Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics

Md Nahid Hossain Sanaz Sheikhi R. Sekar
{mdnhossain,ssheikhi,sekar}@cs.stonybrook.edu
Stony Brook University, Stony Brook, NY, USA.

***Abstract*—We are witnessing a rapid escalation in targeted cyber-attacks called Advanced and Persistent Threats (APTs). Carried out by skilled adversaries, these attacks take place over extended time periods, and remain undetected for months. A common approach for retracing the attacker's steps is to start with one or more suspicious events from system logs, and perform a dependence analysis to uncover the rest of attacker's actions. The accuracy of this analysis suffers from the *dependence explosion* problem, which causes a very large number of benign events to be flagged as part of the attack. In this paper, 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, 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.**

I. Introduction

There has been a spate of targeted high-profile attacks on many large enterprises, including Target [13], Equifax [1], Deloitte [11] and the US Office of Personnel Management (OPM) [9]. Often called "Advanced and Persistent Threats" (APTs) [2], these are sophisticated attack campaigns carried out by skilled attackers. For instance, the OPM breach lasted more than 8 months, and leaked highly sensitive information used in security clearance of 21.5M individuals [9]. The Equifax breach went undetected for 2.5 months, and compromised sensitive information such as social security numbers of over 140M users.

Unlike traditional malware attacks that are indiscriminate, APTs choose their targets deliberately, e.g., high-value data such as intellectual property and financial, account or health information. They are able to bypass existing defenses using a combination of social engineering and advanced exploit techniques. Consequently, intrusion detection systems (IDS), often called Security Information and Event Management (SIEM) systems (e.g., IBM QRadar [5]), are the first to flag signs of these attacks. Unfortunately, today's IDS face many challenges against sophisticated APTs:

- *Needle-in-a-haystack:* Today's systems employ a wide range of IOCs ("indicators of compromise") that capture *isolated* indicators such as malware signatures, abnormal traffic volume, or modification of specific Windows registry keys. Unfortunately, many of these IOCs may be triggered by benign activities as well. Since attacks constitute relatively rare events among billions of benign background events, a flood of false positives results, making it extremely difficult to identify real attacks.
- *Connecting the dots:* Although existing systems can identify individual events that trigger IOCs, they don't provide much help in understanding the "big picture." To respond to sophisticated APTs in real-time, techniques are needed to connect seemingly disparate low-level events to uncover an 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 [89], [31]. Many detection and analysis tasks require graph searches that can be very expensive when operating on such large datasets.

Consequently, human analysts today find it very difficult to understand the progression of APT campaigns, causing them to remain undetected for months [13], [59], [1], [11], [9].

Provenance-based Detection and Forensics

Researchers have proposed the use of *provenance* to overcome some of the difficulties summarized above. For attack detection, provenance provides *additional context* to prune away false positives that result when events are considered in isolation. For instance, it is common for processes to execute scripts or load binary code during normal operation, but the very same actions may be used in attacks. Using provenance, also called *taintedness* in this line of research, it is possible to distinguish between benign behavior and attacks such as control-flow hijack, SQL injection and cross-site scripting [75], [62], [63], [86], [71], or malware execution [30].

For forensic analysis, *Backtracker* [36] used coarse-grained provenance from system-call logs to construct a dependence graph of system events. Given a system event corresponding to an attacker action, a backward graph search can be used to identify the potential entry point (e.g., IP address) of this attack. Then, a forward search from the entry point can be used to understand the totality of attacker's actions. These *backward* and *forward analyses* serve to connect together all of the attacker's steps, thus presenting the overall attack scenario to a cyber analyst.

Dependence Explosion

Most forensic analysis techniques [36], [26], [30], [51], [57] 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. Dependence

This work was supported primarily by DARPA (contract FA8650-15-C-7561), and in part by grants from NSF (CNS-1918667) and ONR (N00014-17-1-2891).

explosions can also occur due to files that are read and written by numerous processes, e.g., `.bash_history`. Coarse-grained provenance causes all readers of this file (i.e., all instances of `bash` shell processes) to be dependent on all writers (i.e., all previous `bash` processes). 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.

Fine-grained Provenance

Fine-grained information flow tracking [62], [87], [14], [35] can trace back specific output bytes of a process to specific input bytes consumed by it. As a result, a single output can often be associated with a single input, thereby avoiding the massive precision loss incurred by coarse-grained tracking.

A major challenge faced by fine-grained tracking techniques is their high performance overhead, ranging from 2x to 10x. But an even bigger problem is the need for extensive instrumentation of applications and/or OS code [62], [87], [14], [35], [53], [52], [32], [33]. Consequently, enterprises cannot rely on fine-grained provenance unless vendors ship their applications with such instrumentation. There is no indication that such a measure is under consideration by any vendor, so enterprises are limited to coarse-grained dependence for the foreseeable future. Indeed, coarse-grained provenance tracking is already built into today's OSes such as Linux (Linux auditing system) and Windows (ETW — Event Tracing for Windows).

Previous Mitigation Techniques and Limitations

Recognizing the difficulties of obtaining fine-grained provenance data, researchers have begun to develop alternative techniques to mitigate dependence explosion in COTS audit data. SLEUTH [30] associates a cost measure with each edge in the dependence graph, and prunes away higher-cost paths to arrive at compact *scenario graphs* summarizing attacker activity. While this approach is effective on fast moving attacks, we find that for long-running attacks, it can produce graphs with numerous benign nodes. HOLMES [57] exploits the multi-stage structure of typical APTs to generate a compact high-level scenario graph (HSG). However, it faces challenges on APTs that lack this structure, e.g., ransomware attacks.

