Secure and Trustworthy Provenance Collection for Digital Forensics

Adam Bates, Devin J. Pohly, and Kevin R. B. Butler

Abstract Data provenance refers to the establishment of a chain of custody for information that can describe its generation and all subsequent modifications that have led to its current state. Such information can be invaluable for a forensics investigator. The first step to being able to make use of provenance for forensics purposes is to be able to ensure that it is collected in a secure and trustworthy fashion. However, the collection procsss along raises several significnt challenges. In this chapter, we discuss past approaches to provenance collection from application to operating system level, and promote the notion of a *provenance monitor* to assure the complete collection of data. We examine two instantiations of the provenance monitor concept through the Hi-Fi and Linux Provenance Module systems, discussing the details of their design and implementation to demonstrate the complexisty of collecting full provenance information. We consider the security of these schemes and raise challenges that future provenance systems must address to be maximually useful for practical forensic use.

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL, USA, e-mail: {bates,butler}@cise.ufl.edu

Department of Electrical Engineering and Computer Science
Pennsylvania State University, University Park, PA, USA, e-mail: djpohly@cse.psu.edu

1 Introduction

Successful forensics investigations rely on the trustworthiness of data that is retrieved. As a result, the ability to retrieve trustworthy logs and other information that explains where information was generated can be critical to successful inquiries. Such artifacts go part of the way, but not all of the way, towards answering the following question: how can we be assured that the chain of custody for data from the time it was originated until it arrived in its current state is both secure and trustworthy?

Data *provenance* provides a compelling means of providing answers to this challenging problem. The term *provenance* comes from the art world, where it refers to the ability to trace all activities related to an piece of art, in order to establish that is is genuine. An example of this usage is with Jan van Eyck's Arnolfini portrait, currently hanging in the National Gallery of London. The provenance of this celebrated portrait can be traced back almost 600 years to its completion in 1432, with metadata in the form of markings associated with its owners painted on the painting's protective shutters helping to establish the hands through which it has passed over the centuries [23].

More recently, data provenance has become a desired feature in the computing world. From its initial deployment in the database community [17] to a more recent focus to its proposed use as an operating sytems feature [43], data provenance provides a broad new capability for reasoning about the genesis and subsequent modification of data. In contrast to the current computing paradigm where interactions between system components are largely opaque, data provenance allows users to track, and understand how a piece of data came to exist in its current state. The realization of provenance-aware systems will fundamentally redefine how computing systems are secured and monitored, and will provide important new capabilities to the forensics community. Ensuring its efficacy in a computer system, though, is an extremely challenging problem, to the extent that the Department of Homeland Security has included provenance as one of its Hard Problems in Computing [14]. Ensuring that information is collected in a trustworthy fashion is the first problem that needs to be solved in order to assure the security of provenance. Without adequate protections in place, adversaries can target the collection mechanisms to destroy or tamper with provenance metadata, calling the trustworthiness of data into question or using it in a malicious fashion.

This book chapter focuses on how to ensure the secure and trustworthy collection of data provenance within computing systems. We will discuss past approaches to provenance collection and where and why those fall short, and discuss how taking a systems security approach to defining trustworthy provenanace collection can provide a system that fulfills the qualities necessary for a secure implementation. We will then discuss approaches from the research community that have attempted to ensure the fine-grained secure collection of provenance, as well as our own work in this area to provide a platform for deploying secure provenance collection as an operating system service.

2 Provenance-Aware Systems

Data provenance provides the ability to describe the history of a data object, including the conditions that led to its creation and the actions that delivered it to its present state. The potential applications for this kind of information are virtually limitless; provenance is of use in any scenario where a context-sensitive decision needs to be made about a piece of data. Specifically, provenance can be used to answer a variety of historical questions about the data it describes. Such questions include, but are not limited to, *"What processes and datasets were used to generate this data?"* and *"In what environment was the data produced?"* Conversely, provenance can also answer questions about the successors of a piece of data, such as *"What objects on the system were derived from this object?"*

A necessary prerequisite to the use of data provenance is its reliable capture and management, which is facilitated by provenance-aware software systems. Provenance capture mechanisms can be deployed at various layers of system operation, including applications, middleware, and operating systems. They can broadly be grouped into two categories: *disclosed* and *automatic*. In disclosed systems, provenance is recorded based on manual annotations created by the operator, while in automatic systems, software is instrumented to automatically generate and record lineage information.

2.1 Disclosed Provenance-Aware systems

The earliest efforts in provenance tracking arose from the scientific processing and database management communities. While the potential use cases for data provenance have broadened in scope over time, early investigators aims were to maintain virtual data descriptions that would allow them to explain data processing results and re-constitute those results in the event of their deletion. One of the earliest efforts in this space was Chimera [17], which provided a virtual data management system that allowed for tracking the derivations of data through computational procedures. Chimera is made up of a *virtual data catalog* that represents computation procedures used to derive data and a *virtual data language interpreter* for constructing and querying the catalog. It uses transformation procedures (i.e., processes) as its integral unit; its database is made up of *transformations* that represent executable programs and *derivations* that represent invocations of transformations. All other information (e.g., input files, output files, execution environment) are a subfield in the process' entry.

