Taming the Costs of Trustworthy Provenance through Policy Reduction

ADAM BATES, University of Illinois at Urbana-Champaign
DAVE (JING) TIAN and GRANT HERNANDEZ, University of Florida
THOMAS MOYER, MIT Lincoln Laboratory
KEVIN R. B. BUTLER, University of Florida
TRENT JAEGER, Pennsylvania State University

Provenance is an increasingly important tool for understanding and even actively preventing system intrusion, but the excessive storage burden imposed by automatic provenance collection threatens to undermine its value in practice. This situation is made worse by the fact that the majority of this metadata is unlikely to be of interest to an administrator, instead describing system noise or other background activities that are not germane to the forensic investigation. To date, storing data provenance in perpetuity was a necessary concession in even the most advanced provenance tracking systems in order to ensure the completeness of the provenance record for future analyses. In this work, we overcome this obstacle by proposing a *policy-based approach to provenance filtering*, leveraging the confinement properties provided by Mandatory Access Control (MAC) systems in order to identify and isolate subdomains of system activity for which to collect provenance. We introduce the notion of *minimal completeness* for provenance graphs, and design and implement a system that provides this property by exclusively collecting provenance for the trusted computing base of a target application. In evaluation, we discover that, while the efficacy of our approach is domain dependent, storage costs can be reduced by as much as 89% in critical scenarios such as provenance tracking in cloud computing data centers. To the best of our knowledge, this is the first policy-based provenance monitor to appear in the literature.

CCS Concepts: • **Security and privacy → Operating systems security**; **Information flow control**;

Additional Key Words and Phrases: Provenance, mandatory policy, integrity, TCB

This material is based upon work supported by the National Science Foundation under Grants No. CNS-1408880, No. CNS-1540216, No. CNS-1540217, and No. CNS-1657534. Authors from Penn State acknowledge support from the Air Force Office of Scientific Research (AFOSR) under Grant No. AFOSR-FA9550-12-1-0166. The Lincoln Laboratory portion of this work was sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Authors' addresses: A. Bates, Department of Computer Science, Siebel Center, 201 N. Goodwin, Urbana, IL, 61801-2302; email: batesa@illinois.edu; D. (Jing) Tian, G. Hernandez, and K. R. B. Butler, E301 CSE Building, PO Box 116120, Gainesville, FL 32611 USA; emails: {daveti, grant.hernandez, butler}@ufl.edu; T. Moyer, Lincoln Laboratory, Massachusetts Institute of Technology, Secure Resilient Systems and Technology Group, 244 Wood Street, Lexington, MA 02420-9108; email: tmoyer@ll.mit.edu; T. Jaeger, The Pennsylvania State University, Department of Computer Science and Engineering, 346A IST Building, University Park, PA 16802; email: tjaeger@cse.psu.edu.

https://doi.org/10.1145/3062180

ACM Reference format:
Adam Bates, Dave (Jing) Tian, Grant Hernandez, Thomas Moyer, Kevin R. B. Butler, and Trent Jaeger. 2017. Taming the Costs of Trustworthy Provenance through Policy Reduction. *ACM Trans. Internet Technol.* 17, 4, Article 34 (September 2017), 21 pages.
https://doi.org/10.1145/3062180

1 INTRODUCTION

Data provenance, the history of data as it is processed on a system, has proven invaluable in protecting data integrity, conducting forensic analysis, and ensuring regulatory compliance. Automatic provenance-aware systems collect and maintain metadata about every data processing event on the system, allowing developers to invest less time into making applications provenance-aware while still leveraging the benefits of data provenance. However, one major drawback of automatic provenance-aware systems is the sheer volume of data that is collected. Unchecked, such systems have been known to generate gigabytes of provenance in just minutes under heavy system load (Bates et al. 2015; Gehani and Tariq 2012; Muniswamy-Reddy et al. 2006). This not only imposes an excessive storage burden, but also impedes the performance of subsequent analyses of the provenance.

Systems like Linux Provenance Modules (LPM) (Bates et al. 2015) and ProTracer (Ma et al. 2016) collect provenance on every interaction in the system, even if such information is later determined to be of no value to the administrator. Ideally, it would be possible to record only the provenance of system events that are interesting and important. Unfortunately, without additional context it is difficult, if not impossible, to determine ahead of time whether or not the provenance of a particular system event is important. For example, the provenance of a system library may have little value until such time as it is rewritten by an adversary, at which point the history of the library becomes key to understanding the attack. As a result, past provenance-aware systems have needed to record *every* system event. To address this concern, existing efforts have looked for ways to efficiently compress provenance information (Xie et al. 2013), prune the information after collection (Lee et al. 2013b), or condense provenance records through taint propagation (Ma et al. 2016). While these approaches show promise, they rely on post-facto analysis to reduce the amount of data that is ultimately stored, and do not provide a general solution for the removal of unimportant events. Additional work aims to address a related problem of dependency explosion in provenance logs (Lee et al. 2013a), but requires dynamic analysis of the target application before use, and depends on the integrity of the application in order to function correctly. Instead, what is needed is a mechanism that allows the system to filter provenance at the time of collection while maintaining the desired properties of a secure provenance system.

In order to safely filter provenance, the system requires some context about the current and future relationships between objects. One source of such context is the security policy that the system is enforcing, such as the Mandatory Access Control (MAC) policy, which explicitly defines permissible actions on the system. By analyzing this policy as an information flow graph, and identifying the provenance-sensitive objects for an application within the graph, it is possible to exclude events from the provenance log that cannot impact a particular application. In this way, we can collect provenance only for those objects that reside in an application-specific Trusted Computing Base (TCB). *If the enforced security policy prohibits the flow of data between an object outside of an application's TCB to one within, then the object outside of the TCB cannot impact the application, and can therefore be excluded from the recorded provenance.* In a recent position paper (Bates et al. 2015), Bates et al. suggested that MAC-aware provenance collection was a possible means of reducing storage burden; however, they did not demonstrate methods of achieving this goal, define or prove the desired system properties, or design and evaluate an actual system.

In this work, we introduce ProvWalls, a provenance monitor that analyzes a system's MAC policy to identify the security labels that can flow into a target application. These types represent an application-specific TCB, which is used by ProvWalls as a *provenance policy* to generate finely scoped data provenance. We first consider means of mining MAC policies to identify application-specific TCBs, then demonstrate that tracking the system objects within this TCB generates provenance graphs that are both complete and minimal with respect to a target application. We then consider a number of forensic scenarios where provenance-aware systems may be deployed. In evaluation, we discover that our system imposes just 1.5% runtime overhead under realistic workloads, and show that depending upon the deployment scenario, storage overheads can be reduced by at least 36% and up to 89%. We also consider how further optimization is possible through deploying existing reduction techniques in tandem with ProvWalls. Our contributions are summarized as follows:

— **Provenance Completeness:** We present a formal definition for completeness in provenance graphs, and introduce relaxed definitions for selective and minimal completeness that facilitate a tradeoff between the size and expressivity of provenance.
— **Policy-Reduced Provenance:** We design and implement ProvWalls, a policy-based provenance monitor, and demonstrate that security mechanisms in the operating system can be leveraged by the provenance layer to reduce log growth. Our work extends past techniques in MAC analysis to identify a partition of security labels that satisfies the minimal completeness property for a target application.
— **Evaluation and Case Studies:** We perform an extensive performance evaluation of ProvWalls to determine that it imposes as little as 1.5% overhead under realistic conditions. We go on to consider several deployment scenarios, and find that our system reduces provenance storage costs by at least 36% and as much as 89%.

2 BACKGROUND

2.1 Data Provenance

Data provenance is the practice of recording the history of data as it is processed and accessed on a computer system. Provenance has been shown to be of value to fundamental security considerations such as access control (Bates et al. 2013; Nguyen et al. 2012; Ni et al. 2009; Park et al. 2012), conducting forensic analysis (Bates et al. 2014; Gehani et al. 2010; Tariq et al. 2011; Zhou et al. 2010, 2011), enforcing regulatory compliance (Aldeco-Pérez and Moreau 2008; Bates et al. 2013), and establishing the confidentiality and integrity of data (Acar et al. 2012; Cheney 2011).