PRIOTRACKER [51] prioritizes edges that represent anomalous dependencies (i.e., dependencies rarely been witnessed before) in order to speed up forensic investigations. NODOZE [29] generalizes the approach to target anomalous paths rather than individual edges. An important drawback of anomaly based approaches is the need for *representative* training data. Problems with training data can lead to false positives, false negatives, or both.

Even more important, both PRIOTRACKER and NODOZE aim to identify attacker activities that are usually performed by attacker-provided malware. They assume that these activities will be anomalous, and differ from benign background activity. Unfortunately, attackers have a great deal of control over the behavior of their malware, and can hence intentionally avoid unusual dependencies that trigger these systems.

We therefore propose a new approach that:

- avoids optimistic assumptions about malware behavior,
- avoids the need for any training data,
- significantly reduces dependence explosion, and
- yields compact scenario graphs, even for stealthy campaigns.

We have implemented our approach into a tool called MORSE.

A. Approach Overview and Summary of Contributions

We begin with a motivating example in Section II that illustrates the challenges in detecting and summarizing stealthy attack campaigns. We then introduce our techniques for mitigating dependence explosion in Section III. As is common with information flow techniques, we associate *tags* with data items, and propagate these tags along with the data. We identify two types of tags:

- *data tags* that capture the integrity and confidentiality aspects of a data item, and
- *subject tags* that are associated only with subjects, and indicate our level of suspicion that a particular subject is malicious.

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 IV, we present a policy-based attack detection approach that takes advantage of tag attenuation and decay to significantly reduce false positives. Our techniques for attack campaign reconstruction are described in Section V.

The implementation of MORSE is sketched in Section VI. We use the motivating example from Section II to illustrate its operation in Section VII. Our experimental evaluation (Section VIII) shows that MORSE is effective in detecting a range of stealthy APT-style campaigns, where some of the critical steps 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 VIII, followed by related work and conclusions in Sections IX and X.

II. 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

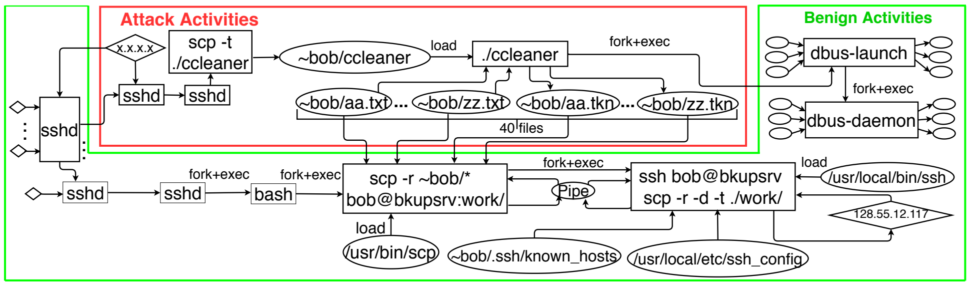


Fig. 1: Motivating example: CCleaner ransomware. Rectangles denote subjects (processes), while oval-shaped nodes denote files and diamonds denote network objects. Edges denote events, and are oriented in the direction of information flow. Edges without a specific event label denote reads and writes.

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 [85] 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. 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. 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. 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 malware 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 are even more formidable. Because of intermixing of benign and attack activities, a forward analysis starting at the `ccleaner` process yields a graph with tens of thousands of nodes, illustrating the effect of dependence explosion. Existing forensic analysis techniques that rely on coarse-grained provenance are ill-equipped to deal with this explosion. Specifically, SLEUTH's [30] approach of cost-based pruning of forward paths is 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.

HOLMES [57] copes with dependence explosion by looking for and linking together the steps in a typical APT life-cycle, including initial compromise, privilege escalation, internal recon, lateral movement, exfiltration and clean-up. Although HOLMES can cope with a few missing phases, `ccleaner` poses a stiff challenge, as it exercises just a single APT stage (initial compromise).

NODOZE [29] prioritizes anomalous information flows to reduce the size of the graph generated by forward analysis. Unfortunately, the flows manifested by malicious `ccleaner` program are similar to those of benign `ccleaner`. This can make it difficult to ensure that malware activities are faithfully captured in the output. Moreover, flows created by the backup process share similarities with the malware, making it difficult to prune out this benign background activity. PRIOTRACKER's [51] isn't directly aimed at constructing a compact scenario graph, but instead, it aims to include as much of attack activity as possible within a given amount of analysis time. Their use of event rareness can prioritize edges from `ccleaner` malware, but because many of the files read by the backup process are also new, those benign reads have to be prioritized as well.

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 I: 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 VIII.

Since NoDoze and PrioTracker rely on anomalous ("rare") events and/or anomalous information flows, sophisticated attackers can evade them by designing their malware to match the behaviors of benign applications. Moreover, these techniques require an external attack detector to initiate the analysis. *In contrast, we present a tag prioritization method that automates both attack detection and scenario graph construction,* and moreover, *operates in real-time.*

III. 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 II, Sections V.B and VII. 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.

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

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 I and II 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 I and II. 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.

Propagation Rules for Operations on Data

The first row in Table I corresponds to object creation. The object simply inherits the subject's data tag in all cases. Note, however

[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.

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

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 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 II 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 VII.) 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,

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 III: 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.

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.

IV. Provenance-Based Attack Detection

Provenance-based attack detection using system audit logs has been proposed before in the SLEUTH system [30], and its overall effectiveness demonstrated. Our key contribution in this paper 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 III 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 III abstract some of the essential steps of APT attacks [8], [2], [57], including the initial exploit, foothold establishment, privilege escalation, and exfiltration of sensitive data.

The first row of Table III 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.

