Linux Provenance Modules: Secure Provenance Collection for the Linux Kernel

Adam Bates, Kevin R. B. Butler
Department of Computer and Information Science
University of Oregon, Eugene, OR
{amb, butler}@cs.uoregon.edu

Thomas Moyer
Lincoln Laboratory
Massachusetts Institute of Technology, Lexington, MA
thomas.moyer@ll.mit.edu

***Abstract*—In spite of a growing interest in provenance-aware systems, mechanisms for automated provenance collection have failed to win acceptance in mainstream operating systems. This is due in part to a lack of consensus within disparate provenance development communities on a single general solution – provenance collection mechanisms have been proposed at a variety of operational layers wthin host systems, collecting metadata at a variety of scopes and granularities. Since provenance-aware systems must meet the needs of a variety of users in academic, enterprise, and government sectors, any provenance mechanisms must be capable of supporting many different *provenance models* while simultaneously ensuring the security of the provenance they collect. We present the Linux Provenance Modules (LPM), the first general framework for the development of provenance-aware systems that imposes as little as 0.6% performance overhead on system operation. A key feature of LPM is its ability to leverage Linux's existing security features to provide strong provenance security assurances. We go on to introduce a mechanism for *policy-reduced provenance* that reduces the costs of provenance collection by up to 74% by identifying a system's trusted computing base. To our knowledge, this is the first working policy-based provenance monitor proposed in the literature.**

I. Introduction

Provenance is a well-known concept in the art world, but is relatively new to computer science. The idea is that a system can gather and report metadata that describes the history of each object being processed. This allows system users to track, and understand, how a piece of data came to exist in its current state on the system. The application of provenance is presently of enormous interest at different scopes and levels in a variety of disparate communities including scientific data processing, databases, software development, storage [1], [15], operating systems [14], [20], access control [17], [19], and distributed systems [5], [30], [31]. In spite of many proposed models and frameworks, mainstream operating systems still lack support for provenance collection and reporting. This may be due to the fact that the community has yet to reach a consensus on how to best prototype new provenance proposals, leading to redundant efforts, slower development, and a lack of adoptability.

Exacerbating this problem is that, due to a lack of better alternatives, researchers often choose to implement their provenance-aware systems by overloading other system components [15], [20]. Unfortunately, this introduces further security and interoperability problems; in order to enable provenance-aware systems, users currently need to disable their MAC policy [20], instrument applications [12], [30], gamble on experimental storage formats [15], or sacrifice other critical system functionality. These issues point to a pressing need for a dedicated platform for provenance development. We present the design of a **Linux Provenance Monitor framework (LPM)**, the first generalized framework for the development of automated, whole-system provenance collection on the Linux operating system. LPM was designed with consideration for a variety of automated provenance systems that have been proposed in the literature, and in fact includes a re-implementation of the Hi-Fi system [20]. The framework is designed in such a way to allow for experimentation with new provenance collection mechanisms, and permits interoperability with other security mechanisms.

Using LPM as a starting point, we go on to address another major obstacle to the adoption of automated provenance systems – excessive storage overhead [7]. We do so by presenting a novel scheme for *policy-reduced provenance*; we make the surprising discovery that a system's existing security contexts can be utilized in order to record *complete* provenance over a *subset* of system operations. We call this provenance monitor **Provenance Walls**, because it extends Jaeger et. al.'s *Integrity Walls* [25] work by leveraging the confinement properties of a mandatory access control policy to record a complete provenance history inside (or outside) of an application's trusted computing base (TCB). In evaluation, we observe up to a 74% reduction in storage overhead when using Provenance Walls instead of an LPM-based Hi-Fi module, and no more than 2.8% performance overhead on under realistic workloads. As a result, Provenance Walls significantly decreases the performance barriers that currently hinder the widespread deployment provenance-aware systems.

Our contributions can thus be summarized as follows:

- **Linux Provenance Modules (LPM) Framework**, a patch for Red Hat Linux that permits the secure development of provenance-aware Linux systems. LPM supports the needs of many past proposals while being minimally invasive to the rest of the Linux kernel.
- **Policy-reduced provenance**. We show that a system's existing MAC policy can be used to record *complete* provenance over a policy-specified subset of system activities. Using this approach our Provenance

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.

Walls system reduces both the storage requirements and computational overhead of automated provenance collection. To our knowledge, this is the first policy-based provenance monitor that has been proposed in the literature.

- **Performance Analysis**. We perform microbenchmarks on Linux system calls, as well as macrobenchmarks based on compiled the Linux kernel. We show that the LPM patch introduces negligible overhead to system operation, and that LPM modules also introduce minimal system costs. LPM also facilitates independent evaluation, which we show by comparing the performance of Hi-Fi [20] to a novel provenance collection mechanism.

The rest of this paper is organized as follows. In Section II we survey past proposals for automated provenance collection systems. We unify the functional needs of these disparate projects in Section III, where we present our design for the LPM Framework, including a full security model and analysis for a provenance-aware adversary. Section V introduces Provenance Walls, an LPM module that performs policy-reduced provenance collection. Section VI includes case studies of provenance use cases in which we visualize whole-system provenance collected by LPM, as well as our performance analysis. Discussion follows in Section VII, and we conclude in Section VIII.

II. Automated Provenance Collection

In this work, we consider the security of observed-provenance systems that perform automated collection of provenance, requiring little to no user intervention [7]. We refer to such systems as *provenance monitors*. In past proposals, provenance monitors have operated under a variety of different scopes, assumptions, and goals. We argue that each of these proposals operates under different implicit provenance models, and that our development framework must anticipate and support all of these models.

A. Operational Layer

Provenance collection mechanisms have been implemented at a variety of host layers. To date, we have seen provenance-driven forensic systems that have been implemented as network monitors [4], applications [3], [12], [26], [30], platforms [9], operating systems and kernels [1], [15], [20].

The choice of operational layer is closely linked to the granularity and expressiveness of the provenance collected. Network monitors such as SNooPy [30] and PVP [4] track the provenance of network events, but treat internal host events as opaque. Application layer monitors such as REDUX [3] and Trio [26] excel at capturing workflow context, but cannot observe the underlying system events. PASS [15] and Hi-Fi [20] are able to view workflows within kernel space, and even replay command line workflows in some circumstances [15], but their core mechanisms are unable to understand higher level application functionality. For example, PASS does not understand SQL Engines, and therefore would not be able to replay a workflow that inserts a record into a SQL database.

In spite of the great diversity of these proposals, many are a poor foundation for a trustworthy provenance-aware system. In particular, application-level provenance monitors are popular as easy proof-of-concept implementations, but cannot offer any meaningful assurance that the monitor is *tamperproof*. For example, Hasan et. al.'s SProv library relies on users to instrument their programs with alternate `stdio` functions [12]; a malicious user could easily forge false provenance records in this scheme, even without root access to the host. Zhou et. al.'s SNooPY scheme for network provenance also relies on application instrumentation [30]. While they account for the possibility of compromised hosts through a checks-and-balances system between nodes, they also fail to detect forgeries when multiple compromised hosts collude. Even the proposals for kernel layer provenance monitors suffer from fundamental security concerns. Highly influential works such as Muniswamy-Reddy et. al.'s PASS [15] and Lineage FS [1] do not address the matter of securing provenance metadata. Hi-Fi's LSM-based instrumentation protects its monitor, but fails to protect the userspace components in its TCB, and in fact reduces system security by preventing a MAC policy or capabilities system from being enabled [20]. As a result, any adversary that can compromise a root-owned process is able to subvert the provenance monitor.

For security reasons, it is important that all provenance monitors be anchored in protected kernel space. More generally, provenance monitors should be protected at the lowest possible system layer. This is a necessary prerequisite to providing a tamperproof provenance monitor. However, an unfortunate consequence of this decision is that valuable context is sacrificed at the application layer. To account for this, the PASS system permits the annotation of volunteered provenance data from higher operational layers [15]. We note that this is also an attack vector, and discuss how to securely implement such a mechanism in Section III.