A variety of approaches have been taken to provenance collection. One option is for applications to *disclose* provenance by inserting the appropriate information into a provenance log at key points within their execution, often using libraries and Application Programming Interfaces (APIs) (Hasan et al. 2009; Macko and Seltzer 2012; Moreau et al. 2011; Muniswamy-Reddy et al. 2009) or automated instrumentation (Lee et al. 2013b) to simplify development. Disclosed provenance systems assume application integrity, without which no strong guarantees of accuracy or completeness can be assured, and are thus best suited for benign environments. Our work focuses on the alternative, *automatic* provenance collection, in which provenance is collected by the operating system without depending on the cooperation of applications. These systems provide stronger guarantees about the completeness and integrity of the provenance, but at a price—the volume of provenance data often dwarfing the volume of regular data on the system.

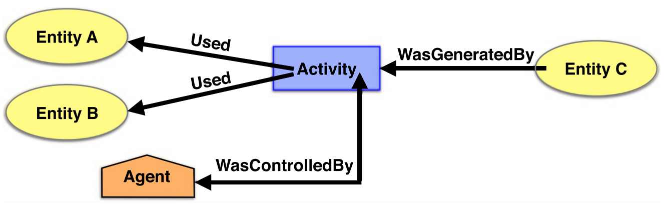


Fig. 1. An example provenance graph represented in the W3C PROV specification. Data objects are referred to as *Entities*, Processes as *Activities*, and Users as *Agents*. PROV defines edges that encode relationships between vertices. This particular graph depicts the history of the **Output** object, Entity C.

After collection, provenance metadata can be processed into a directed acyclic graph representing the history of system execution. This graph representation can be constructed using open standards such as the W3C PROV specification (Consortium et al. 2013), which permits the interchange of provenance information in heterogeneous environments. Figure 1 shows an example provenance graph in the W3C PROV model comprised of dependencies between entities, activities, and agents; these concepts can be mapped into the operating system primitives that we consider in this work. *Entities* are persistent and ephemeral data objects including files, inodes, sockets, and shared memory. *Activities* are the actions performed on data in the system, which in the case of operating systems includes user space processes as well as actions not in user space taken by the kernel. *Agents* control activities, and represent users and groups in system parlance. The PROV data model also specifies a list of relationships between vertices that map to system activities. For example, reading a file corresponds to the *Used* relationship, while writing a file is signified by the *wasGeneratedBy* relationship. As these relationships all point backwards in time, querying the history of an object in a provenance graph involves traversing its successors. A full description of the W3C PROV specification can be found at Consortium et al. (2013).

2.2 Mandatory Access Control

Access control systems are broadly broken into two categories. The first are discretionary access control systems, where users have some degree of autonomy to grant other users access to data, e.g., the standard Unix style where the owner can alter the permissions. The second are mandatory access control systems, where the security policy is centrally managed, and users have no ability to alter, or delegate, access. While MAC has been used as a tool to protect the trusted computing base of provenance-aware systems, the possibility that it can be used to reduce the amount of provenance collected has not been explored.

Our implementation makes use of SELinux as an exemplar MAC system (Smalley et al. 2002). SELinux was originally proposed by the U.S. National Security Agency (NSA) as a MAC framework for the Linux Kernel, where policies are written in terms of types and the operations are permitted between types. SELinux assigns labels (types) to every subject and object within the system, such as the files, sockets, processes, and users. These labels are then used to determine access permissions when the subjects interact with objects. For example, the Apache Web Server has a label `httpd_t`, and web content is labeled `httpd_sys_content_t`. Rules within the SELinux policy then allow subjects with the `httpd_t` domain to read content with the label `httpd_sys_content_t`. One advantage of the SELinux model is that, by generalizing access permissions to labels, it becomes easier to add new objects to the system without extensive rewriting of policy. For example, a new file created in the Apache web directory will automatically be labeled with `httpd_sys_content_t`, and even adding another web server to the system can be achieved by simply labeling the new server process with the `httpd_t` label. In this way, the SELinux policy is not tied to specific applications and content, but to broader types, resulting in policies that are somewhat agnostic to the particular implementation.

Although MAC systems can be used to achieve a variety of security properties, such as enforcing least privilege and protecting runtime platform integrity, it is important to note that MAC is not a "silver bullet" to software security. In particular, MAC cannot prevent the exploitation of vulnerabilities in software. Once compromised, an attacker inherits all privileges on the system that have been granted to the application by the MAC policy. In the case of web services, this likely implies access to the most valuable information on the system, and yet the complexity of such applications often makes it difficult to quickly diagnose and recover from attack. It is here that data provenance is able to provide transparency to the administrator, allowing them to respond more quickly to intrusions. This work is motivated by the need for cost-efficient provenance context in MAC-enabled systems.

3 DESIGN

3.1 Threat Model and Assumptions

We consider a MAC-enabled system that is the target of a remote attacker that has gained access to the host by exploiting a software vulnerability in a network-facing application. Once inside the system, the attacker may take any action permitted by the MAC policy. In an Advanced Persistent Threat (APT) scenario, the adversary will also likely take a "low and slow" approach, meaning they will attempt to avoid taking actions that could be observed by monitoring systems such as our provenance capture agent. The adversary may also attempt to tamper with or disable provenance mechanisms. In spite of this, our provenance capture agent must be able to maintain a fully accurate record of attacker activities from intrusion onwards.

We make the following assumptions with regard to the system. First, we assume that the kernel's security subsystem is correctly implemented to enforce the information flows specified in the MAC policy. We assume that the MAC policy protects the runtime integrity of the kernel; however, the policy cannot completely prohibit undesirable actions on the system, such as a compromised application rewriting its own files to gain persistence on the host. We also assume that the user space utilities introduced in our system are fully protected by the MAC policy, which is reasonable because they only need to interact with a limited number of system objects. The trusted computing base of our system is therefore the kernel, our own helper utilities, the provenance storage, and the base MAC policy responsible for ensuring the integrity of the kernel.

3.2 Design Goals

A summary of the notation used throughout this paper is shown in Table 1. Colloquially, the goal of our architecture is to allow an administrator to collect provenance for all of the system activities they care about, and none that they do not. This provenance must be accurate and complete in its description of system events, in spite of it being collected in a malicious environment that may feature active adversaries on the host. However, while the provenance log may prove valuable for diagnosing system intrusions, it is also of value to the administrator for other purposes, such as identifying benign application misconfigurations.

To arrive at a more specific description of this property, let E be a set of all event tuples $\langle a \in A, s \in S, o \in O \rangle$ that occur during system execution, with a representing an operation being performed by subject s on object o. Let the set $C = S \cup O$ represent all system artifacts. Let $P(G, x)$ be a function that, given a provenance graph G and a system artifact $x \in C$, returns all tuples in G pertaining to the history of x. Depending on the contents of G, this may be either a complete history of x, an incomplete but non-empty history of x, or no history of x. We define a **complete** provenance

Table 1. A Summary of Notation Used in this Article

Notation	Description
A, S, O	The sets of all access types, subjects, and objects on the system.
I	The set of subjects and objects *of interest* to a system administrator.
$\langle a, s, o\rangle$	An access type, subject, and object that comprises a provenance event tuple.
E	The set of all authorized event tuples that occur during system execution.
G	A graph representation of the event tuples recorded during system execution.
$P(G, x)$	A function that returns the event tuples in G pertaining to the provenance of x.
$\top_x$	Represents a subset of E that contains a full history of x.
$\textsc{MayWrite}(x, y)$	A function that returns true if subject x has permission to write to y.
$\textsc{App}(x)$	A function that returns the set of helper applications for subject x.
$\textsc{SecurityLabel}(x)$	A function that returns the MAC label for artifact x.
L	The set of security labels used in the MAC policy.
T_K	The set of MAC subjects that are trusted by the kernel ($T_K \subseteq L$).
$T_{W(s)}$	The set of MAC subjects that have write permission on subject s's executable ($T_{W(s)} \subseteq L$).
$T_{H(s)}$	The set of MAC subjects that are trusted by the helper applications of subject s ($T_{H(s)} \subseteq L$).
TCB_s	The trusted computing base of a target subject s ($TCB_s \subseteq L$).