V. Attack Scenario Reconstruction

The central goal of this paper 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.

A. 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.

B. 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.

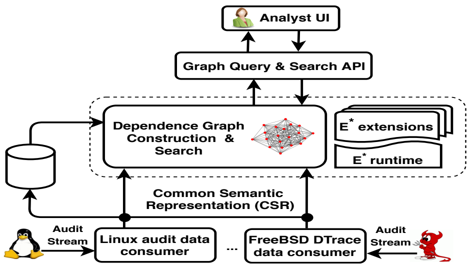
Next, a depth-first search of the dependence graph is initiated at the entry point node. This search does not visit nodes whose data integrity tag is above a set threshold (which defaults to 0.5, but may be changed by the analyst). This search identifies the nodes that will be included in the scenario 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.

VI. Implementation

As shown in Fig. 2, our system consists of three layers that implement its core functionality, together with an UI for an analyst. The lowest layer consists of data consumers that process input from COTS auditing systems. We used two consumers in our evaluation, one for Linux `auditd` and another for FreeBSD DTrace [4] data. These systems have the ability to log important system events, including most system calls. There are two variants of the Linux consumer. The first one directly inputs Linux audit logs, while the second version inputs roughly the same information, but in the Apache Avro format used by DARPA Transparent Computing dataset [3]. FreeBSD consumer supports only the second format. These consumers are written in C++ and consist of about 6KLoC. They translate OS-specific events into a platform-neutral set of operations provided by the middle layer. This platform-neutral representation can be stored on disk, in a format we call Common Semantic Representation (CSR) [31].

The middle layer can either operate from CSR files, or interface directly to the data consumers. It is responsible for constructing and traversing the dependence graph, and is implemented in 11 KLoC of C++. It builds on our earlier SLEUTH system, incorporating many further optimizations and refinements, including our dependency-preserving graph compaction technique [31]. Another major new feature of MORSE is its runtime environment, which exposes platform-neutral events to *extension modules*. These extensions are written in $E^*_,$ our domain-specific language for event monitoring and manipulation. All of the tag initialization, tag propagation and alarm policies are implemented in $E^*_.$ Tag propagation (Tables I and II) was realized using 103 lines of $E^*_,$ while the alarm rules (Table III) required 37 lines. The compiler and runtime environment for E^* consist of about 8KLoC of C++. Due to space constraints, we have omitted a detailed description of $E^*_.$

The third layer consists of a user interface for analysts to monitor alarms, run queries on the graph, construct scenario graphs, etc.



Analyst UI
Graph Query & Search API
Dependence Graph Construction & Search
E* extensions
E* runtime
Common Semantic Representation (CSR)
Audit Stream
Linux audit data consumer
...
FreeBSD DTrace data consumer
Audit Stream

Fig. 2: Implementation Architecture

VII. 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 II. The resulting graph, as seen by the analyst, is shown in Fig. 3. 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 I and II. 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 VIII) 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 II). Additionally, when a number of alarms can be traced back to a subject, that subject is marked suspicious (Section V.A).

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 III. The `ccleaner` process is also marked as a suspect subject by the $exec$ rule in Table II. As a result, its file overwrite (or remove) operations trigger the $Corrupt$ alarm as well.

While the policies shown in Table III 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 V traces back the above $FileExec$ and

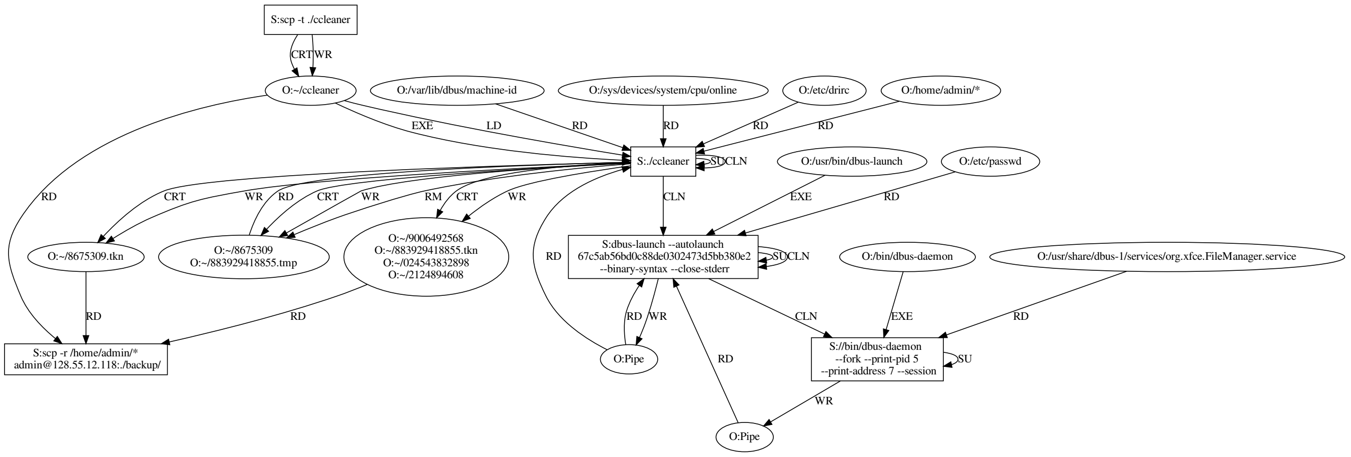


Fig. 3: Scenario Graph constructed by MORSE for CCleaner Ransomware

$Corrupt$ 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 II (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. 3.

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.

VIII. Experimental Evaluation

Platform. The system under attack consisted of multiple hosts running recent 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 [36], [30], [51], [57], [29], 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.