B. Scoping

Beyond operational layer, provenance monitors also consider a variety of combinations of system events, representing a *scope of interest*, while considering activity on other parts of the system to be opaque. For example, PASS' primary goal is to associate provenance metadata with system files; however, since most system objects are represented in Linux as file abstractions, PASS can also capture IPC and network events [15]. Other systems, particularly those that operate above the kernel layer, collect provenance with a reduced scope. The SProv library almost exclusively collects the provenance of file writes [12], while SNooPy treats host-level events as opaque [30], choosing instead to record only the provenance of network packet exchanges. Limited scopes of interest are often valuable, as they are associated with reduced system overhead, and the intended application of the provenance was used to inform the system design.

Using an alternate approach, systems such as Hi-Fi [20] and PASS [15] adopt a wide scope, attempting to offer a comprehensive provenance monitor that observes all system events. However, it is important to note that these proposals cannot anticipate the needs of every provenance application. For example, PASS struggles in handling memory mapping operations, while Hi-Fi elects not to monitor a variety of system events, using just 25 of the LSM Framework's 170 function hooks. One might imagine that these design decisions

Proposal	Layer	Application Context?	File System?	IPC?	Memory?	Network?	Processes?
HI-FI [20]	Kernel (LSM)		✓	✓	✓	✓	✓
Lineage [1]	Kernel		✓	✓		✓	✓
PASS [15]	Kernel (VFS)	Optional	✓	✓		✓	✓
PrIMe [16]	Application	✓		✓			
QUIRE [9]	Platform			✓			
REDUX [3]	Application	✓					
PVP [4]	Network (SDN)					✓	
SNooPy [30]	Application					✓	
SOA [22], [23]	Distributed	✓					
SProv [12]	Application		✓				
Trio [26]	Application	✓					

TABLE I: Automated Provenance-Aware Systems vary by scope, purpose, and operational layer.

may render these systems unsuitable for performing malware analysis, or reasoning about an enterprise's attack surface. It is unlikely that any single proposal will ever produce a one-size-fits-all provenance monitor, due to the great variety of desirable provenance applications.

C. *Collection Paradigms*

Another means of describing a provenance monitor is through the method of which the provenance is generated. That is, once the monitor observes a system event, *what does it do?*

Cryptographic Commitment is a popular paradigm for creating immutable, append-only provenance-aware systems. This is especially the case for application-layer monitors that seek to provide some assurance against adversaries. The SProv library implements Hasan et. al.'s *provenance chain* concept, which creates a tamper-evident chain of provenance records. In this way, a host that becomes compromised is unable to forge provenance records from events that occurred prior to the compromise. SNooPy achieves similar properties by generating append-only tamper-evident logs of network events, and goes a step forward by detecting host compromise by relying on other uncompromised nodes in the network. However, cryptography creates additional overhead, and cannot defend against a compromised host that forges new provenance entries. Moreover, when it is reasonable to place the provenance monitor within the system's trusted computing base, cryptographic commitments are not necessary.

Provenance Stream monitors generate provenance across a variety of simultaneous system events, placing the output into a single stream to be processed by other system components. Stream systems are well suited to the kernel layer. Streams are implemented in the Lineage file system through instrumenting the kernel with `printk`'s, and in Hi-Fi through writing provenance out to user space via a relay buffer. Provenance streams are advantageous in that they reduce complexity and can be stored efficiently, especially when piped through a deduplication algorithm [29]. However, they also require a mechanism for securing the stored provenance. Some proposals have considered this task to be out-of-scope, or have identified the security mechanism as future work [15].

Inline Provenance mechanisms immediately associate generated provenance with the data object that it describes, either by embedding it in the file system or storing it in a database. This is a popular strategy in application layer monitors such as Trio and REDUX. PASS performs inline storage of provenance by associating it with system objects in-memory before eventually writing the provenance to a database on disk. Inline storage can improve query efficiency because provenance is co-located with its data. However, it introduces security challenges as well. Provenance often requires a separate security policy than the data it describes, prompting the need for a provenance-aware security mechanism. In contrast, stream systems like Hi-Fi can store provenance in a single directory, making it more intuitive to protect .

III. Linux Provenance Module Framework

In this section, we present the design of the LPM Framework, bearing in mind the purposes and security needs of past proposals for automated provenance collection. We first present a threat model, demonstrating that provenance monitors' security requirements are similar to those of the classic reference monitor. Next, we leverage the insights of past proposals in this space to define a design space for provenance monitors, ensuring that LPM supports the needs of all stakeholders.

A. *Threat Model*

In this work, we consider an adversary that has gained access to a provenance-aware host or distributed system. We make no assumption about the manner in which access was obtained; they may be an *insider* with legitimate access credentials, or an outsider that has compromised a network application. Once inside of the system, the attacker attempts to go about their business without leaving evidence of their actions in the provenance records. They may attempt to remove those records, insert spurious information into those records, or find gaps in the provenance monitor's ability to record information flows. Because the purpose of provenance is eventually to *apply* it to an important system function, the adversary's purpose on the system may even be to target the provenance monitor itself. The implications and methods of such an attack are area-specific, and so to gain a better understand of them we consider a few examples here:

1) **Scientific Computing**: An adversary may wish to manipulate the provenance records of an e-Science application in order to commit academic fraud, or inject confusion into those records in order to trigger "Climategate"-like controversy [21].

2) **Access Control**: In Provenance-Based Access Control systems, provenance is inspected in the process of mediating over access decisions [19]. An adversary could seek to tamper with provenance history in order to gain unauthorized access, or perform a denial-of-service attack on other users by artificially escalating the security level of data objects.
3) **Networks**: Provenance metadata can also be associated with packets in order to better understand network events in distributed systems [30]. Coordinating multiple compromised hosts, an attacker may attempt to send *unauthenticated* messages so as to avoid provenance generation and perform data exfiltration.

B. Security Goals

In considering the design of provenance-aware systems, it is vitally important that the security of both the provenance monitor and its records be assured [10]. Generally, we assert that the set of security-sensitive operations in a system are a strong base from which to build a set of provenance-sensitive operations. We are not the first to make this observation; McDaniel et. al. liken the needs of a provenance monitor to the reference monitor requirements [14]: tamperproofness, complete mediation, and verifiability [2]. We explore the similarities and dissimilarities between a provenance monitor's needs and *reference monitor* guarantees below:

Tamperproofness. As the provenance will eventually be applied in some way to help users better understand their data, it naturally follows that provenance monitors are an adversarial target. Adversaries may attempt to tamper with provenance records in order to hide their tracks, such as removing records of their illicit activities or injecting confusion with false records. It is therefore important that the provenance monitor be tamperproof; that is, the provenance mechanism cannot be modified or manipulated by processes in user space. This requirement extends to the entire trusted computing base of the provenance monitor (e.g., logs, policy, helper processes).

Complete Observation. If the provenance monitor is not able to capture all forms of information flow on a system, adversaries may also attempt to evade detection by only taking actions that do not appear in the provenance record. This means that provenance monitors seem to require *complete mediation* of a system in order to be effective; however, there are two problems with this assertion. First, since provenance monitors exist to passively record system events, complete mediation is an unnecessary amount of power. Controlling access to system resources is not needed to collect provenance. Second, as a practical matter, there cannot be two system mechanisms that possess the complete mediation property. Because provenance only needs to watch all flows, not control them, we instead argue that provenance monitors require **complete observation** of system activity. To enforce this distinction, provenance monitors should be placed within the protected domain of the reference monitor, record provenance only on actions that have been authorized by the reference monitor, and avoid interference with the mediation of the reference monitor.