graph G_C as

$$\forall x \in (S \cup O), P(G_C, x) = \top_x. \tag{1}$$

Unfortunately, in such systems the consequence of capturing complete provenance is inordinately large storage overheads, on the order of several GB per day. We observe that one source of unnecessary storage overhead stems from the fact many of the events in E will not be of any interest to the administrator. For example, the administrator may only be interested in events that inform the execution of a single subject, such as a web server. Let $I \subseteq C$ where I represents those objects and activities of interest to the administrator, and $\bar{I}$ is the complement of I such that $I \cup \bar{I} = C$ and $I \cap \bar{I} = \emptyset$. A **selectively complete** provenance graph G_I possesses the following properties:

$$\forall x \in I, P(G_I, x) = \top_x, \tag{2}$$

$$\forall x \in \bar{I}, P(G_I, x) \subset \top_x. \tag{3}$$

That is, provenance histories are complete for all objects in I, but may be incomplete for objects in $\bar{I}$. However, a graph G_I may still contain events that are not of interest to the administrator. For example, removing a single event tuple from the graph G_C could be selectively complete. Therefore, we introduce a final graph G_M that satisfies the **minimally complete** property:

$$\forall x \in I, P(G_M, x) = \top_x, \tag{4}$$

$$\forall x \in \bar{I}, P(G_M, x) = \top_x \cap \bigcup_{x' \in I} P(G_M, x'). \tag{5}$$

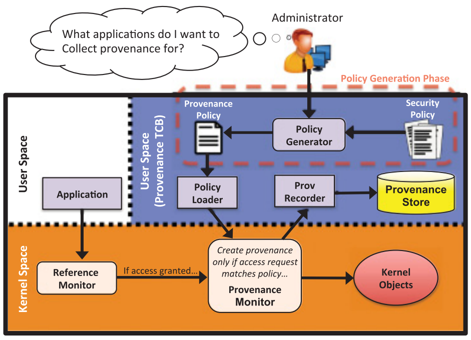


Fig. 2. Overview for the ProvWalls architecture.

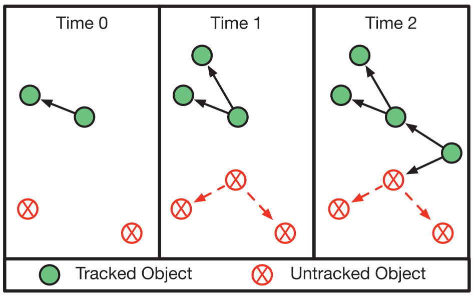


Fig. 3. A time-lapsed system graph demonstrating the difficulty of selective provenance collection.

That is, G_M does not contain any extraneous provenance except for that which is necessary to describe the set I. The goal of our system is to produce minimally complete provenance records given an administrator-specified set I, which constitutes a *provenance policy*.

3.3 Design Overview

An overview for ProvWalls architecture is shown in Figure 2. During a training phase, the *Policy Generator* takes as input the system's active *Security Policy* as well as high-level *Administrator Preferences* (e.g., an application to track) and outputs a *Provenance Policy*. At runtime, this policy is transmitted to the kernel by the *Policy Loader*. As system events occur and are authorized by the system's *Reference Monitor*, they pass through a *Provenance Monitor*. The Provenance Monitor examines the *security contexts* of each object involved in the access request. If any of these contexts are included in the provenance policy, a new record is created and relayed to the *Provenance Recorder* for storage. If the contexts do not match policy, the Provenance Monitor does not generate any new provenance. Technical descriptions of our provenance monitor, which extends LPM (Bates et al. 2015), are included in Section 5.

3.4 Provenance Collection

ProvWalls collects provenance by defining a set of ProvRecord functions that are placed around the kernel such that one check is called before each operation on *controlled data types*, which include files, inodes, superblocks, tasks, modules, sockets, skbuffs, message queues, and shared memory (Zhang et al. 2002). To ensure that the provenance record is an accurate reflection of *authorized* operations, the record is placed directly after system security checks. This is a common approach to provenance collection introduced by Hi-Fi (Pohly et al. 2012). ProvWalls differs in that each ProvRecord function performs a policy check before determining whether or not to create provenance, as shown in Algorithm 1. Given the event tuple $\langle a, s, o \rangle$, where a represents an operation requested by s to be performed on o, ProvRecord examines the contexts of s and o to see if they appear in a pre-determined policy. If *either* appears in the policy, a new provenance record is created for the event tuple. In order to ensure that the provenance record is complete, ProvRecord functions also have the ability to deny access requests, which occurs in the event that the function is unable to generate provenance (e.g., failed to allocate memory).

ALGORITHM 1: Policy check routine for a generic record function in the kernel.

```
Require: $a$ is an access type, $s$ is a subject, $o$ is an object
1: procedure ProvRecord($a, s, o$)
2:     if PolicyMatch($s$) or PolicyMatch($o$) then
3:         Result $\leftarrow$ GenerateProvenance($\langle a, s, o \rangle$)
4:         if Result $\neq$ SUCCESS then
5:             return Deny($\langle a, s, o \rangle$)
6:         end if
7:     end if
8:     return Authorize($\langle a, s, o \rangle$)
9: end procedure
```

3.5 Policy-Reduced Provenance

Several challenges arise when attempting to perform policy-based provenance collection. First, and most fundamentally, we require a flexible language in which to express our collection policy. A naïve approach would be to use standard Unix descriptors such as filenames to identify the objects for which we want provenance. This approach would lead to inordinately long policies, and would have difficulty tracking derivations of the objects named in the policy. Instead, our language must be broad enough to concisely describe broad classes of system objects. Second, any policy-based filtering of provenance will lead to a loss of generality; because filtered provenance is not a general-purpose tool for explaining system events, we must have a clear sense ahead of time of exactly what we would like to collect provenance for.

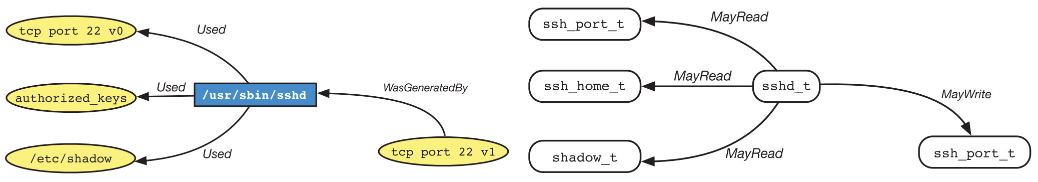
Our approach must also provide assurance that the events omitted from the provenance record cannot affect objects within the provenance policy. This must be true not only at the time the event occurs; it must also be true in perpetuity. Figure 3 demonstrates the consequences of failing to provide this property. At Time 0, the provenance capture agent determines that objects marked by red X's will not affect the objects specified in the provenance policy (tracked objects). Later, at time 2, a tracked object comes to depend on an untracked object, resulting in an incomplete history.

We propose that MAC provides an environment in which these challenges can be overcome. In a MAC-enabled system, every system object is assigned a security label, with similar objects sharing a label, and a policy dictates the permissible interactions between different labels. Because every artifact in a provenance graph will map to a label in the MAC policy, security labels present a general and flexible language for our provenance policy. Finally, MAC also provides a solution to the challenge of ensuring completeness. It has been shown that MAC policies can be analyzed in order to understand the relationships between system objects (Vijayakumar et al. 2012). If we can identify the set of MAC labels that a particular application depends on through similar analysis, then record provenance for all of the labels in that set, we can be confident that that provenance of the target application will be complete in perpetuity. Our PolicyMatch function is defined as shown in Algorithm 2.