A. Dataset

Many previous works [36], [46], [53], [51], [29] 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.

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 IV.

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,

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. 7).
			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. 12).
			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. 8).
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. 13)
			CCleaner ransomware	*VNC attack:* Motivating example discussed in Section II (Fig. 3).
			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. 9).
			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. 10).
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. 11).

Table IV: 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.

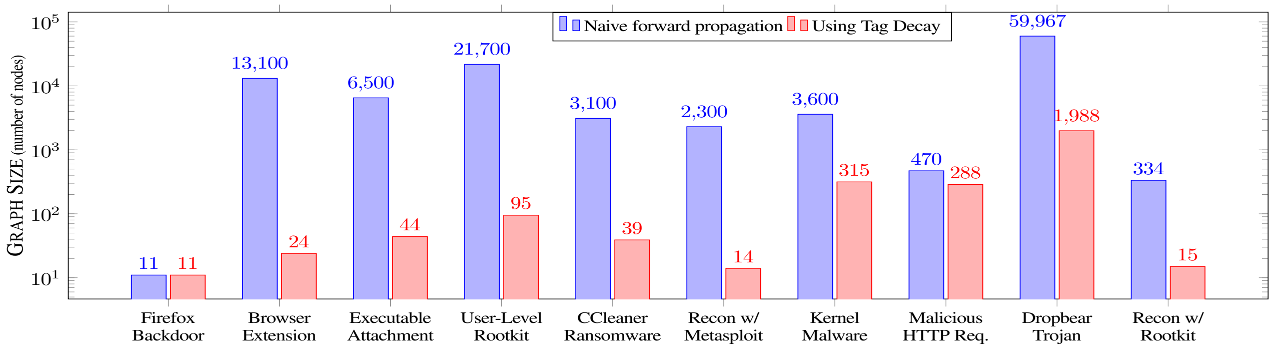


Fig. 4: Reduction in scenario graph size achieved using tag attenuation and decay. The average size reduction is 35 times, and no relevant nodes were dropped.

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 IV.

Data from DARPA TC Engagement 4

The L-4 and F-4 datasets shown in Table IV 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.

B. Effectiveness of Tag Attenuation and Decay

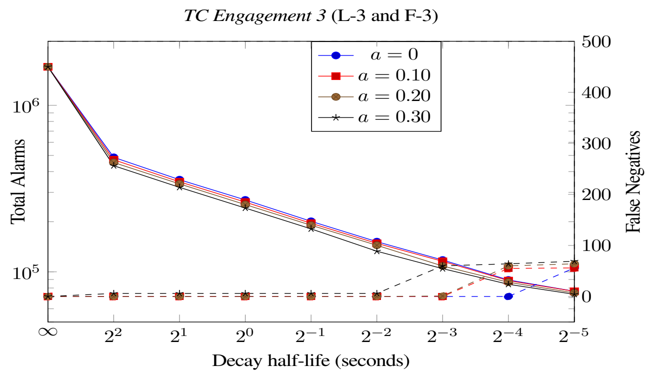
Parameter Selection. Our method is characterized by the rates of attenuation and decay for benign and suspect environment subjects. Values of these four parameters (a_b, d_b, a_e and d_e) can be chosen based on a high-level understanding of how they affect alarms. For instance, consider a benign subject 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 $a_b = 0.2$, 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.

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 V: 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 [30], while "Ours" show the reduction achieved by MORSE.



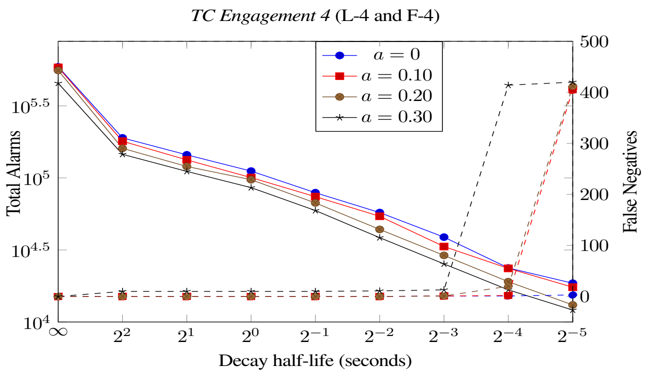


Fig. 5: Total number of alarms and false negatives on *TC Engagement 3* and *Engagement 4* datasets using different attenuation and decay rates. The scale for total number of alarms is on the left, while the false negative scale is on the right. The total number of true positives are 126 and 425. The 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.

Scenario Graph Size Reduction

Figure 4 summarizes the reduction in scenario graph sizes achieved using tag attenuation and decay. These graphs were generated as described in Section V.B: 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 [30], 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 V. 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 V. *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. 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. 4.

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 VI: 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 VII: Main memory size of dependence graphs.

C. Runtime Performance

Table VI 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* [31] that eliminates the need to store the vast majority of events, while guaranteeing accurate forensic analysis results. Table VII 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.

D. 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 II. 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^*.)

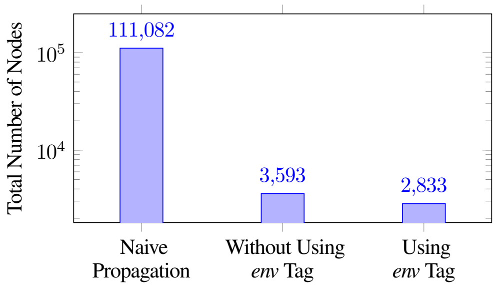


Fig. 6: Size of the scenario graph withou decay or attenuation, without using env tag (i.e., no decay or attenuation for suspect environment subjects) and using env tag.

