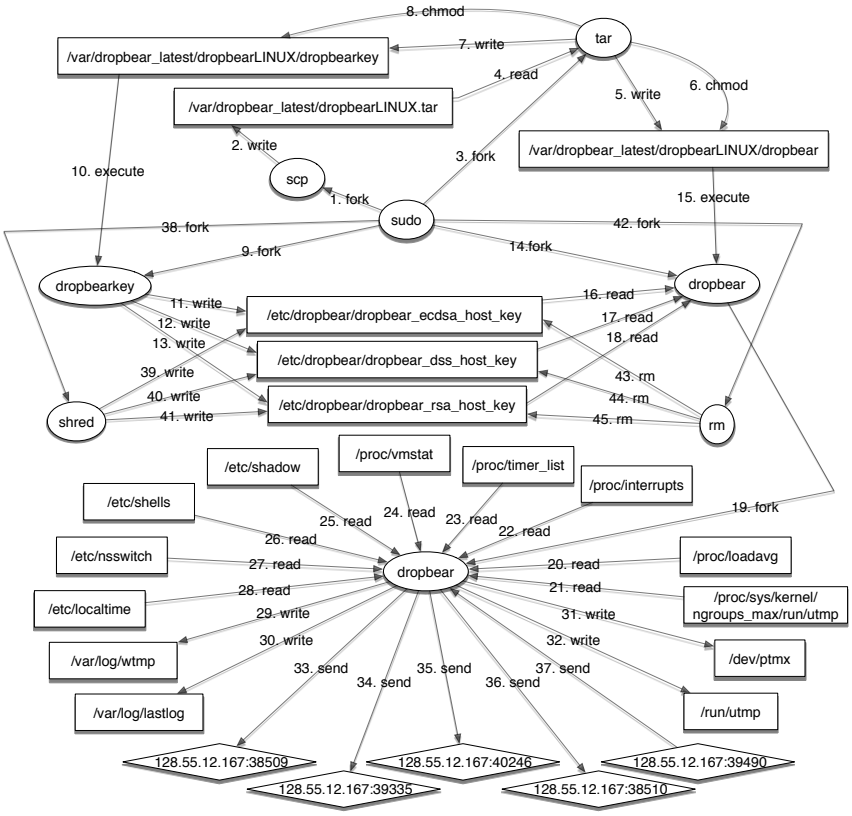


**Fig. 16:** Scenario graph reconstructed from campaign L-3.

track information flows through a program. Such fine-grained tainting can provide much more precise provenance information, at the cost of higher overhead. Our approach can take advantage of finer granularity provenance, when available, to further improve accuracy.

**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., address space randomization, control-flow integrity, taint-based defenses and so on. Offline *intrusion detection* using logs has been studied for a much longer period [15, 36, 19]. In particular, *host-based IDS* using system-call monitoring and/or audit logs has been investigated by numerous research efforts [57, 32, 47, 55, 18, 29].

Host-based intrusion detection techniques mainly fall into three categories: (1) *misuse-based*, which rely on specifications of bad behaviors associated with known attacks; (2) *anomaly-based* [19, 32, 47, 20, 30, 11, 48], which rely on learning a model of benign behavior and detecting deviations from this behavior; and (3) *specification-based* [27, 54], which rely on specifications (or policies) specified by an expert. The main drawback of misuse-based techniques is that their signature-based approach is not amenable to detection of previously unseen attacks. Anomaly detection techniques avoid this drawback, but their false positives rates deter widespread deployment. Specification/policy-based techniques can reduce these false positives, but they require application-specific policies that are time-consuming to develop and/or rely on expert knowledge. Unlike these approaches, SLEUTH relies on *application-independent policies.* We develop such policies by exploiting provenance information computed from audit data. In particular, an audit event

**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 [10] and Biba [12], relied on strict policies. These strict policies impact usability and hence have not found favor among contemporary OSes. Although IFC is available in SELinux [34], 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 [59, 17, 28] 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 par-

ticular, 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 [50, 38, 33, 53, 51, 52, 49] 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.