The intuition behind this approach is shown in Figure 4. MAC policies can be viewed as information flow graphs that encode the *permissible interactions* between objects in future system executions. Provenance graphs encode the history of *actual interaction* between objects during past system executions. Due to this similar representation, information flow graphs can be overlayed onto provenance graphs, and traversal between the information flow and provenance layers is achieved by defining relationships between system objects and their associated security labels as specified in the MAC policy. Because the MAC policy authorized all events that appear in the provenance graph as they occurred, we can expect to see that all relationships in the provenance graph

ALGORITHM 2: Policy check routine for a generic record function in the kernel.

```
Require: Policy is the provenance policy (a set of security labels)
Require: $x$ is a system artifact
1: procedure PolicyMatch($x$)
2:     $l \leftarrow$ SecurityLabel(x)
3:     if $l \in Policy$ then
4:         return true
5:     else
6:         return false
7:     end if
8: end procedure
```



(a) Provenance Graph (b) Corresponding Information Flow Policy

Fig. 4. Provenance objects and relations can be mapped to mandatory access control labels and transitions, making it possible to information flow overlay to the provenance graph. In the information flow plane, edges encode permissible future actions (e.g., *MayWrite*), whereas in the provenance plane edges encode historic events (e.g., *WasGeneratedBy*).

will correspond to an authorization in the information flow graph. For example, the relationship `sshd_t` *may_write* `ssh_port_t` in the information flow plane corresponds to a *wasGeneratedBy* event in the provenance plane that marks the transmission of a network packet.

3.5.1 Provenance Walls. In this section, we present an algorithm for generating a policy that produces a minimally complete provenance graph for a subject application. We do so by adapting Vijayakumar et. al.'s *Integrity Walls* algorithm (Vijayakumar et al. 2012), which was originally deployed to reason about application attack surfaces by first identifying an application's TCB. For subject s, TCB_s describes the complete set of subjects and objects that s depends on; in other words, TCB_s marks the complete set of subjects and objects that could appear in $P(G_C, s)$. Our goal is therefore to partition the set of MAC policy labels L based on whether or not s depends on the label, ultimately creating the set $TCB_s \subseteq L$.

To begin, we identify the TCB of the system itself. We first manually identify K, the set of types that are critical to the operating system, which include the label for the boot partition, system libraries, and MAC configuration files (e.g., `boot_t`, `lib_t`, and `selinux_config_t` in SELinux). Write access to these types could be used to compromise the kernel; as the system informs the execution of s, we must track the provenance of all subjects that can write to these types. We iteratively calculate the set of subjects that are permitted to write to these types, $T_K \subseteq L$, as follows:

$$T_K^0 = K, \tag{6}$$

$$T_K^i = T_K^{i-1} \cup \{s_2 \mid \exists s_1 \in T^{i-1}, \textsc{MayWrite}(s_2, s_1)\}, \tag{7}$$

$$T_K = \bigcup_{i \in N} T_K^i. \tag{8}$$

We also need to identify the executable writers of the subject s, $T_{W(s)} \subseteq L$:

$$T^0_{W(s)} = s, \tag{9}$$

$$T^i_{W(s)} = T^{i-1}_{W(s)} \cup \left\{ s_2 \mid \exists s_1 \in T^{i-1}_{W(s)}, \textsc{MayWrite}(s_2, s_1) \right\}, \tag{10}$$

$$T_{W(s)} = \bigcup_{i \in N} T^i_{W(s)}. \tag{11}$$

Additionally, many applications consist of multiple distinct processes that run with different permissions. Helper subjects denote distinct processes upon which the subject application depends; for example, Apache depends on the htpasswd process. We must therefore compute the set of subjects that can write to helper subjects for a given application $T_H \subseteq L$.

$$T_{H(s)} = \{s_1 \mid (s_1 \in (\mathsf{App}(s) - \{s\})) \wedge (T_{W(s_1)} \supseteq (\mathsf{App}(s) \cup T_{W(s)}))\}. \tag{12}$$

Given T_K, $T_{W(s)}$, and $T_{H(s)}$, we can then compute s's trusted subjects $T_{S(s)} \subseteq L$ as

$$T_{S(s)} = T_K \cup T_{W(s)} \cup T_{H(s)}. \tag{13}$$

Finally, to complete TCB_s we take the union of $T_{S(s)}$ and the set of trusted objects of s. Trusted objects include configuration files and other data objects that exist in the confined space of the target application.

$$TCB_s = T_{S(s)} \cup \{o \mid \nexists s_1 \in (L - T_{S(s)}), \textsc{MayWrite}(s_1, o)\}. \tag{14}$$

The label set TCB_s provides a complete description of the system objects that are permitted to flow into the subject application s. TCB_s therefore constitutes a provenance policy that satisfies completeness for s. In fact, because TCB_s constitutes the minimum trusted computing base for s, it should provide minimal completeness when applied as a provenance policy.

4 ANALYSIS

In this section, we show that ProvWalls satisfies the desired graph properties. To do so, we define the function $P(G, x)$ from Section 3.2, shown in Algorithm 3. Provenance is a work-list algorithm that, given a provenance graph G and a system artifact x iteratively builds the set of event tuples related to the history of x. A helper function, GetParents, moves one step back in an artifact's history by traversing the appropriate edges depending on whether the artifact is an object (*wasGeneratedBy, wasDerivedFrom*) or subject (*Used, wasControlledBy*).

To demonstrate these properties, we will compare the output of $P(G, s)$ on two graphs, G_C and G'. Assume that G_C satisfies the **completeness** property, i.e., G_C contains all event tuples E that occurred during system execution. G_C can be captured using provenance-aware systems such as LPM (Bates et al. 2015), or Hi-Fi (Pohly et al. 2012); while neither provide a formal proof of completeness, capturing complete whole-system provenance is a stated goal of both systems. Demonstrating completeness is orthogonal to the goal of this work; rather, our aim is to demonstrate that, given the ability to capture G_C, it is also possible to produce a graph G' that exhibits application-specific completeness properties at a reduced cost. G' is generated over the same set of events E as G_C, but by the ProvWalls monitor configured such that *Policy* for the PolicyMatch function is set to TCB_s for a particular target application s.

Selective Completeness. G' is selectively complete for s if $P(G_C, s) \equiv P(G', s)$. To demonstrate, let us assume that G' is not selectively complete for s. This means that there exists some event tuple in I that is encoded in G_C but not in G'. Because ProvWalls records event tuples on the basis of

ALGORITHM 3: Report x's provenance history in G (Appears as $P(G, x)$ in Section 3.2. For readability, some PROV relationships have been omitted from Lines 17 and 19.

```
Require: $x$ is a system artifact $\in (S \cup O)$.
 1: procedure Provenance($G$, $x$)
 2:     $E_x \leftarrow$ GetParents($G$, $x$)
 3:     for each $\langle a, s, o \rangle \in E_x$ do
 4:         $E_y \leftarrow$ GetParents($G$, $s$) $\cup$ GetParents($G$, $o$)
 5:         for each $e \in E_y$ do
 6:             if $e \notin E_x$ then
 7:                 Append($E_x$,$e$)
 8:             end if
 9:         end for
10:     end for
11:     return $E_x$
12: end procedure
13:
14: procedure GetParents($G$, $x$)                          ▷ Traverses 1 step back in $x$'s history.
15:     for each $\langle a, s, o \rangle \in G$ do
16:         $L \leftarrow \emptyset$
17:         if ($o = x$ and $a \in \{wasGeneratedBy, wasDerivedFrom\}$) then
18:             Append($L$,$\langle a, s, o \rangle$)
19:         else if ($s = x$ and $a \in \{Used, wasControlledBy\}$) then
20:             Append($L$,$\langle a, s, o \rangle$)
21:         end if
22:     end for
23: end procedure
```

whether the tuple's subject or object is in TCB_s, for clarity we will focus on the equivalent problem of a particular system artifact x that is absent from G'.

$$\exists\, x \in I \mid x \in P(G_C, s),\ x \notin P(G', s).$$

That is, there is a particular system artifact x that appears in $P(G_C, s)$ but not $P(G', s)$. Let l_x be SecurityLabel(x). By definition $l_x \notin TCB_s$, which means that $l_x \in (L - TCB_s)$.