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.

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

[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.

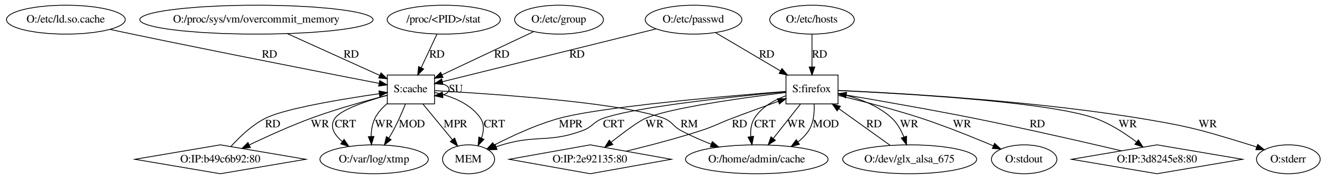


Fig. 7: ***Firefox Backdoor.*** Firefox was first compromised by a malicious ad server, resulting in an in-memory payload. This generated multiple $MemExec$ alarms. Next, an $Escalate$ alarm was triggered, as the attacker escalated privilege using a kernel implant. Installed prior to the engagement, this implant was accessed using the device */dev/glx_alsa_675*. 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.

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. 6.) 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 VII: 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 VI).

E. 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. 7.
- *Browser extension:* This attack exploited a vulnerable Firefox extension. Its scenario graph is shown in Fig. 12.
- *Malicious HTTP request:* The attacker tried compromising the *sshd* process on the FreeBSD system but failed. The scenario graph shown in Fig. 8 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 VII.
- *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. 9.
- *Kernel malware:* This attack uses pre-installed kernel malware for privilege escalation, and compromising an existing *sshd* process, as described in Fig. 10.

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,

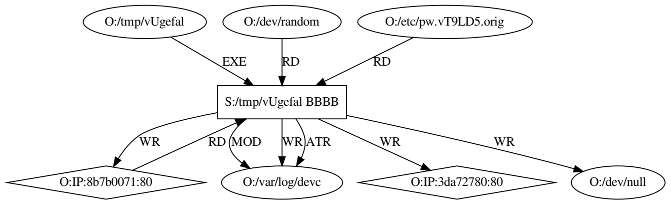


Fig. 8: ***Malicious HTTP Request.*** This figure shows one of the more successful attempts of this attack, which began with an exploit of *nginx*. A malicious file */tmp/vUgefal* was then downloaded and executed, raising a $FileExec$ alarm. The attacker went on to write another file */var/log/devc*, 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.

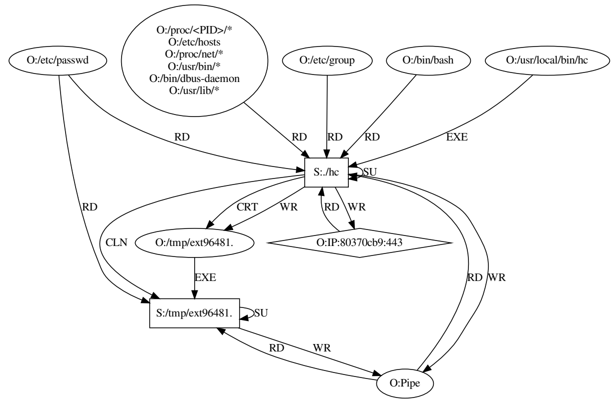


Fig. 9: ***Recon with Metasploit.*** This attack began with a malicious file *hc* that was `scp`'d onto the victim host using previously stolen credentials. When this file was executed, a $FileExec$ alarm was triggered. This process, together with another piece of downloaded malware */tmp/ext96481*, 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.

we propagate the data tags from the 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. 13 for additional details.
- *Recon with rootkit:* The F-4 attack in Fig. 11 is simpler, consisting of two instances of the same attack on two machines.

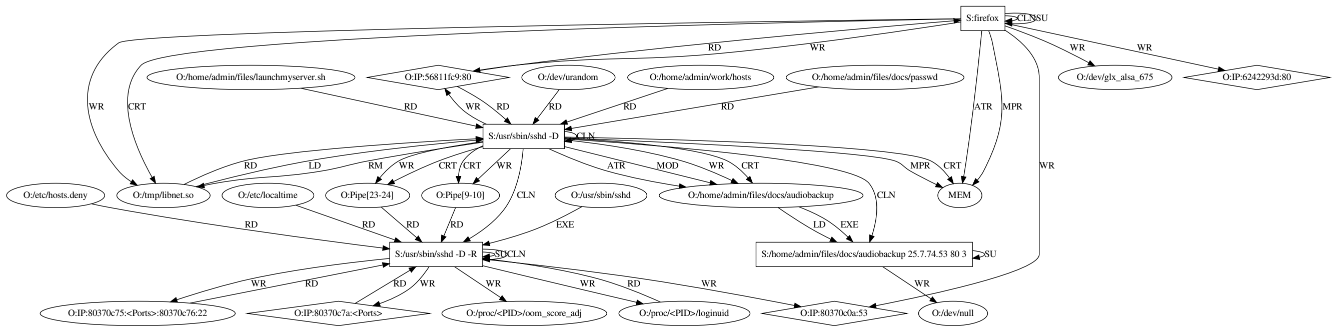


Fig. 10: Kernel Malware. *Firefox,* compromised by a malicious website, executed an in-memory payload that triggered several $MemExec$ alarms. Next, an $Escalate$ alarm was triggered, as the attacker escalated privilege using a kernel implant installed prior to the engagement. *Firefox* then downloaded 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.