Verifiability. If a provenance monitor was conceptually complex, it would be difficult to ensure that it was tamperproof, or that it observed all provenance-sensitive system flows. This

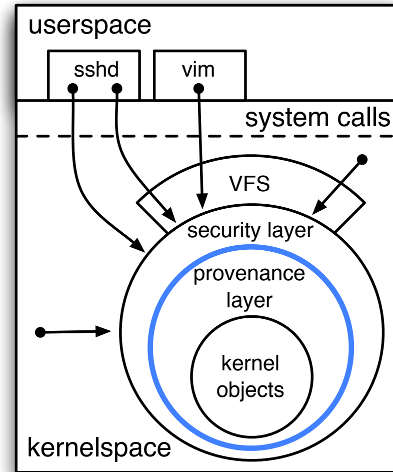

Fig. 1: The LPM Framework runs within the LSM Framework, observing system activity without interfering with the work of the security mechanism.

underscores the importance of having a provenance mechanism that is small enough to be verifiably correct. In this context, correctness means that the mechanism can be shown to collect accurate provenance over all provenance-sensitive tasks. In the presence of a policy component, correctness implies that the mechanism strictly follows the collection procedure defined by the policy specification. The verifiability requirement also extends to the entire TCB of the provenance monitor.

We know from decades of research and development that these properties are surprisingly hard to obtain. Rather than start from scratch with provenance, we present a design in Section III-C that leverages the established Linux Security Module (LSM) Framework [27], [28] in order to bootstrap trust in provenance-aware systems.

The requirements of a provenance monitor extend beyond the classic goals of a reference monitor. Past work in provenance-aware systems has also identified several additional desirable properties:

Secure Channel. Many existing provenance collection mechanisms require a secure communication between networked hosts [5], [14], [20], [30]. Failure to provide such a channel could result in corrupt provenance records or the leakage of sensitive provenance information. Provenance monitors must be able to communicate over an adversary-controlled network without being vulnerable to eavesdropping or tampering attacks.

Secure Annotation. While operating at a lower operational layer enhances the completeness and security of provenance collection, this comes at the sacrifice of fine-grained application layer information. In PASS, Muniswamy-Reddy et. al provide a mechanism that allows for applications to volunteer provenance to the kernel mechanism, allowing for the recovery of this context. Unfortunately, PASS did not provide a means of ensuring integrity and authenticity of these annotations. This

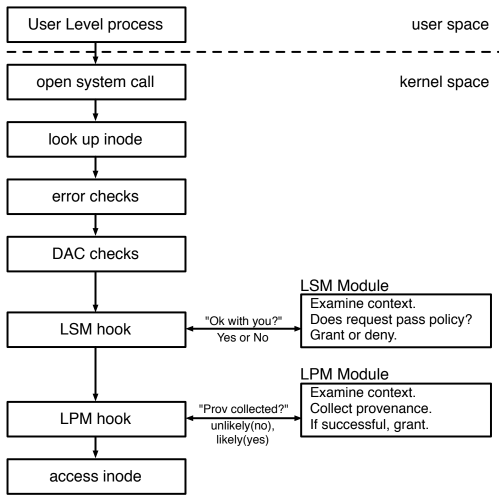

Fig. 2: LPM Hook Architecture for the `open` system call.

introduces new vulnerabilities to PASS, as an attacker could exploit the annotation mechanism, violating the tamperproof requirement. In order to safely accommodate application layer annotations, a provenance monitor must feature gateways that permit secure annotation of volunteered application provenance in such a way that preserves the integrity of the provenance history.

C. Design

Gaining confidence in the security of a system takes time, requiring community access and independent review. In order to expedite this process, we choose to implement LPM as a parallel framework to LSM. This makes LPM easy to understand, and allows it to benefit from the decade of community review that LSM has enjoyed. Where LSM seeks to mediate access to internal kernel objects, LPM seeks to observe access to those objects, creating provenance records that make assertions such as ”Subject `S` performed kernel operation `OP` on internal kernel object `OBJ`."

Like LSM, our provenance framework is “truly generic" [28], in that loading a different provenance monitor is as simple as loading a new kernel module. It is also minimally invasive, requiring roughly the same amount of kernel instrumentation as was required by LSM. A diagram of LPM's observation point can be found in Figure 1. This Figure is an update to the design in the original Hi-Fi work, in which there was not a dedicated provenance layer and the security layer was overloaded to collect provenance [20].

Provenance Hooks

The LPM patch introduces a set of *provenance hook* functions in the Linux kernel. These hooks behave similarly to the LSM Framework's *security hooks* in that they facilitate modularity, and take no action unless a module is enabled. Throughout the kernel, the LPM patch places a provenance hook directly beneath each security hook. In doing, we assure that the provenance hooks avoid time to check to time of use (TOCTOU) races [6], and that provenance is collected just before the kernel actually performs the requested service [28]. By first checking to ensure that the LSM has permitted the operation, we fulfill the *complete observation* requirement while simultaneously ensuring the accuracy of the collected provenance, as rejected access attempts will not be included in the record. An overview of the hook architecture is depicted in Figure 2, and the complete list of provenance hooks is shown in Table II.

Hooks	Count	Purpose
BPRM	5	Observe program execution operations.
SB	19	Observe filesystem operations.
Inode	38	Observe inode operations.
File	11	Observe file operations.
Dentry	1	Observe dentry open.
Task	30	Observe task operations.
Netlink	2	Observe Netlink Message.
Unix	2	Observe Unix Domain Networking.
Socket	35	Observe socket operations.
IPC	10	Observe System V IPC Message Queues.
SHM	5	Observe System V Shared Memory Segments.
SEM	19	Observe System V Semaphores.

TABLE II: Summary of LPM's Function Hooks.

```
1  int vfs_readdir(struct file *file, filldir_t
       filler, void *buf){
2    struct inode *inode =
         file->f_path.dentry->d_inode;
3    int res = -ENOTDIR;
4    if (!file->f_op || !file->f_op->readdir)
5      goto out;
6
7    res = security_file_permission(file, MAY_READ);
8    if (res)
9      goto out;
10
11   provenance_file_permission(file,
         file->f_provenance, MAY_READ);
12
13   res = mutex_lock_killable(&inode->i_mutex);
14   if (res)
15     goto out;
16
17   res = -ENOENT;
18   if (!IS_DEADDIR(inode)) {
19     res = file->f_op->readdir(file, buf, filler);
20     file_accessed(file);
21   }
22   mutex_unlock(&inode->i_mutex);
23   out:
24     return res;
25 }
26
27 EXPORT_SYMBOL(vfs_readdir);
```

Fig. 3: Example provenance hook from the vfs_readdir function in `fs/readdir.c`. Our framework inserts Line 12.

An example hook placement is shown in Figure 3. The `vfs_readdir` functions attempts to read a file's directory, placing it in the `buf` pointer. LPM introduces Line 12 of the function; immediately after the `security_file_permission` affirms that the subject has permission to take this action (Lines 7-9), `provenance_file_permission` is called so that LPM can record the event.

Striving for *complete observation*, LPM hooks are not

```
1 policy_module(uprovd, 1.0.0)
2
3 ########################################
4 #
5 # Declarations
6 #
7
8 type uprovd_t;
9 type uprovd_exec_t;
10 init_daemon_domain(uprovd_t, uprovd_exec_t)
11
12 permissive uprovd_t;
13
14 type uprovd_log_t;
15 logging_log_file(uprovd_log_t)
16
17 type uprovd_rw_t;
18 files_type(uprovd_rw_t)
19
20 ########################################
21 #
22 # uprovd local policy
23 #
24 allow uprovd_t self:process { signal };
25
26 allow uprovd_t self:fifo_file rw_fifo_file_perms;
27 allow uprovd_t self:unix_stream_socket
      create_stream_socket_perms;
28
29 manage_dirs_pattern(uprovd_t, uprovd_log_t,
      uprovd_log_t)
30 manage_files_pattern(uprovd_t, uprovd_log_t,
      uprovd_log_t)
31 logging_log_filetrans(uprovd_t, uprovd_log_t, { dir
      file })
32
33 manage_dirs_pattern(uprovd_t, uprovd_rw_t,
      uprovd_rw_t)
34 manage_files_pattern(uprovd_t, uprovd_rw_t,
      uprovd_rw_t)
35
36 domain_use_interactive_fds(uprovd_t)
37
38 files_read_etc_files(uprovd_t)
39
40 miscfiles_read_localization(uprovd_t)
```

Fig. 4: SELinux Policy Module for Hi-Fi: Type Enforcement for protecting Hi-Fi's user space daemon.

intended to mediate system access, or alter the control path of the function in any way. In designing LPM, we considered enforcing this requirement structurally by having hooks return `void`, as checking a return value could alter the control flow of the function. Additionally, hooks could be passed `const` parameters as often as possible, so that developers did not accidentally modify system objects. Unfortunately, provenance hooks need to return error codes, such as when an attempt to allocate memory for provenance record creation fails. When this error occurs, the associated system call cannot be permitted to continue, or else the provenance history would be incomplete.