> **Case 1: IsSubject(l_x):** If l_x is a subject, then $\exists\, k \in TCB_s \mid MayWrite(l_x, k) = true$. However, if $k \in T_K$, then $l_x \in T_k$ because T_K is a transitive closure of the writers of K. The same holds true if $k \in T_W$ or if $k \in T_H$. It follows that $l_x \in T_s$, which contradicts $l_x \notin TCB_s$.
> **Case 2: IsObject(l_x):** Trusted objects are calculated in Equation (14), which states that an object is trusted if there does not exist an untrusted subject that may write to it. Therefore, if $l_x \notin TCB_s$, there exists an untrusted subject $\exists u_x \in (L - TCB_s), MayWrite(u_x, l_x)$. However, for the same reason as Case 1, if s depends on l_x then u_x must be a member of the transitive closure of T_K, T_W, T_H. This contradicts $l_x \notin TCB_s$.

Because $l_x \in TCB_s$ contradicts $\exists\, x \in I \mid x \in P(G_C, s),\ x \notin P(G', s)$, we conclude that $P(G_C, s) \equiv P(G', s)$. Because ProvWalls only records provenance for events which contain a subject or object in TCB_s, it trivially follows that $\forall x \in \bar{I}, P(G_I, x) \subset \top_x$.

Minimal Completeness. G' is minimally complete if all system artifacts in G' have an associated security label in TCB_s. To demonstrate, let us assume that G' is not minimally complete for

Table 2. Summary of Hooks Implemented by ProvWalls Kernel Module

Hooks (Count)	Purpose
Credential (5)	Track Forks and Active UIDs
Inode (10)	Track Inode Access, Creation, Linkage, etc.
Superblock (3)	Track superblock mount, dismount
File (4)	Track file access, creation, mapping
IPC (7)	Track IPC over shared mem, msg queues, etc.
Network (16)	Track INET and UNIX communications.

s, which would mean

$$\exists\, x \in I \,|\, x \in G',\ l_x \notin TCB_s.$$

That is, there is a particular system artifact x in $P(G', s)$ for which $\textsc{SecurityLabel}(x)$ is not in TCB_s. x appears in $P(G', s)$ if and only if $\textsc{PolicyMatch}(x)$ returns true, implying that $l_x \in Policy$. Because $Policy = TCB_s$ this contracts $l_x \notin TCB_s$. Note that minimal completeness does not imply that all x in G' must appear in $P(G', s)$; this is because an artifact x may not currently appear in s's ancestry, but has the potential to eventually flow into s because $l_x \in TCB_s$.

5 IMPLEMENTATION

ProvWalls has been implemented for Linux Provenance Modules, which is based on Red Hat Linux Kernel 2.6.32.[1] Our primary development machines were running CentOS 6.5.

Policy Generator. The tool for generating provenance policies was written in C++ using the Stanford SNAP graph library.[2] It takes as input a compiled SELinux policy, an executable-to-subject mapping, a list of kernel subjects, and a target application s. It then generates TCB_s as described in Section 3. The tool outputs the list of types required to record complete provenance for the subject application. In our evaluation, we make use of the SELinux Targeted Policy. While we confirmed that the tool also works with the MLS and reference policies, we chose to use the targeted policy because it is the default SELinux policy in Red Hat Enterprise Linux and is subject to less compatibility issues. Each of the applications we evaluated ran in a confined domain.

Kernel Module. The ProvWalls Module implemented 45 hook functions for LPM's provenance hooks, a summary of which is shown in Table 2. The provenance policy is transmitted to kernel space by a script in `rc.local` once user space has loaded. This is accomplished by writing to a directory in `securityfs` that is created by the kernel module during startup.

The provenance collection logic was taken from LPM's Provmon module, which ProvWalls extends by introducing policy logic. For each hook, the module extracts the security contexts for the involved system entities, and passes them through a policy check. For example, in the `inode_permission` hook, a policy check is performed on both the inode and the credentials associated with the fork attempting to access the inode. We determined which entities to check for each hook by referencing the SELinux kernel module implementation. In the policy check, the module extracts the security ID (sid) of the object and then translates the sid to a character string using the `selinux_sid_to_string` function defined in the SELinux module. The type is extracted from the context, and then compared to the policy entries. If there is a policy match, a new provenance record is created and relayed to the provenance recorder. Otherwise, the hook takes no action and returns control to the calling function.

[1]Red Hat has continued to use a modified version of the Linux 2.6.32 kernel through 2015. Red Hat Enterprise Linux 6 and CentOS 6, which are both based on a 2.6.32 kernel, will not reach end of life until 2021.
[2]See http://snap.stanford.edu.

Performing comparisons on character string security contexts, as supposed to sids, is necessary because there is a one-to-many mapping between the type (contained in the context) and sids. However, we implement two optimizations to reduce the burden of performing string operations. First, after a sid has been inspected once, we cache the result. This way, there is only a single string comparison for each sid in the system. Second, once a specific system object's sid has been checked, we store the result by setting a tracking flag within the provenance/security struct of the object. In this way, policy checks amortize to constant overhead for long-lived kernel objects. Performing an additional policy check is only required when an event could cause a kernel object to transition to a different sid, e.g., `task_fix_setuid`.

Provenance Recorder and Analysis Tool. The Provenance Recorder was written in C++, and recorded provenance relayed from kernel space into an in-memory graph using the SNAP library.[2] The Recorder is launched by `initd` in `rc.local` during the boot process after user space is loaded. After being executed, the Recorder reads a copy of the SELinux information flow graph into memory, as well as the active provenance policy. Because some kernel hooks are called from an interrupt context, it was important to reduce the time taken by provenance generation to an absolute minimum. Therefore, to reduce the size of messages, the ProvWalls module transmits sids in the relay instead of character string security labels. In user space, the Recorder translates the sids back into security labels, then extracts the type. The Recorder also serves as an analysis tool that allows us to track the number of event tuples and issue queries to the provenance graph.

6 EVALUATION

In this section, we investigate whether ProvWalls provides comparable performance to a standard Red Hat kernel for CentOS 6.5 and LPM's Provmon module. Our benchmarks were run on a bare metal Dell PowerEdge R610 blade server with 12GB memory and two Intel Xeon quad-core CPUs. We used the Red Hat 2.6.32 kernel, which was compiled and installed with three different configurations: (1) all provenance disabled (*Vanilla*), (2) LPM installed with the Provmon module enabled (*Provmon*), and (3) LPM with ProvWalls enabled.

6.1 Collection Performance

In the following benchmarks, ProvWalls was configured to use a provenance policy that contained 2,000 security contexts. Every file in the benchmark directories, including the software binaries, had SELinux labels that were contained in the policy, meaning that provenance was generated for every operation that was performed in the evaluation. We choose to use a policy of 2,000 labels, much larger than was needed to track the provenance of the tests, in order to ensure that any performance footprint associated performing policy checks in the kernel would be present in the results.