**Alert Correlation** Network IDSs often produce myriad alerts. *Alert correlation* analyzes relationships among alerts, to help users deal with the deluge. The main approaches, often used together, are to *cluster* similar alerts, prioritize alerts, and identify causal relationships between alerts [14, 43, 46, 44, 56]. Furthermore, they require manually supplied expert knowledge about dependencies between alert types (e.g., consequences for each network IDS alert type) to identify causal relationships. In contrast, we are not interested in clustering/statistical techniques to aggregate alerts. Instead, our goals are to use provenance tracking to determine causal relationships between different alarms to reconstruct the attack scenario, and to do so without relying on (application-dependent) expert knowledge.

## 8 Conclusion

We presented an approach and a system called SLEUTH for real-time detection of attacks and attack reconstruction from COTS audit logs. SLEUTH uses a main memory graph data model and a rich tag-based policy framework that make its analysis both efficient and precise. We evaluated SLEUTH on large datasets from 3 major OSes under attack by an independent red team, efficiently reconstructing all the attacks with very few errors.

## References


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

[2] Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. http://www.lockheedmartin.com/content/dam/lockheed/data/corporate/documents/LM-White-Paper-Intel-Driven-Defense.pdf. Accessed: 2016-11-10.

[3] 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.

[4] Neo4j graph database. https://neo4j.com/.

[5] Network-x graph database. https://networkx.github.io/.

[6] Titan graph database. http://titan.thinkaurelius.com/.

[7] Chloe Albanesius. Target Ignored Data Breach Warning Signs. http://www.pcmag.com/article2/0,2817,2454977,00.asp, 2014. [Online; accessed 16-February-2017].

[8] 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.

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

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

[11] Konstantin Berlin, David Slater, and Joshua Saxe. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, 2015.

[12] 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.

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

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

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

[16] David Ediger, Robert McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC)*. IEEE, 2012.

[17] 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.

[18] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *S&P*. IEEE, 2003.

[19] Stephanie Forrest, Steven Hofmeyr, Aniln Somayaji, Thomas Longstaff, et al. A sense of self for unix processes. In *S&P*. IEEE, 1996.

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

[21] Ashish Gehani and Dawood Tariq. Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*. Springer, 2012.

[22] A. Goel, W. C. Feng, D. Maier, W. C. Feng, and J. Walpole. Forensix: a robust, high-performance reconstruction system. In *25th IEEE International Conference on Distributed Computing Systems Workshops*, 2005.

[23] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. *SIGOPS Oper. Syst. Rev.*, 2005.

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

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

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

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

[28] 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.

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

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

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

[32] Wenke Lee, Salvatore J Stolfo, and Kui W Mok. A data mining framework for building intrusion detection models. In *S&P*. IEEE, 1999.

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

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

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, and Artur Klauser et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

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

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

[38] 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.

[39] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications*. ACM, 2014.

[40] Stephanie Mlot. Neiman Marcus Hackers Set Off Nearly 60K Alarms. http://www.pcmag.com/article2/0,2817,2453873,00.asp, 2014. [Online; accessed 16-February-2017].

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

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

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

[44] Steven Noel, Eric Robertson, and Sushil Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *ACSAC*. IEEE, 2004.

[45] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-fi: collecting high-fidelity whole-system provenance. In *ACSAC*. ACM, 2012.

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

[47] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *S&P*. IEEE, 2001.

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

[49] Weiqing Sun, R Sekar, Zhenkai Liang, and VN Venkatakrishnan. Expanding malware defense by securing software installations. In *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA)*. Springer, 2008.

[50] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *S&P*. IEEE, 2008.

[51] Wai Kit Sze, Bhuvan Mital, and R Sekar. Towards more usable information flow policies for contemporary operating systems. In *Proceedings of the 19th ACM symposium on Access control models and technologies*, 2014.

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

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

[54] Prem Uppuluri and R Sekar. Experiences with specification-based intrusion detection. In *RAID*. Springer, 2001.

[55] David Wagner and Drew Dean. Intrusion detection via static analysis. In *S&P*. IEEE, 2001.

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

[57] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *S&P*. IEEE, 1999.

[58] 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.

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