We identify an important exception to the passive observation rule, where LPM must modify system objects. In order to provide a secure channel for communication between provenance monitors, LPM must be permit the modification of network messages. Several existing proposals require the ability to securely embed or encapsulate provenance information within network packets [20], [30]. We see this as a necessary requirement for facilitating provenance-aware distributed systems, as such systems require the ability for provenance monitors to securely communicate. Therefore, LPM leverages Netfilter hooks that permit packet modification as messages enter and exit the system. Pre-empting other functions, LPM is nearly the last function to process packets before transmission, and nearly the first function to process packets on reception, making this procedure transparent to the rest of the host system.

```
1
2 /sys/kernel/debug/provenance0   --
      gen_context(system_u:object_r:uprovd_rw_t,s0)
3
4
5 /usr/bin/uprovd   --
      gen_context(system_u:object_r:uprovd_exec_t,s0)
6
7 /var/log(/.*)?
      gen_context(system_u:object_r:uprovd_log_t,s0)
```

Fig. 5: SELinux Policy Module for Hi-Fi: File Contexts for protecting Hi-Fi's user space daemon.

LPM hooks are not limited to those that mirror security hooks. While the primary hook set for LPM will be the approximate 170 hooks that are included in the LSM Framework, we identify at least once instance in which the introduction of new hooks is necessary: Hi-Fi identifies a fundamental need for provenance to ensure the correct ordering of packets received by networked hosts, ensuring this property by introducing a new security hook, `socket_post_recvmsg`. The absence of such a security hook implies that packet ordering is not security-sensitive, and LPM modules must be trusted anyway, so the new provenance hook does not impact the security of the system. As we introduce the LPM framework, we will continue to investigate new LPM hooks in response to developer needs; however; we note that our design constraints may make it impossible to fully satisfy every provenance project. The original LSM project made a similar concession, opting for a primarily *restrictive* hook set in order to minimize invasiveness, sacrificing full support for *permissive* security models such as POSIX.1e capabilities logic [28].

Securing the Provenance Monitor

Provenance security is its own rich area of study [5], [10], [11], [12], [30], and yet proposals for provenance-aware systems have at times failed to incorporate security into the foundations of their design [15]. Through the LPM Framework, developers no longer need worry about implementing their own security mechanisms. Instead, they can simply write policies for existing mandatory access control (MAC) mechanisms in order to protect the provenance TCB. We envision a future in which new provenance modules will be released with an associated MAC policy, thus allowing for faster, more secure deployment of provenance-aware systems. We demonstrate this ability by creating an SELinux policy module for an LPM-based re-implementation of Hi-Fi, a portion of which is shown in Figures 4 and 5. This interplay between provenance modules and security policy allows for fantastic new opportunities for

provenance. We introduce a new method of leveraging this mechanism in Section V through creating a policy-driven provenance module.

Disclosed Provenance

Collecting provenance at the kernel layer leads to some loss in application context. This is because data representations in an application often do not map perfectly to kernel objects. For example, a database engine organizes data in records, but may write the records to a single flat file; the kernel is oblivious to the manner in which each individual record was derived. To address this problem, PASS supports annotating kernel layer provenance with disclosed provenance that is volunteered from user space [15]. However, they do not provide a mechanism to evaluate the integrity and authenticity of the volunteered provenance.

We propose that *provenance itself* can be used as a means to validate volunteered provenance. If an object representing volunteered provenance was generated on an LPM-instrumented system, provenance was collected that describes the history of that object. If LPM was given out-of-band knowledge of how the volunteered provenance was supposed to have been generated, it would be able to evaluate the integrity of the annotation prior to accepted it into the official history. This would restrict an attacker's ability to exploit volunteered provenance as a means of subverting the provenance monitor.

LPM will support this functionality by providing a *gateway* that upgrades volunteered provenance to a high integrity state before appending it to the official provenance history. When developers release a provenance-aware application, they can make it compatible with LPM in the following way. First, the application is instructed to write each annotation out as a separate file to a provenance directory in the `securityfs` file system. Second, the developer releases a generalized external specification that describes the *system-level* provenance of the application's annotations. This specification is distinct from the MAC policy that was previously discussed in this section, and is also different from the provenance policy we make use of in V. When LPM receives the annotation file, it reconstructs a provenance graph of the file using provenance that was collected by the kernel. It then compares the graph to the external specification, appending the annotation to the official history if validation is successful. Optionally, annotations that fail validation can be appended but marked as untrusted.

We will now explain in greater detail how LPM performs this validation. Figure 6 shows the raw provenance records associated with the file `annotation`, which was written by a provenance-aware application `annotate`. Each record includes a provenance identifier in brackets, the user controlling the event in parenthesis, the event type, and then the partition identifier, inodes, and dentries. This program loads `libc-2.12.so` (Line 2), writes `annotation` (Line 3), changes the permissions of the file (Line 4), and then links the newly created inode to an LPM-controlled directory in `securityfs` (Line 5). LPM is able to automatically recover this provenance when it reads the file `annotate`.

Figure 7 contains an external specification that describes the volunteered provenance created by the program `annotate`. Each line is a generalized description that maps to the corresponding line in Figure 6. The `i_SECURITYFS` and `d_SECURITYFS` map to stored configuration values for the inode and full path of the `securityfs` directory, while all other variables are wildcards that are assigned based on the raw provenance at the time of validation. We are currently developing a script that processes raw provenance in order to produce these specifications.

```
1 [2678] (500) exec
     cbaadf33-c28f-4336-b3679b9028edef42 917695
     ./annotate
     /sys/kernel/security/provenance/annotation <ENV>
2 [2678] (500) fperm R
     cbaadf33-c28f-4336-b3679b9028edef42 83281
     /lib/libc-2.12.so
3 [2678] (500) fperm W
     cbaadf33-c28f-4336-b3679b9028edef42 917694
     annotation
4 [2678] (500) setattr
     cbaadf33-c28f-4336-b3679b9028edef42 917694
     500:500 m=100440
5 [2678] (500) link
     cbaadf33-c28f-4336-b3679b9028edef42
     917843:/sys/kernel/security/provenance to
     917694:/sys/kernel/security/provenance/annotation
```

Fig. 6: A provenance-aware application `annotate` writes the file `annotation` to an LPM-owned directory in `securityfs`.

```
1 [provid1] (uid) exec    <root_partition> <inode1>
     annotate <d_SECURITYFS>/<dentry1> <ENV>
2 [provid1] (uid) fperm R <root_partition>
     <inode2>:/lib/libc-2.12.so
3 [provid1] (uid) fperm R <root_partition>
     <inode3>:<d_SECURITYFS>/<dentry1>
4 [provid1] (uid) setattr <root_partition> <inode5>
     uid:gid m=100440
5 [provid1] (uid) link  <root_partition>
     <i_SECURITYFS>:<d_SECURITYFS> to
     <inode3>:<SECURITYFS>/<dentry1>
```

Fig. 7: An external specification that validates the volunteered provenance shown in Figure 6.