6.1.1 Microbenchmarks. We used LMBench to microbenchmark ProvWalls's impact on system calls, context switching, networking, file, and memory latencies. Table 3 shows the overhead Provmon and ProvWalls incur against the Vanilla kernel for each microbenchmark. Our results show that for most test programs the difference between Provmon and ProvWalls is negligible. However, both introduce a small overhead compared to the Vanilla kernel. There are also many instances in which the instrumented kernels marginally outperform Vanilla; this can be attributed to cache collision anomalies (Inouye et al. 1992; Wright et al. 2002), and is consistent with the observations of Bates et al. (2015). There are some test cases where the additional overhead is noteworthy for both Provmon and ProvWalls: *null I/O*, *stat*, and *open/close file* in process testing, and *file delete* in file and memory latency testing. The overhead in these cases is due to disproportionately higher disk I/O overhead in Provmon, an explanation of which can be found in Bates

Table 3. LMBench Measurements for Provenance Kernels (Average of Five Trials). Percent Overhead for Modified Configurations are Shown in Parentheses

Test Type	Vanilla	ProvMon	ProvWall
Process tests, times in μsec (smaller is better)			
null call	0.14	0.14 (0%)	0.14 (0%)
null I/O	0.21	0.35 (66.7%)	0.37 (76.2%)
stat	2.07	4.41 (113.0%)	4.02 (94.2%)
open/close file	3.00	6.10 (103.3%)	5.45 (81.7%)
select TCP	4.28	3.47 (0%)	3.41 (0%)
signal install	0.25	0.25 (0%)	0.25 (0%)
signal handle	1.41	1.37 (0%)	1.39 (0%)
fork process	407.6	405.8 (0%)	405.6 (0%)
exec process	1,001.4	1,038.2 (3.7%)	1,070.2 (6.9%)
shell process	3,240.6	3,372.2 (4.1%)	3,415.4 (5.4%)
File and memory latencies in μsec (smaller is better)			
file create (0k)	57.2	54.6 (0%)	54.8 (0%)
file delete (0k)	9.4	17.3 (84.0%)	16.1 (71.3%)
file create (10k)	75.0	74.9 (0%)	73.3 (0%)
file delete (10k)	13.1	19.4 (48.1%)	21.5 (64.1%)
mmap latency	1,105.6	1056.4 (0%)	1078.4 (0%)
protect fault	0.328	0.364 (11.0%)	0.318 (0%)
page fault	0.02770	0.02758 (0%)	0.2740 (0%)
100 fd select	1.554	1.534 (0%)	1.538 (0%)
Local Communication latencies in μsec (smaller is better)			
Pipe	12.02	13.28 (10.5%)	13.30 (10.6%)
AF UNIX	12.20	22.08 (81.0%)	19.06 (56.2%)
UDP	30.08	33.06 (9.9%)	33.74 (12.2%)
TCP	43.34	49.08 (13.2%)	47.66 (10.0%)
TCP conn	50.00	50.00 (0%)	56.20 (12.4%)

et al. (2015). More importantly, each of these test cases shows ProvWalls performs comparably to Provmon in spite of the introduction of policy checks.

For our communications tests, we examined *Pipe*, *AF UNIX*, *UDP*, *TCP*, and *TCP conn* (Table 3, Part 3). Similar to Provmon, ProvWalls embeds policy checks for different types of sockets in the system. For UNIX-domain sockets, these sockets involve file I/O, which burdens Provmon/ProvWalls and makes them slower than UDP/TCP sockets. Both Provmon and ProvWalls not only check the local sockets during creation, but also inspect the peer sockets (within the same machine) after connection. Even though Provmon and ProvWalls add latency in the setup phase of network communication, Figure 5 shows the throughput of local communications stays stable in all three kernels. Compared to Vanilla, both Provmon and ProvWalls introduce a negligible overhead to process IPCs, sockets, file sharing, and memory operations.

6.1.2 Macrobenchmarks. To show how ProvWalls impacts a production system, we applied three macrobenchmarks that represent realistic system workloads. The results of these tests are summarized in Table 4. For the Kernel Compile, we fixed the kernel configuration and rebooted the machine before each compilation to prevent any caching effects. We used four threads for each compilation. Unlike the overheads in the microbenchmarks, none of the macro tests show

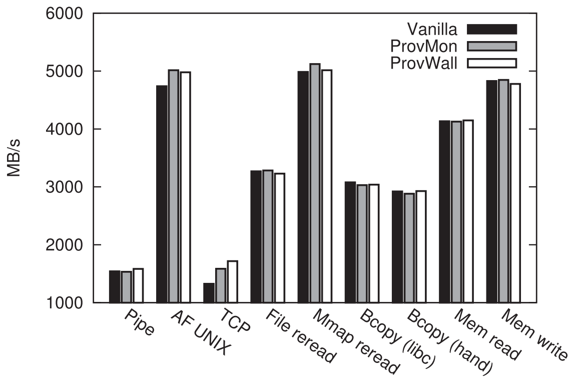


Fig. 5. LMBench throughput measurements for local communication bandwidth using different kernels.

Table 4. Results for Various System Benchmarks Under Three Different Kernel Configurations. Percent Overhead for Modified Configurations are Shown in Parentheses. Average of Five Trials

Test	Vanilla	ProvMon	ProvWall
Kernel Compile	1,028.5 sec	1,030.0 sec (0.1%)	1,040.5 sec (1.2%)
Postmark	13.6 sec	13.8 sec (1.5%)	13.8 sec (1.5%)
Blast	377.8 sec	369.4 sec (0%)	369.8 sec (0%)

overheads above 2%. ProvWalls introduces 1.2% overhead, while Provmon imposes 0.1% overhead on kernel compilation. ProvWalls implements extra policy checking and provenance filtering in kernel space compared to Provmon. Moreover, the overhead imposed by ProvWalls is only 12 seconds, which is tiny compared to the time of complete kernel compilation. The Postmark benchmark simulates the operation and workload of an email server. We configured it to run 15,000 transactions with file sizes ranging from 4KB to 1MB in 10 subdirectories and up to 1,500 simultaneous transactions, based on the official configuration recommendations. Both Provmon and ProvWalls imposed 1.5% overhead on this task, which shows that ProvWalls could be integrated into email servers running Provmon without reducing the performance. We then ran the Blast benchmarks to estimate ProvWalls's overhead on scientific computation. Blast simulates different biological sequence analysis workloads, collected by the National Institutes of Health (NIH). Since this workload is in user-space and CPU-bounded, neither Provmon nor ProvWalls display any overhead compared to Vanilla, demonstrating ProvWalls can be deployed in scientific computation environments.

6.2 Case Studies

As ProvWalls provides application-specific scoping for provenance collection, it follows that its storage reduction performance will also be domain-specific. To characterize this aspect of performance, we now consider several enterprise deployment scenarios in which provenance tracking would be a helpful capability and MAC is likely to be present. For each scenario, we generate a sizable workload for the provenance-aware host by repeatedly performing different kinds of system access. We then ran these workloads on a Virtual Machine (VM) under two different kernel configurations. In the first trial, we execute the workload with LPM's ProvMon module enabled. In the second trial, we defined a provenance policy using the algorithm described in Section 3, then repeated the workload with ProvWalls enabled. The remote side of the workloads were initiated

Table 5. ProvWalls Provenance Reduction Compared to LPM's ProvMon Under Different Workloads. Numbers Represent the Number of Event Tuples Recorded by the Kernel During the Trial. ProvWalls Experienced up to 89% Reduction in Storage Overhead Compared to ProvMon

Test	Workload	ProvMon	ProvWall	Reduction
`lighttpd`	Resource Access	120 MB	77 MB	35.5%
`proftpd`	Command Execution	3.6 MB	1.7 MB	54.3%
`qemu`	Filter Guest VM	79 MB	9 MB	88.5%

by running a script on the host machine. Throughout both trials, the machine was in a quiescent state outside of the workload; to reflect normal background noise, we did not disable daemons and cron jobs that ran by default on CentOS, but we did not interact with these applications in any way during the trial.

Technical details for each of the scenarios follows:

(1) **Resource Access Attack:** A vulnerability in lighttpd 1.4.18 and earlier gives rise to an information disclosure vulnerability that allows an attacker to read arbitrary system files (United States Computer Emergency Readiness Team 2008). In this scenario, we configured ProvWalls to run with a policy that monitored the TCB of the `lighttpd_t` subject. We then launched the attack by modifying an exploit of this vulnerability found in the Exploit Database. To generate a sizable provenance log, we repeated the attack 1,000 times for each kernel configuration.