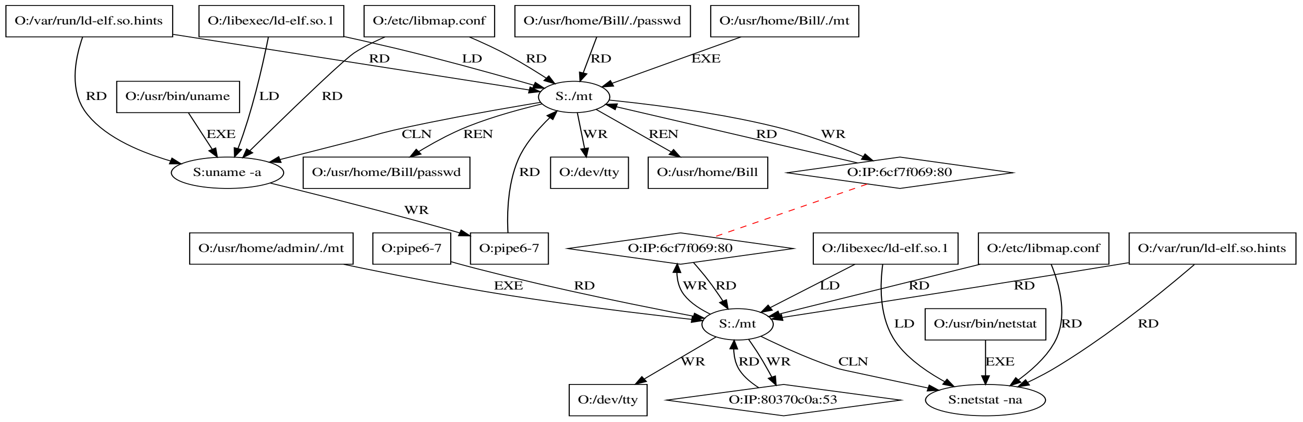


Fig. 11: Recon with Rootkit attack. This attack began with uploads of *mt,* a rootkit, to two FreeBSD hosts. When *mt* was executed, a $FileExec$ alarm was triggered. As *mt* gathered and exfiltrated sensitive information to an external IP address, $DataLeak$ alarms were raised. These alarms were clustered 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.

IX. Related Work

Fine-grained Taint-Tracking: Fine-grained taint tracking [62], [87], [14], [35], [44], [32], [33] 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 [17], 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 [46], PROTRACER [53] and MPI [52] developed a novel and efficient mechanism called *execution-partitioning,* targeting applications such as servers and web browsers that are prone to dependence explosion. MCI [45] and PROPATROL [55] 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.

Attack Detection: A number of research efforts on attack detection/prevention focus on "inline" techniques that are incorporated into the protected system, e.g., address space randomization [54], [16], [48], control-flow integrity [12], [91], memory safety [81], [88], [61], [28], [43], and so on. Unfortunately, attackers have repeatedly bypassed these techniques using a combination of social engineering and advanced exploit techniques. Enterprises have to rely on *intrusion detection systems* to piece together such attacks from system logs.

Intrusion detection techniques fall into three main categories: (i) *misuse detection* [69], [42], [83], [39], which relies on patterns of bad behaviors ("signatures") associated with known attacks; (ii) *anomaly detection* [20], [47], [72], [19], [21], [15], [41], [74], which relies on learning a model of benign behavior and detecting deviations from this behavior; and (iii) *specification-based detection* [38], [82], which relies on specifications of expected behaviors of applications.

Misuse-based techniques face challenges in detecting novel attacks since their signatures, by definition, are not available. Anomaly detection techniques can detect novel attacks, but they experience significant false positive rates that have deterred widespread deployment. 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.

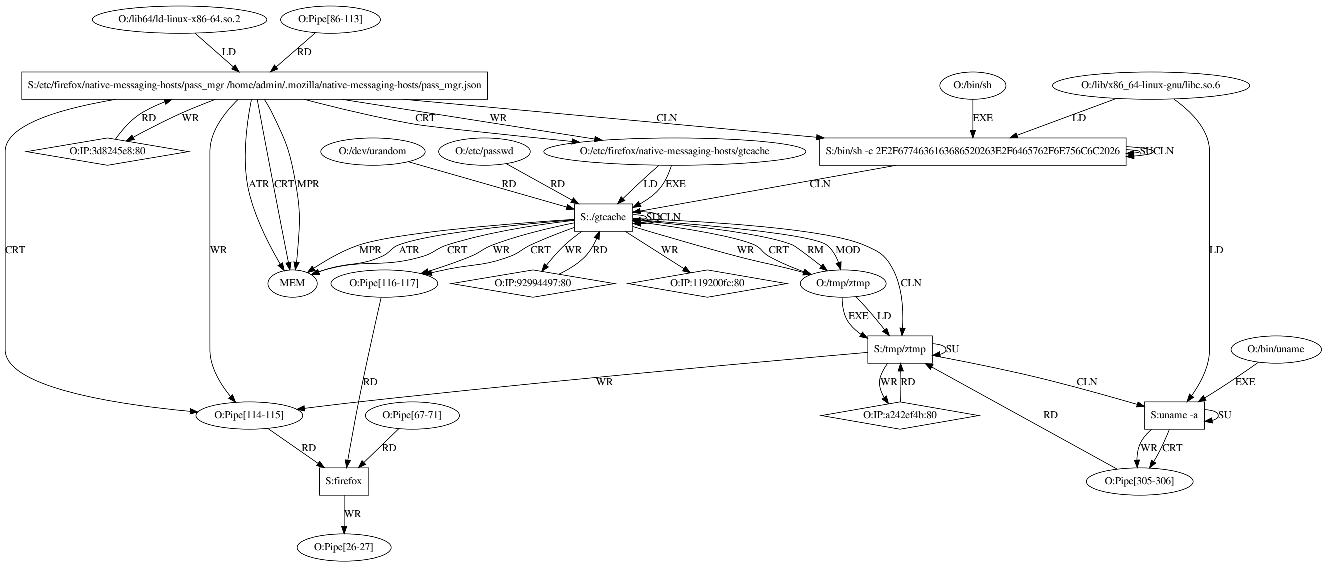


Fig. 12: Browser extension. The attack started when a vulnerable browser-plugin *pass_mgr* got compromised while visiting a malicious website. This raised $MemExec$ alarms. Next, the compromised plug-in downloaded a program *gtcache* and executed it, resulting in a $FileExec$ alarm. In turn, *gtcache* 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.