D. Prototype Modules

Coinciding with the release of the LPM Framework, we will be releasing a re-implementation of the Hi-Fi system. By acquiring source code from the authors, we were able to transform Hi-Fi from an LSM into an LPM. The original Hi-Fi code base required 2278 lines of code; 723 lines needed to be modified in the transition. These changes were primarily cosmetic, e.g., the `security_file_permission` function was renamed to `provenance_file_permission`. We also modified Hi-Fi's packet header modification routines to make them more consistent with the Red Hat Kernel's network stack.

Administrators and developers can enable, disable, and choose between provenance modules using the menuconfig user interface prior to kernel compilation. Similar to the Integrity Measurement Architecture (IMA), provenance options can be found as a submenu under security options. A

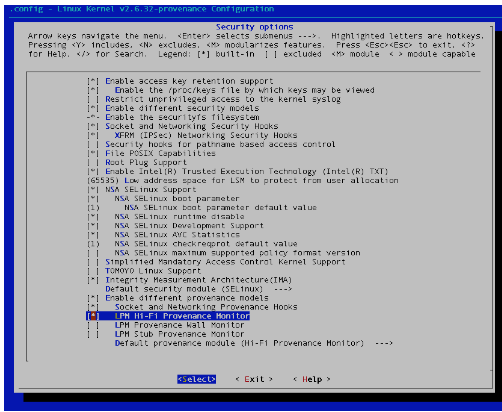

Fig. 8: LPM modules can be enabled under “Security Options” in the kernel configuration menu.

screenshot of the modified “Security Options" menu can be found in Figure 8.

IV. Security Analysis

In designing LPM, we set out to provide 5 security assurances: *tamperproofness*, *complete observation*, *verifiability*, *secure channel*, and *secure annotation*. Here, we measure the extent to which LPM achieves these goals against a provenance-aware adversary. We anchor our trust of LPM security in a Linux Security Module, meaning that we make the assumption that a system administrator has selected, enabled, and properly deployed a MAC policy. This MAC policy must protect the kernel and enforce least privilege over the LPM's trusted computing base.

Tamperproofness. Because of the MAC policy, an attacker that takes control of a user space process would not be able to subvert the provenance monitor. MAC policies must prevent the loading or unloading of kernel modules (e.g., SELinux default policy), meaning that even with root privilege the attacker could not access kernel space to modify or disable LPM. Likewise, the attacker would not be able to edit the provenance logs. We demonstrated in Section III-C that a MAC policy could be created that limited the ability to write to these logs to just the provenance monitor's user space daemon. Our daemon implementation contains only 143 lines of code, making it unlikely that an attacker would be able to find an exploitable bug. The only influence an attacker could exert on the provenance system would be to take actions on the system, which would be correctly recorded into the log. This creates the potential for a denial of service attack, but this is not useful ability to a covert adversary.

Complete Observation. LPM monitors a superset of the system events that have been deemed to be security-sensitive by the LSM project [28], which has benefited from over a decade of community review. As a result, we provide strong assurances that LPM can observe all accesses to *internal* kernel objects, provided that they are of a security-sensitive nature. As interest provenance-aware systems grow, it may be that systems accesses of an exclusively *provenance-sensitive* nature come to light. LPM is positioned to easily accommodate these needs through the introduction of additional provenance hooks. A known family of exclusively provenance-sensitive operations is high-level application context that is not observable from the kernel; to accommodate this need, LPM provides support for secure annotation of application layer provenance.

Verifiability. LPM's modular composition means that a provenance monitors's logic is centrally located, our adoption of the LSM security hooks means that many kernel developers would be able to readily understand the functionality of the module. The verifiability of a particular LPM module is bounded by its complexity. LPM modules need be no larger than LSM modules; our Hi-Fi implementation is 1400 lines of code, and our Provenance Wall implementation is 2500 lines of code.

Secure Channel. Through use of Netfilter hooks [24], LPM is able to mangle network messages in order to support distributed provenance collection. LPM can embed provenance data immediately before a packet is sent, and immediately after a packet is received, making it impossible for an adversary-controlled application running in user space to interfere. In the event that secrecy is also required, an LPM module can implement a user space component that establishes an TLS/SSL tunnel to other provenance-aware hosts. This component can also be secured with a MAC policy. It is also worth noting that many security-critical system components rely on a user space helper process (e.g., SELinux, TPMs).

Secure Annotation. Provenance that is volunteered from the application layer is an attack vector, as the provenance monitor did not observe the events first hand or collect the resulting provenance. Fortunately, we have shown that the integrity of volunteered provenance can be evaluated by leveraging our trust in the kernel layer provenance monitor. LPM includes a gateway that allows applications to write a provenance file out to a `securityfs` directory, resulting in provenance collection within the kernel. Before including the volunteered provenance in the official history, the monitor can validate the system-level provenance of the volunteered provenance file against a developer-provided external specification. When making use of this gateway, it is important to note that the provenance-aware application falls into the provenance monitor's TCB. If the application contains exploitable bugs, an adversary could take control of the application in order to forge provenance records. However, the adversary would still be unable to edit or remove previously-collected provenance, and their access to internal kernal objects would still be accurately recorded.

Because LPM modules have full access to the kernel's address space, the security and trustworthiness of any provenance module will depend on the skill and diligence of the developer. In order to ensure complete observation, an LPM module must be able to send an access-denying error code if they are unable to successfully generate a new provenance record for an event. Otherwise, the provenance history would be incomplete. Moreover, This weak module safety means that an administrator must rust the module provider before making the decision to load an LPM. However, this limitation is shared by any other fully privileged kernel module.

V. Policy-Reduced Provenance

A fundamental issue in automated provenance collection is the introduction of storage overhead. Provenance can quickly

grow to dwarf the data it describes. For example, [15] shows that a small change to a Linux kernel source file generates an extra 2 KB of provenance in PASS. Even worse, automated provenance monitors are prone to collecting information that is completely uninteresting or irrelevant to the system's purpose [7], making it harder to justify the incurred storage costs. *Provenance pruning* has been proposed as a method of reducing this overhead, in which the provenance monitor automatically reduces storage costs by omitting or contracting provenance. The notion of *interesting* provenance is often domain-specific, leading Braun et. al. to call for a policy-driven method of provenance pruning, but unfortunately such a proposal has yet to appear in the literature.

Leveraging the LPM Framework, we now introduce a novel method of *policy-reduced provenance* that only collects metadata over an administrator-defined subset of system activities. We do so by leveraging the insight that, on many systems, an existing MAC policy can be used to quickly identify the *interesting* system components. For example, Jaeger et. al. show that SELinux policies can be mined to discover application-specific trusted computing bases (TCB) [25]. Alternately, an administrator may be interested in tracking system resources that are potentially under adversary control, which is also captured by MAC labels (e.g., `unconfined_t` in SELinux).

We now introduce a new provenance monitor, **Provenance Walls**, that implements this functionality. Provenance Walls is designed to collect a complete provenance history over any set of SELinux labels that is provided by an administrator. When a hook is triggered in the kernel, the *provwall* LPM module collects the security context of each system object involved in the event, then selectively generates provenance after comparing these contexts to a policy. Provenance Walls relies on both a kernel-level provenance monitor and an active Linux Security Module, which would not have been possible prior to the introduction of the LPM Framework. The module introduces two components, a **policy generator** and a **policy-driven provenance monitor**, which are shown in a design overview in Figure 9. This system allows administrators to provide a policy that can describe as little as a single application, resulting in dramatic runtime and storage savings.

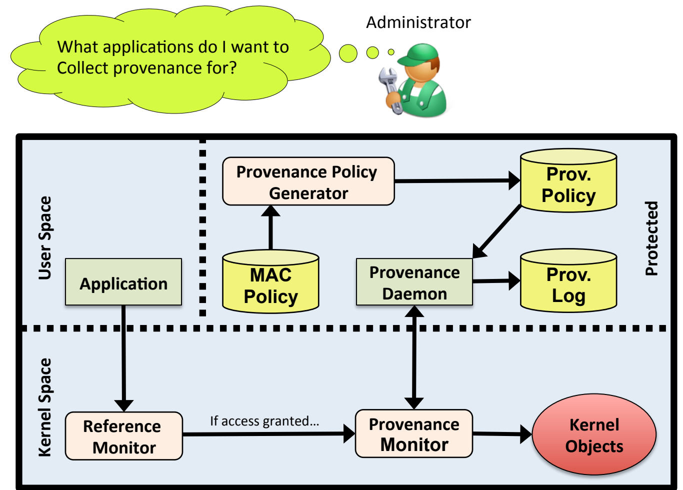

Fig. 9: Provenance Wall Design Overview.