(2) **Remote Command Execution:** A vulnerability in the `mod_copy` module of ProFTPD 1.3.5 allows remote attackers to read and write to arbitrary system files (United States Computer Emergency Readiness Team 2015). We launched the attack by modifying an exploit of this vulnerability found in the Exploit Database. During the trial, we loaded a policy representing the TCB for the `ftpd_t` subject. The attack was launched 1,000 times for each kernel configuration.

(3) **Filter Guest VM:** This scenario is not motivated by a forensic investigation, but instead considers provenance tracking in data center environments. Administrators need a way of evaluating the configuration and integrity of their machines, but system auditing and traditional provenance collection are ill suited to this task because they also capture the activities of jobs being executed on the machine on behalf of unprivileged users. In cloud environments, this is largely unnecessary; because the virtual hypervisor enforces isolation between guest VMs and the host machine, guest activities cannot inform the execution of the host machine.[3] For this trial, we generate a policy for the `qemu_t` subject, but then removed the `svirt_t` subject, which is assigned to the Guest VM process, from the policy. While this label is a part of the `qemu_t` TCB according to SELinux, we know in practice that this subject should not affect the TCB due to virtualized isolation. The decision to remove this label can be likened to the iterative refining of security policy that is common to any MAC deployment. In the guest machine, we then ran 3,000,000 transactions with the `postmark` tool.

6.2.1 Performance Results. The results for each trial are shown in Table 5. Unsurprisingly, the reduction percentage observed varied significantly. This reflects the fact that our approach to provenance filtering is domain-specific; effectiveness will vary based on the percentage of system activity that is pruned by the label space partition. However, even in the worst cases, ProvWalls

[3]While possible, there are few public disclosures of exploits that allow a guest VM to break isolation and execute code on the host machine.

enjoyed significant reductions in overhead. In the `lighttpd`, and `proftpd` tests, system interaction was dominated by our workload. In spite of this, overhead was reduced by 35.5% and 54.3%, respectively. The reason for this is twofold. First, the interactions of unrelated system processes that are not a part of the target TCB, which represented at least 48% of the label space in each trial, are filtered from the provenance stream. Second, the provenance of outputs from the target application are also filtered, provided that they cannot flow back into the TCB.

In the `qemu` trial, ProvWalls was able to reduce the size of the provenance log by 88.5%. The reason for the improved result in this trial is that we were able to express a policy that filtered the vast majority of system activity (i.e., the `postmark` workload running inside of the VM). We expect ProvWalls to perform comparably in any scenario in which the provenance of the target application represents a minority of overall system activity.

7 DISCUSSION

Availability of MAC policies for use with ProvWalls?

ProvWalls leverages MAC's insight into permissible future actions on the system in order to provide finely scoped provenance collection. It therefore requires the presence of MAC; however, we do not consider this as a limitation to our approach, as robust and fine-grained security policies are already effectively ubiquitous on Linux systems. While our approach could be adapted to any MAC mechanism, we chose to implement ProvWalls using SELinux due to its widespread availability and the presence of stable analysis tools. Although there are known administrative difficulties to configuring SELinux for custom applications, enabling SELinux for popular applications on common Linux distributions is trivial. In fact, the targeted policy we make use of in this work is enabled by default on all Red Hat Linux distributions. The reliance on MAC is therefore not an impediment to the use of our system.

Will MAC-scoped provenance contain valuable information?

In ProvWalls, we provide a mechanism that facilitates a tradeoff between provenance cost and expressivity. As a consequence, provenance collected by our system does not support arbitrary queries. Instead, the administrator must identify ahead of time the scope of their inquiries. However, the scoping mechanism provided by ProvWalls is optimally conservative in identifying potentially valuable system activity. This is because, through MAC policy analysis, ProvWalls identifies every system type that may eventually become important to the function of a particular application. For instance, the web server policy used in our case study captures provenance for the `ftpd`, `firefox`, and `ssh` applications, as each of these applications may potentially interact with the web server. This example serves to demonstrate that ProvWalls does not track individual applications, but entire ecosystems of related applications whose relationships would be difficult to manually enumerate. While MAC is useful in protecting platform integrity following an application compromise, the ProvWalls mechanism is complementary in that it allows an administrator to understand the nature of the attack. For example, provenance collected by ProvWalls could be instrumental in perfoming root cause analysis in web services attacks such as the vulnerabilities explored in Section 6.2.

Can ProvWalls interoperate with other provenance reduction techniques?

Past approaches to provenance reduction have exploited either graph properties or data processing artifacts, neither of which are affected by ProvWalls. Xie et al. leverage web graph compression algorithms to reduce the size of provenance graphs (Xie et al. 2011). They specifically depend on the locality and similarity of objects in the graph; these properties are not only preserved by ProvWalls, but are likely enhanced, as objects that are dissimilar from the target subject and its dependencies are filtered from the graph. Xie extends this technique by leveraging dictionary encoding to further reduce the size of the graph (Xie et al. 2012, 2013). Relative to the size of the graph, we expect ProvWalls to increase the frequency of string occurrences, so dictionary encoding remains a viable approach. Rather than relying on graph properties, Lee et al. leverage observations about common data processing paradigms to reduce the size of provenance. Most notably, they demonstrate that short-lived temporary files can often be pruned from the provenance graph without loss of forensic context (Lee et al. 2013b). These kinds of files are still present in ProvWalls' provenance logs, as they can be read by subjects within the TCB of the target application, so this technique should remain applicable. We intend to investigate the feasibility of combining these techniques in future work.

8 RELATED WORK

ProvWalls is a MAC-aware implementation of a *provenance monitor* (McDaniel et al. 2010), a provenance mechanism that satisfies the reference monitor concept (Anderson 1972). The goal of the provenance monitor is to collect high integrity provenance that cannot be manipulated by an adversary on the system. Pohly et al.'s Hi-Fi is a Linux Security Module (LSM) that collects *whole-system provenance* that details the actions of processes, Inter-Process Communication (IPC) mechanisms, and even the kernel itself (which does not exclusively use system calls) (Pohly et al. 2012). The LPM generalizes this approach by introducing a dedicated provenance layer in the kernel (Bates et al. 2015). LPM has the added benefit of avoiding interference with active security modules, such as SELinux. This interaction between the security and provenance subsystems is what makes our work possible. While these past systems leverage the security layer to ensure provenance integrity, ProvWalls is the first provenance monitor to leverage security guarantees to decrease the cost of provenance capture and management.

Information flow analysis has been employed previously to reason about the security and integrity of systems. The Integrity Walls system performs static analysis on MAC policies in order to identify application attack surfaces by differentiation between an application's trusted inputs and adversary-controlled inputs (Vijayakumar et al. 2012) The Policy-Reduced Integrity Measurement Architecture (PRIMA) reduces the number of entities that need to be known and trusted by remote verifiers of a system (Jaeger et al. 2006). It accomplishes this by extending the Linux Integrity Measurement Architecture (IMA) (Sailer et al. 2004) and SELinux (Runge 2004), and provides remote verifiers with the active MAC policy and a Code-Subject mapping. This allows the verifier to confirm that all subjects permitted to interact with the target are either trusted or filtered. Like remote attestation of system integrity, in practice analyzing provenance logs can be inordinately complex, requiring foreknowledge of all system activities. ProvWalls leverages knowledge of information flows on the system to reduce this cost and complexity.

Recognizing storage overhead as a fundamental challenge to automatic collection (Braun et al. 2006), considerable attention has been paid to techniques that reduce the storage burden of provenance through compression or filtering. Web compression and deduplication have adapted to provenance to reach storage reduction ratios of 3.31:1 (Xie et al. 2011); a "Web+Dictionary" technique further improves the compression ratio up to approximately 5:1 (Xie et al. 2012, 2013). The SPADE system allows provenance to be filtered based on filename blacklists, but it is not possible to reason formally about the completeness properties of the remaining provenance using this approach (Gehani and Tariq 2012). Chen et al. (2013) considers techniques for reducing provenance storage costs through use-inspired filters, analyzing previous-collected provenance graphs to determine which new events require provenance generation. Lee et al. (2013b) proposes a garbage collection technique for audit logs that removes *unreachable objects*, such as temporary files, that neither influence nor are influenced by processes besides their owner. Danger et al. (2015) and Cadenhead et al. (2011) consider access control methods for data provenance that elide information from a graph that a user does not have access to, but these techniques are for offline view generation and not online filtering. In contrast to the above approaches, our work is the first to consider how information flow policy can be leveraged to perform provenance filtering at the time of collection.