SLEUTH and MORSE policies can be thought of as specifications. As a result, they can hold down false positives while detecting unknown attacks, including those carried out during adversarial engagements. At the same time, they avoid the per-application development effort associated with previous specification-based techniques. We accomplish this by developing *application-independent policies* that exploit provenance. In particular, an audit event is analyzed to determine if it advances an attacker's high-level objectives, thereby providing a *motive* for the attack; while the provenance derived from the entire audit history is used to determine if the attacker had the *means* to influence this event. This combination of means (provenance) and motive (policies) has proved very successful in other contexts as well, such as the detection of memory corruption and injection attacks [86], [62], [71], securing untrusted code [50], [76], and OS-wide integrity protection [77], [49], [79], [80].

Unfortunately, dependence explosion dilutes the value of provenance in SLEUTH, resulting in numerous false positives on our dataset. MORSE cuts down these by an order of magnitude using tag attenuation and decay.

Alert Correlation: IDSs often produce numerous alerts. Alert correlation techniques combine related alerts into one, thus helping users deal with the deluge. The main approaches, often used together, are clustering of similar alerts, prioritization, and statistical correlation [18], [65], [70], [40], [34], [66], [84], [67]. Industry tools also use similar techniques in building SIEMs [7], [5], [10] for alert correlation and enforcement based on disparate logs.

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 [64], or models that capture typical progression of attacks [27]. For multi-stage attacks, provenance provides a more principled (and often, far more accurate) basis to correlate attack steps [36], [90]. For this reason, recent works have come to rely on provenance to correlate attack steps [78], [30], [57], [51], [29]. We discuss these techniques in more detail below.

Coarse-Grained Provenance Based Forensic Analysis: *Backtracker* [36] was the first to perform forensic intrusion investigation using dependence analysis of system-call logs. Other works on attack investigation [90], [26], [37] and provenance [60], [68], [24], [25] capture information flow at the coarse granularity of system calls. This invariably leads to the dependence explosion problem.

To mitigate dependence explosion, SLEUTH [30] uses split integrity tags (called *trustworthiness* tags in their terminology). *Code trustworthiness* tag captures the dependency of a subject's code (i.e., whether the code has a dependency on untrusted sources), while *data trustworthiness* captures the dependency of its data. By limiting its alarms and forensic analysis to follow subjects with untrusted code tag, it achieved orders of magnitude reduction in false alarms as well as scenario graph sizes on the simpler attack scenarios contained in DARPA TC Engagement 1 dataset. Unfortunately, the attacks in Engagements 3 and 4 were stealthier, leading SLEUTH to miss most attacks.

MORSE's subject tags are related to of SLEUTH's code trustworthiness tag. However, unlike SLEUTH, which simply forwards code tags from inputs to outputs, MORSE's subject tags can be thought of as *tag transformation functions.* This more general view enabled the development of tag attenuation and decay, and their selective application to benign subjects.

Unlike MORSE, HOLMES [57] aims for a much higher level summary of an APT campaign. Individual steps are recognized using a hybrid approach that combines SLEUTH-style detection policies with signatures based on MITRE's Adversarial Tactics, Techniques and Common Knowledge Base (ATT&CK) [58]. 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