A. Policy Generation

Because provenance must offer a complete history of the data it describes, selective provenance collection is a difficult problem. Failure to collect all of the necessary information could result in gaps in a provenance history, rendering it unfit for later use. For this reason, it is important that our method for policy generation be able to produce complete provenance records.

To do so, we extending Jaeger et. al.'s *Integrity Walls* method of mining SELinux policies to identify attack surfaces and TCBs [25], adapting it in order to create a set of labels that produce complete provenance records. Integrity Walls builds a TCB for subject applications s by partitioning a system policy P into a set of trusted labels I_s and an untrusted set O_s. Given a base SELinux policy, an application policy module, and its dependency modules, Integrity Walls identifies the following groups of labels: *executable writers* that have permission to write s's executable file, *kernel subjects* with permission to write s's underlying kernel objects, and *helper subjects*, which are distinct applications whose subject labels appear in s's policy module and are trusted by s. Together, these labels from a trusted computing base of the subject application of s.

There are several ways that this methodology can be applied to provenance collection. First, a provenance policy can be created using I_s that allows an administrator to reason about the core functions of a system. I_s captures kernel activity, as well as the activity of a given set of applications s, while avoiding extraneous activity that falls outside of the TCB. This provenance could be used assist in code optimization, attack impact analysis, or system configuration. Second, a provenance policy can be created using O_s, allowing an administrator to monitor the activities of users and potentially adversary-controlled resources. This provenance could be used in attack surface analysis, assessing data quality, enforcing data segmentation and performing malware analysis.

While both approaches are valid, we adopt the latter approach. By performing additional processing O_s, we are able to extend the forensic abilities of Integrity Walls; rather than simply identifying attack surfaces [25], Provenance Walls can produce provenance graphs of adversary-controlled inputs into a system's TCB.While we use SELinux types as the basis for our policy, we believe that our approach is generally applicable to other kinds of security contexts as well.

B. Policy-Driven Provenance Monitor

Provenance Walls introduces Provwall, a new LPM-based module for provenance collection. Provwall makes use of the same hooks as Hi-Fi for observing file system, IPC, memory, network, and process events. Rather than naïvely generating provenance for all system event, however, it first performs an efficient check that compares an administrator-defined policy against the security contexts of all the system objects associated with the event. If any of the objects appear in the policy, provenance is generated; otherwise, no further action is taken.

1) Loading the Policy: Similar to many LSMs, Provwall loads its policy through a special-purpose virtual filesystem,

```
1  /*
2   * Sending a message to an XSI message queue
3   */
4  static int provwall_msg_queue_msgsnd(struct
       msg_queue *msq, struct msg_msg *msg,
5                  int msqflg) {
6          const struct cred_provenance *curprov =
               current_provenance();
7          const struct msg_provenance *msgprov =
               msg->provenance;
8          struct provmsg_mqsend logmsg;
9
10         if (curprov->flags & CPROV_OPAQUE)
11                 return 0;
12
13         if ( policy_check_ipc(&msq->q_perm) == 0
14              && policy_check_msg(msg) == 0
15              && policy_check_current_creds() == 0)
16           return 0;
17
18         msg_initlen(&logmsg.msg, sizeof(logmsg));
19         logmsg.msg.type = PROVMSG_MQSEND;
20         logmsg.msg.cred_id = curprov->cpid;
21         logmsg.msgid = msgprov->msgid;
22
23         write_to_relay(&logmsg, sizeof(logmsg));
24         return 0;
25 }
```

Fig. 10: Provwall checks system objects' security contexts against a policy before generated provenance records. In the example above, these checks occur in Lines 13-16.

securityfs, that is created in memory during system boot. The policy is automatically written into securityfs by *uprovd* [20], the user space daemon responsible for reading provenance records out of a kernel buffer and writing them into memory. The uprovd launch command is included in `/etc/rc.local`, and immediately writes policy to securityfs upon being executed. Each time a provenance hook is triggered in the Provwall module, a check is performed to see if the policy file has yet been written by uprovd. Once the policy file is detected, the module parses the SELinux types and loads them into a linked list.

2) Policy Checks: Prior to generating a provenance record of an event, each Provwall hook function first checks the security contexts of the associated objects. If *any* of the contexts match the policy, a provenance record is created. Otherwise, the function returns. An example of this functionality for the `msg_queue_msgsnd` provenance hook is shown in Figure 10; if any of the `policy_check` functions return true, a record is generated.

Policy checks work as follows. Inside of the function, each of the object's *security ids (SIDs)* is mapped into a security context character string. To simplify this process, we exported a new function from SELinux, `selinux_sid_to_string`. We do not feel that this impacts system security, as `selinux_string_to_sid` is already exported, and these security labels can also be accessed from the command line via the *getfattr* utility. After the context is recovered, it is parsed to recover the SELinux type. This type is then compared to the types in the policy linked list. If a match is discovered, the policy check returns true, otherwise it returns false.

Performing string comparisons during the policy check is suboptimal; it would have been much faster and more desirable to perform integer comparisons of SIDs. Unfortunately, in addition to defining a set of SELinux types in the TCB, SID comparisons would have required enumerating the space of possible SELinux roles, domains, and security levels that were also pertinent to the TCB. This would have drastically increased the size and complexity of the policy, leaving us open to the possibility of misconfiguration. Instead, we reduce the number of string comparisons required by our system by embedding a boolean tracking flag in each object's provenance struct. When the object is first inspected by Provwall, it is set to either true or false, after which time the policy decision becomes static. The tracking flag need only be re-evaluated in the event of a policy update, which requires a system re-boot in our current implementation.

3) Early Boot Provenance: The Linux kernel's security subsystem, which loads and registers an LSM and then an LPM, is initialized before both the virtual file system layer and user space. As a consequence, both the policy and the provenance stream relay are not available on start. To address this problem, [20] stores early boot provenance in a separate boot buffer, and registers a callback function to be executed once the VFS has been initialized. Once called, the function flushes the buffer into the relay. We extend this solution to account for the fact that Provwall needs to wait for the policy to be loaded before it can write provenance.

In early boot, Provwall conservatively creates provenance records for each system event, storing them in the buffer along with the associated SIDs of the event. Once Provwall is notified that the policy is loaded, it replays the buffer and performs policy checks of each of the associated SIDs, flushing only those events to the relay that represent a policy match. This leads to a larger early boot buffer when compared to [20].

VI. Testing and Functionality

A. Case Study: Provenance Visualization

The fundamental application of provenance is to understand the derivation of a data object. Here, we include several visualizations of object lineage that illustrate the power of collecting whole-system provenance with LPM. Each visualization serves as a succinct summary of the object it describes. The provenance for these visualizations was collected using the Hi-Fi and Provwall LPM modules. The visualizations were generated automatedly using a Python script that output provenance graphs in the GraphViz `dot` format. The script accepted a provenance log and a *query* in the form of a dentry, inode, or process for which to generate lineage. For the purpose of a succinct visualization, we omitted much of the contextual information being collected by the LPM module. For example, the following graphs omit the *iperm X* records that represent a process crawling through directories before opening a file, but the *fperm* records in which the file is opened have been included.