9 CONCLUSION

Provenance can offer insight into history of objects being processed on a system, but the post-facto nature of forensic inquiry means that large amounts of provenance metadata must be stored indefinitely in order to ensure a complete explanation. Unfortunately, a means of safely removing extraneous or useless information from the provenance record has eluded even state-of-the-art provenance systems, as doing so requires a means of assuring that filtered provenance will not eventually flow into objects of interest on the system. In this work, we introduce ProvWalls, a mechanism for performing finely scoped provenance filtering with assurances of completeness and minimality for a targeted set of applications. We demonstrate the correctness of our approach, and consider a variety of scenarios in which ProvWalls could dramatically improve the cost-benefit ratio of provenance collection. In evaluating our system, we show that it introduces negligible run-time overheads for realistic workloads, and can reduce storage costs by as much as 89%. ProvWalls is thus a powerful new mechanism for providing low-cost provenance to secure computing deployments.

ACKNOWLEDGMENTS

We would like to thank Mugdha Kumar, Nirupama Talele, and Divya Muthukumaran for their technical assistance on the LPM and Integrity Walls codebases.

REFERENCES

Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2012. *Principles of Security and Trust: First International Conference*. Springer, Berlin, 410–429. DOI: http://dx.doi.org/10.1007/978-3-642-28641-4_22

Rocío Aldeco-Pérez and Luc Moreau. 2008. Provenance-based auditing of private data use. In *Proceedings of the 2008 International Conference on Visions of Computer Science (VoCS'08)*.

James P. Anderson. 1972. *Computer Security Technology Planning Study*. Technical Report ESD-TR-73-51. Air Force Electronic Systems Division.

Adam Bates, Kevin Butler, Andreas Haeberlen, Micah Sherr, and Wenchao Zhou. 2014. Let SDN be your eyes: Secure forensics in data center networks. In *Proceedings of the NDSS Workshop on Security of Emerging Network Technologies (SENT)*.

Adam Bates, Kevin R. B. Butler, and Thomas Moyer. 2015. Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs. In *Proceedings of the 7th International Workshop on Theory and Practice of Provenance (TaPP'15)*.

Adam Bates, Ben Mood, Masoud Valafar, and Kevin Butler. 2013. Towards secure provenance-based access control in cloud environments. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*.

Adam Bates, Dave (Jing) Tian, Kevin R. B. Butler, and Thomas Moyer. 2015. Trustworthy whole-system provenance for the Linux kernel. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*.

Uri Braun, Simson Garfinkel, David A. Holland, Kiran kumar Muniswamy-Reddy, and Margo I. Seltzer. 2006. Issues in automatic provenance collection. In *International Provenance and Annotation Workshop (IPAW)*. Springer, 171–183.

Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. A language for provenance access control. In *Proceedings of the 1st ACM Conference on Data and Application Security and Privacy (CODASPY'11)*.

P. Chen, B. Plale, and T. Evans. 2013. Dependency provenance in agent based modeling. In *Proceedings of the IEEE 9th International Conference on eScience*.

James Cheney. 2011. A formal framework for provenance security. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium*.

World Wide Web Consortium and others. 2013. PROV-overview: An overview of the PROV family of documents. (2013).

Roxana Danger, Vasa Curcin, Paolo Missier, and Jeremy Bryans. 2015. Access control and view generation for provenance graphs. *Future Generation Computer Systems* 49 (2015), 8–27. DOI: http://dx.doi.org/10.1016/j.future.2015.01.014

A. Gehani, B. Baig, S. Mahmood, D. Tariq, and F. Zaffar. 2010. Fine-grained tracking of grid infections. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing (GRID'10)*.

Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference (Middleware'12)*.

Ragib Hasan, Radu Sion, and Marianne Winslett. 2009. The case of the fake Picasso: Preventing history forgery with secure provenance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*.

Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. 1992. The effects of virtually addressed caches on virtual memory design and performance. *SIGOPS Opering Systems Review* 26, 4 (Oct.1992), 14–29. DOI:http://dx.doi.org/10.1145/142854.142859

Trent Jaeger, Reiner Sailer, and Umesh Shankar. 2006. PRIMA: Policy-reduced integrity measurement architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT'06)*.

Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013a. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 20th ISOC Network and Distributed System Security Symposium (NDSS)*.

Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013b. LogGC: Garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*.

Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of the 23rd ISOC Network and Distributed System Security Symposium (NDSS)*.

Peter Macko and Margo Seltzer. 2012. A general-purpose provenance library. In *4th Workshop on the Theory and Practice of Provenance (TaPP'12)*.

P. McDaniel, K. Butler, S. McLaughlin, R. Sion, E. Zadok, and M. Winslett. 2010. Towards a secure and efficient system for end-to-end provenance. In *Proceedings of the 2nd Conference on Theory and Practice of Provenance (TaPP'11)*.

Luc Moreau, Trung Dong Huynh, Mike Jewell, Amir Sezavar Keshavarz, Jamal A. Hussein, and Danius Michaelides. 2011. ProvToolbox. Retrieved from *http://lucmoreau.github.io/ProvToolbox/*.

Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*.

Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. 2009. Layering in provenance systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (ATC'09)*.

Dang Nguyen, Jaehong Park, and Ravi Sandhu. 2012. Dependency path patterns as the foundation of access control in provenance-aware systems. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance (TaPP'12)*.

Qun Ni, Shouhuai Xu, Elisa Bertino, Ravi Sandhu, and Weili Han. 2009. An access control language for a general provenance model. In *Secure Data Management*.

Jaehong Park, Dang Nguyen, and R. Sandhu. 2012. A provenance-based access control model. In *Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST)*.

D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. 2012. Hi-Fi: Collecting high-fidelity whole-system provenance. In *Proceedings of the 2012 Annual Computer Security Applications Conference (ACSAC'12)*.

Chris Runge. 2004. SELinux: A new approach to secure systems. (July2004).

Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. 2004. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*.

Stephen Smalley, Chris Vance, and Wayne Salamon. 2002. *Implementing SELinux as a Linux Security Module*. Technical Report. NAI Labs Report #01-043.

Dawood Tariq, Basim Baig, Ashish Gehani, Salman Mahmood, Rashid Tahir, Azeem Aqil, and Fareed Zaffar. 2011. Identifying the provenance of correlated anomalies. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC'11)*.

United States Computer Emergency Readiness Team. 2008. Vulnerability Summary for CVE-2008-1270. Retrieved from https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1270.

United States Computer Emergency Readiness Team. 2015. Vulnerability Summary for CVE-2015-3306. Retrieved from https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3306.

Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman, and Trent Jaeger. 2012. Integrity walls: Finding attack surfaces from mandatory access control policies. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS'12)*.

Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. 2002. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*.

Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell D. E. Long. 2012. A hybrid approach for efficient provenance storage. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM'12)*.

Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell D. E. Long. 2013. Evaluation of a hybrid approach for efficient provenance storage. *Transactions on Storage* 9, 4 (Nov.2013), Article 14, 29 pages. DOI:http://dx.doi.org/10.1145/2501986

Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Darrell D. E. Long, Ahmed Amer, Dan Feng, and Zhipeng Tan. 2011. Compressing provenance graphs. In *Proceedings of the 3rd Workshop on the Theory and Practice of Provenance (TAPP'11).*

Xiaolan Zhang, Antony Edwards, and Trent Jaeger. 2002. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium.*

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

Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD).*

Received July 2016; revised January 2017; accepted March 2017