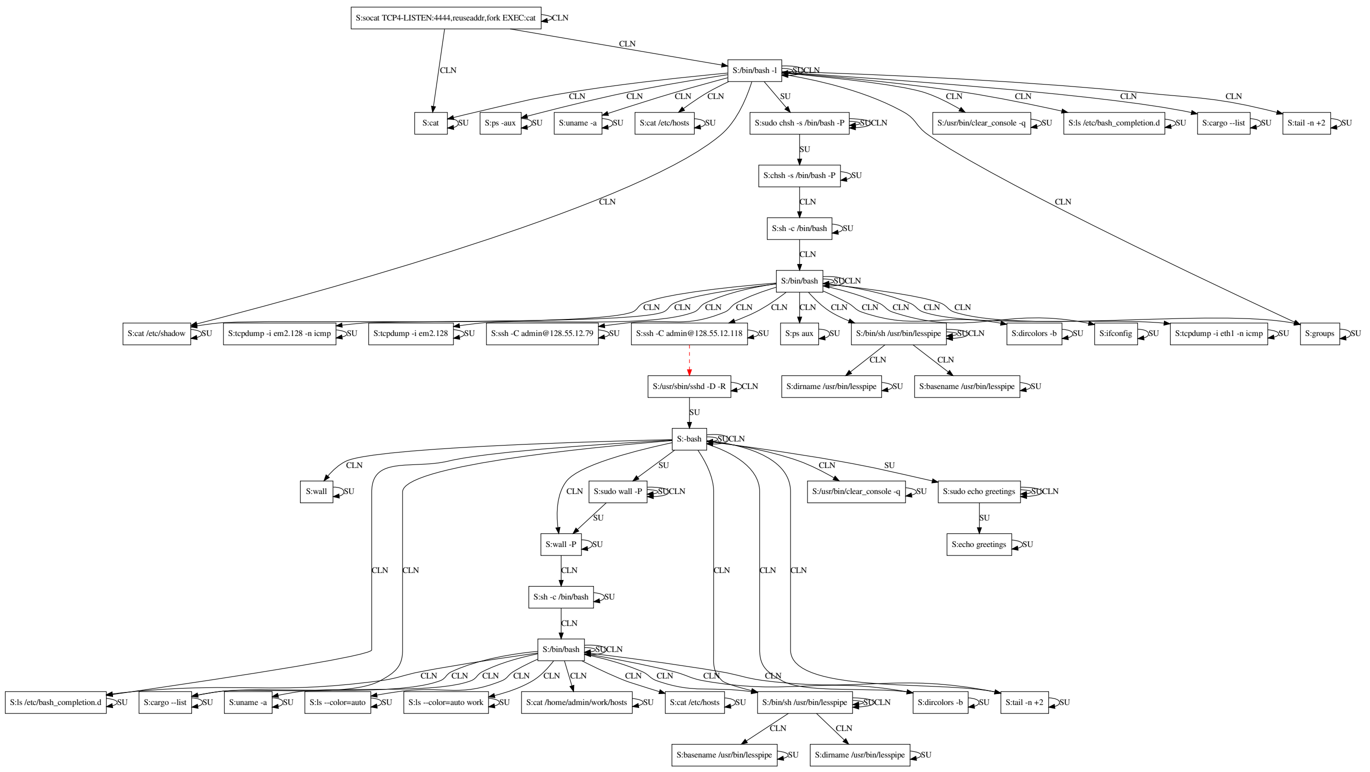

S:socat TCP4-LISTEN:4444,reuseaddr,fork EXEC:cat
S:/bin/bash -l
S:cat
S:ps -aux
S:uname -a
S:cat /etc/hosts
S:sudo chsh -s /bin/bash -P
S:/usr/bin/clear_console -q
S:ls /etc/bash_completion.d
S:cargo --list
S:tail -n +2
S:chsh -s /bin/bash -P
S:sh -c /bin/bash
S:/bin/bash
S:cat /etc/shadow
S:tcpdump -i em2.128 -n icmp
S:tcpdump -i em2.128
S:ssh -C admin@128.55.12.79
S:ssh -C admin@128.55.12.118
S:ps aux
S:/bin/sh /usr/bin/lesspipe
S:dircolors -b
S:ifconfig
S:tcpdump -i eth1 -n icmp
S:groups
S:/usr/sbin/sshd -D -R
S:dirname /usr/bin/lesspipe
S:basename /usr/bin/lesspipe
S:-bash
S:wall
S:sudo wall -P
S:/usr/bin/clear_console -q
S:sudo echo greetings
S:wall -P
S:echo greetings
S:sh -c /bin/bash
S:/bin/bash
S:ls /etc/bash_completion.d
S:cargo --list
S:uname -a
S:ls --color=auto
S:ls --color=auto work
S:cat /home/admin/work/hosts
S:cat /etc/hosts
S:/bin/sh /usr/bin/lesspipe
S:dircolors -b
S:tail -n +2
S:basename /usr/bin/lesspipe
S:dirname /usr/bin/lesspipe
CLN
SU
SUCLN

Fig. 13: 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.

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. Since the goals, the outputs, and datasets used differ across HOLMES and MORSE, a direct comaprison of their results is not meaningful.

PRIOTRACKER [51] speeds up forward analysis by using a prioritized graph exploration that assigns higher priority to edges representing unusual events. NODOZE [29] 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, as discussed before, attackers have a great deal of control over their malware, and can alter their behavior to blend in with benign background activity, as was the case with the *CCleaner* ransomware example. In contrast, we showed in Section VIII.D how MORSE resists such evasion.

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 dectectors deployed in the organization. Researchers have begun to build tools and frameworks to assist such *threat hunting* efforts. Gao et al. [22], [23] present query languages for threat hunters, and a system for processing their queries. Shu et al. [73] 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 [56] 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.

X. Conclusions

In this paper, we presented a new approach for fast and accurate reconstruction of APT campaigns. It relied on two new techniques, tag attenuation and tag decay, to mitigate the dependence explosion problem. Our experimental evaluation demonstrates 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. Starting from logs containing many millions of events, these graphs pick out just a few dozen events representing an attacker's activities.

References

[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 data release. https://github.com/darpa-i2o/Transparent-Computing/. Accessed: 2019-1-14.

[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] 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/.

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

[11] Source: Deloitte Breach Affected All Company Email, Admin Accounts. https://krebsonsecurity.com/2017/09/source-deloitte-breach-affected-all-company-email-admin-accounts/.

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

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.

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

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

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

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

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

[38] 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.

[39] 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.

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

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

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

[43] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *Operating Systems Design and Implementation*, 2014.

[44] 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.

[45] 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.

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

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

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

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

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

[51] 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.

[52] 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.

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

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

[55] 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.

[56] 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.

[57] 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.

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

[59] 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].

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

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

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

[63] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, 2005.

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

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

[66] 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.

[67] 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.

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

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

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

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

[72] 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.

[73] 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.

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

[75] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[76] 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.

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

[78] Xiaoyan Sun, Jun Dai, Peng Liu, Anoop Singhal, and John Yen. Using bayesian networks for probabilistic identification of zero-day attack paths. *IEEE Transactions on Information Forensics and Security*, 2018.

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

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

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

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

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

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

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

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

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