Benchmark script: Provenance can be used to explain the behavior of an application. In Section VI-B1, we use a script that microbenchmarks the Linux system calls using `strace`. The provenance of this script is shown in Figure 12. The `bench` binary runs a user-specified number of times

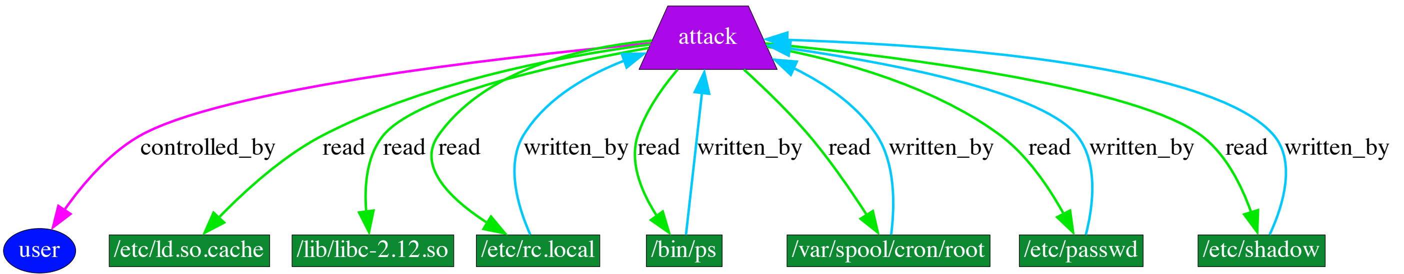

Fig. 11: A provenance graph for a malicious script that obtains persistent access and creates a machine backdoor.

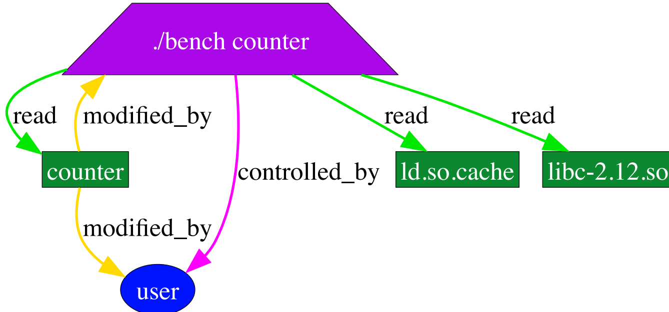

Fig. 12: A provenance graph for the microbenchmark routine that we use in Section VI-B1.

by reading the `counter` file. After executing, it decrements `counter` and then forks a new version of itself. `bench` loads `libc-2.12` and `ld.so.cache` each time it executes.

Malicious script: Provenance can also explain the impact of an attack. Figure 11 shows the provenance graph of a malicious script that has been executed on our provenance-aware system. The script makes several attempts to obtain persistence on the system, adding a line to `/etc/rc.local`, rewriting `/bin/ps`, and adding a cron job in `/var/spool/cron`. It then creates machine backdoor by modifying `/etc/shadow` and `/etc/passwd` to create a new root user.

B. Performance

We evaluated the performance of LPM, Provenance Walls, and our Hi-Fi re-implementation, first by microbenchmarking Linux system calls with different modules enabled, and then by performing a kernel compilation macrobenchmark. Benchmarking was performed on a KVM virtual machine whose kernel was recompiled between trials. The VM was provisioned with 4 GB memory, 4 VCPUs, and a qcow-formatted virtual disk that was stored on a RAID. When benchmarking Provenance Walls, we a policy containing 2015 security types. Provenance Walls was benchmarked under 2 conditions: once when the events fell within the policy (tracked), and once when the events fell outside of the policy (untracked).

1) Microbenchmarks: We performed two sets of microbenchmarks, one with `LMBench` and one that measured system call latency with `strace`. The reason for this is that we encountered stability issues with LMBench when Provenance Walls was enabled. When LMBench began to measure bandwidth, Provenance Walls triggered a kernel panic.

Test Type	RHEL2.6.32	RHEL2.6.32-LPM	RHEL2.6.32-HiFi
Process tests, times in μseconds, smaller is better:			
null call	0.65	0.6 (-2%)	0.6 (-2%)
null I/O	0.78	0.8 (3%)	0.8 (0%)
stat	4.72	4.4 (-7%)	4.4 (-7%)
open/close	7.13	7.0 (-1%)	7.2 (1%)
sig inst	1.1K	1.1 (1%)	1.1 (0%)
sig handl	2.37	2.3 (-1%)	2.4 (2%)
fork proc	695	689 (-1%)	742 (7%)
exec proc	1940	1971 (2%)	1999 (3%)
sh proc	5520	5376 (-3%)	5523 (0%)
File and VM system latencies in μseconds, smaller is better:			
0k file creat	47.50	45.8 (-4%)	46.0 (-3%)
0k file del	17.20	17.6 (2%)	16.6 (-3%)
10k file creat	87.70	84.3 (-4%)	84.6 (-4%)
10k file del	24.70	24.4 (-1%)	25.0 (1%)
mmap lat	40.8K	41.4 (1%)	40.7 (0%)
prot fault	0.75	0.7 (-7%)	0.8 (8%)
page fault	2.29	2.6 (14%)	2.3 (1%)

TABLE III: Microbenchmarks for LPM Framework and modules, taken with LMBench: RHEL2.6.32 is an unmodified kernel; RHEL2.6.32-LPM is a kernel with the patch applied, but no module enabled; RHEL2.6.32-HiFi is a kernel with our Hi-Fi re-implementation enabled; RHEL2.6.32-Provwall (Out) is a kernel with Provwall enabled, but the policy does not track the microbenchmarks; RHEL2.6.32-Provwall (In) is a kernel with Provwall enabled and the policy does track the microbenchmarks.

While we investigate this problem, we created an alternate benchmark script: a small binary that we ran with `strace`. The binary triggered each major system call 100,000 times, and the resultant strace output was parsed and averaged upon completion. LMBench and Strace results were confirmed to be representative by repeating 10 iterations of the full benchmark suite.

Table III show the LMBench results for each provenance-aware kernel, with a percent overhead calculation against the vanilla Red Hat Enterprise Linux 2.6.32 kernel. The performance of both the LPM architecture and the Hi-Fi module fall well within the experimental error; it isn't clear from these measurements that there is any observable performance costs of LPM from user space. Table IV shows that, for the strace microbenchmarks, performance was actually best for Provenance Walls in the tracking condition. We attribute this result to experimental error, perhaps due to cache collision

Test Type	Base	LPM Hooks	Hi-Fi	Provwall (Untracked)	Provwall (tracked)
open	32.20	29.9 (-7%)	32.7 (2%)	34.1 (6%)	24.1 (-25%)
close	21.30	19.8 (-7%)	20.4 (-4%)	20.9 (-2%)	16.2 (-24%)
read	28.50	27.2 (-5%)	28.8 (1%)	28.5 (0%)	22.3 (-22%)
write	46.40	43.8 (-6%)	45.2 (-3%)	45.7 (-2%)	35.6 (-23%)
creat	91.9K	85.1 (-7%)	85.9 (-7%)	161.8 (76%)	128.0 (39%)
rename	44.60	41.6 (-7%)	46.3 (4%)	48.2 (8%)	33.9 (-24%)
unlink	59	58.4 (-2%)	61.4 (4%)	61.7 (4%)	46.2 (-22%)
execve	563	560.0 (-1%)	591.0 (5%)	585.0 (4%)	465.0 (-17%)

TABLE IV: Overhead for the LPM Framework and modules: Base refers to benchmarks from the vanilla RHEL 2.6.32 kernel; LPM Hooks to the modified kernel without a module enabled; Hi-Fi to our LPM-based Hi-Fi re-implementation; Provwall (Untracked)/(tracked) to Provenance Walls depending on whether or not the benchmark routine triggered a policy hit.

Kernel	Build Time	Overhead
Base	8.7 min	
LPM	8.7 min	
Hi-Fi	8.9 min	2.2%
Provwall (Untracked)	8.95 min	2.8%
Provwall (Tracked)	8.9 min	2.2%

TABLE V: Kernel Compilation Macrobenchmarks

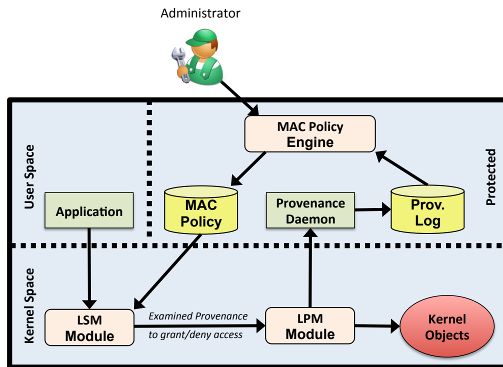

Fig. 13: A design sketch for an LPM/LSM system that implements provenance-based access control.

anomalies [13]. However, there is one valuable measurement shown in this table – overhead is exceptionally high for the `create` system call, regardless of whether the call is in the tracked or untracked condition. This is intuitive, because Provenance Walls cannot used cached results for a newly created file. This results in the module needing to traverse the entire policy.

2) Macrobenchmarks: The results of the kernel compilation macrobenchmark appear in Table V. The first and most important observation we make is that, in spite of the excessive overhead for the `create` call, Provenance Walls fairs very well under realistic workload. It performs comparably to Hi-Fi. Our Hi-Fi benchmarks are also consistent with those from the original study [20]. Interestingly, Provenance wall fairs better in the tracked condition than in the untracked condition. We attribute this to the fact that failed policy checks are more expensive than successful policy checks.

VII. Discussion

A. Enabling New Provenance Applications

The LPM Framework has opened the doors to a variety of exciting new provenance applications. Through the introduction of Provenance Walls, we have shown that the interplay between security and provenance permits exciting new possibilities in the area of attack surface analysis. For example, another application for provenance that has received great interest is access control [5], [8], [17], [18], [19]. Through the introduction of LPM, a practical path to the deployment of such systems is now available. Figure 13 shows a possible design for provenance-based access control in Linux; provenance is collected in an LPM module and then flows into a policy engine that updates a MAC policy for the LSM module. The presence of LPM permits the advantages of modular composition for both the security and provenance modules, and allows the developers to make use of established security modules instead of implementing a new, untested one.

B. Future Work

Through the introduction of the Linux Provenance Module Framework, we have placed the ability to rapidly deploy provenance-aware systems in the hands of researchers and developers. LPM's architecture provenance hooks, MAC policy, and modules are already implemented as described in this paper. One exception is that we are currently developing the gateway for high-integrity volunteered provenance annotations. We intend to use the LPM platform to continue to explore the challenges and opportunities that whole-system provenance monitors represent, particularly further optimizations to provenance storage and application. Due to the growing interest in provenance-aware systems, we are also pursuing incorporating the LPM framework into Linux distributions and/or the mainline kernel.

VIII. Conclusion

The Linux Provenance Module Framework represents an exciting step forward for the provenance community, allowing for rapid prototyping of provenance-aware systems and greater collaboration between researchers. LPM is now fully developed, and will be released upon publication. LPM's release will coincide with the release of several provenance modules, including a re-implementation Hi-Fi as well as the innovative new Provenance Wall module. Through Provenance Walls, we have demonstrated that provenance can be leveraged to discover new attack surfaces in applications, and gain deeper insight into known attack surfaces. The Linux Provenance Module Framework significantly decreases the barriers to the development and adoption of provenance-aware systems.

IX. Acknowledgements

We thank Jun Li and Allen Malony for their valuable comments. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1118046, CNS-1254198, and CNS-1445983. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Lineage FS. http://crypto.stanford.edu/~cao/lineage.html.

[2] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972.

[3] R. S. Barga and L. A. Digiampietri. Automatic capture and efficient storage of e-Science experiment provenance. *Concurr. Comput. : Pract. Exper.*, 20:419–429, April 2008.

[4] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou. Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. In *NDSS Workshop on Security of Emerging Network Technologies*, SENT, Feb. 2014.

[5] A. Bates, B. Mood, M. Valafar, and K. Butler. 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, pages 277–284, New York, NY, USA, 2013. ACM.

[6] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. *USENIX Computing Systems*, 9(2):131–152, 1996.

[7] U. Braun, S. Garfinkel, D. A. Holland, K. Muniswamy-Reddy, and M. Seltzer. Issues in Automatic Provenance Collection. In *Proceedings of the 2006 International Provenance and Annotation Workshop*, Chicago, Illinois, May 2006.

[8] T. Cadenhead, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham. A Language for Provenance Access Control. In *CODASPY '11: Proceedings of the first ACM Conference on Data and Application Security and Privacy*, pages 133–144, San Antonio, TX, USA, 2011. ACM Press.

[9] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[10] R. Hasan, R. Sion, and M. Winslett. Introducing secure provenance: problems and challenges. In *Proceedings of the 2007 ACM workshop on Storage security and survivability (StorageSS '07)*, pages 13–18, New York, NY, USA, 2007. ACM.

[11] R. Hasan, R. Sion, and M. Winslett. SPROV 2.0: A Highly-Confgurable Platform-Independent Library for Secure Provenance. In *ACM CCS*, Nov. 2009.

[12] R. Hasan, R. Sion, and M. Winslett. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *FAST '09: Proceedings of the 7th USENIX Conference on File and Storage Technologies*, 2009.

[13] J. Inouye, R. Konuru, J. Walpole, and B. Sears. The effects of virtually addressed caches on virtual memory design and performance. *SIGOPS Oper. Syst. Rev.*, 26(4):14–29, Oct. 1992.

[14] P. McDaniel, K. Butler, S. McLaughlin, R. Sion, E. Zadok, and M. Winslett. Towards a Secure and Efficient System for End-to-End Provenance. In *TaPP '10: Proceedings of the 2nd USENIX Workshop on the Theory and Practice of Provenance*, 2010.

[15] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-Aware Storage Systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.

[16] S. Munroe, S. Miles, L. Moreau, and J. Vázquez-Salceda. PrIMe: A Software Engineering Methodology for Developing Provenance-aware Applications. In *Proceedings of the 6th International Workshop on Software Engineering and Middleware*, SEM, pages 39–46, New York, NY, USA, 2006. ACM.

[17] D. Nguyen, J. Park, and R. Sandhu. 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, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[18] Q. Ni, S. Xu, E. Bertino, R. Sandhu, and W. Han. An Access Control Language for a General Provenance Model. In *Secure Data Management*, Aug. 2009.

[19] J. Park, D. Nguyen, and R. Sandhu. A Provenance-Based Access Control Model. In *Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST)*, pages 137–144, 2012.

[20] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proceedings of the 2012 Annual Computer Security Applications Conference*, ACSAC '12, Orlando, FL, USA, 2012.

[21] A. C. Revkin. Hacked E-mail is New Fodder for Climate Dispute. *New York Times*, 20, 2009.

[22] M. Szomszor and L. Moreau. Recording and Reasoning over Data Provenance in Web and Grid Services. In R. Meersman, Z. Tari, and D. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 603–620. Springer Berlin Heidelberg, 2003.

[23] V. Tan, P. Groth, S. Miles, S. Jiang, S. Munroe, S. Tsasakou, and L. Moreau. Security issues in a soa-based provenance system. In L. Moreau and I. Foster, editors, *Provenance and Annotation of Data*, volume 4145 of *Lecture Notes in Computer Science*, pages 203–211. Springer Berlin Heidelberg, 2006.

[24] The Netfilter Core Team. The Netfilter Project: Packet Mangling for Linux 2.4. http://www.netfilter.org/, 1999.

[25] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. 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, pages 75–76, New York, NY, USA, 2012. ACM.

[26] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. Technical Report 2004-40, Stanford InfoLab, Aug. 2004.

[27] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Module Framework. In *Ottawa Linux Symposium*, page 604, 2002.

[28] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association.

[29] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, Y. Li, and D. D. E. Long. Evaluation of a Hybrid Approach for Efficient Provenance Storage. *Trans. Storage*, 9(4):14:1–14:29, Nov. 2013.

[30] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure Network Provenance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[31] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient Querying and Maintenance of Network Provenance at Internet-Scale. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2010.