The Earth Science System Workbench, used for processing satellite imagery, also offered support for provenance annotations [18], as did the Collaboratory for the Multi-scale Chemical Sciences [46] and the Kepler system [41]. Specification-based approaches, which generated data provenance based on process documentation [40] also appeared in the literature at this time. Chimera and other early systems relied

on manual annotations or inferences from other metadata as sources for data provenance, and are therefore referred to as *disclosed* provenance-aware systems.

2.2 Automatic Provenance-Aware Systems

Automatic provenance-aware systems can be further divided into several categories based on software layer at which provenance collection occurs. We now consider past proposals for provenance at different operational layers, and identify the opportunities and challenges of each approach. We focus in this discusssion on applications operating system mechanisms to support provenance.

2.2.1 Automatic Provenance in Operating Systems

Capturing data provenance at the operating system layer offers a broad perspective into system activities, providing insight into all applications running on the host. Muniswamy-Reddy et al.'s Provenance-Aware Storage System (PASS) instruments the VFS layer of the Linux kernel to automatically collect, maintain, and provide search for data provenance [43]. PASS defines provenance as a description of the execution history that produced a persistent object (file). Provenance records are attribute/value pairs that are referenced by a unique pnode number. PASS provenance is facilitate a variety of useful tasks, including script generation and document reproduction, detecting system changes, intrusion detection, retrieving compile-time flags, build debugging, and understanding system dependencies. One limitation of the PASS system is that the model for provenance collection was fixed, and did not provide a means of extending the system with additional provenance attributes or alternate storage models.

Gehani and Tariq present SPADE in response to requests for coarser-grained information and the the ability to experiment with different provenance attributes, novel storage and indexing models, and handling provenance from diverse sources [21]. SPADE is a java-based daemon that offers provenance *reporter* modules for Windows, Linux, OSX, and Android. The reporters are based on a variety of methods of inference, including polling of basic user space utilities (e.g., *ps* for process info, *lsof* for network info), audit log systems (e.g., Window's ETW, OSX's BSM), and interposition via user space filesystems like FUSE. Due to its modular design, SPADE can be easily extended to support additional provenance streams.

Both the PASS and SPADE systems facilitate provenance collection through ad hoc instrumentation or polling efforts, making it difficult to provide any assurance of the *completeness* of the provenance that they collect. In fact, there are several examples of how these systems fail to provide adequate tracking for explicit data flows through a system. As SPADE records provenance in part through periodic polling of system utilities, there exists the potential for race conditions in which short-lived processes or messages could be successfully created between polling

intervals. By observing the VFS layer, PASS provides support for non-persistent data, such as network sockets which are represented by a system file; however, it fails to track a variety of forms of interprocess communication, such as signals or shared memory, which provides a covert channel for communicating applications.

2.2.2 Automatic Provenance in Middleware

The earliest disclosed provenance-aware systems were proposed for middleware, such as Chimera [17] and the Earth Science System Workbench [18]. Today, automatic provenance-aware middleware continues to be one of the most widespread and impactful forms of provenance capability. Tools such as VisTrails [54], which tracks the provenance of scientific visualizations, have established themselves as viable platforms in scientific computing comminities. By instrumenting a common computing platform within a community, such as a database management system or scientific computing engine, provenance-aware middleware provides easy access to semantically rich, domain specific provenance metadata.

Propagating lineage information as datasets are fused and transformed is one of the strongest motivations for provenance capabilities. Chiticariu et al.'s DBNotes provides a "post-It note" system for relational data in which annotations are propagated based on lineage [12]. DBNotes provides a SQL extension (pSQL) that allows one to specify how provenance annotations should propagate through a SQL query. pSQL also allows annotations to be queried inline with other relational data. DBNotes also has a visualization feature that demonstrates the *journey* taken by a piece of data through various databases and transformation steps. In related work, Holland et al. [26, 42] present PQL, another query model for data provenance provides a semistructured data and query model for the graph-centric nature of provenance. By extending the Lowel query language to support bidirectional edge traversal and more expressive attributes, PQL compares favorably due to alternate models for provenance querying as relational languages cannot efficiently encode graphs, nor can tree-based structures like XML.

While both of these systems require users to adopt a new query language, Glavic and Alonso [22] present PERM (Provenance Extension of the Relational Model), a system that uses query rewriting to annotate result tuples with provenance information, permitting provenance to be queried, stored, and optimized using standard relational database techniques without impacting normal database operations. Given a query q, Perm creates q^+ that produces the same result as q but extended with additional attributes. Through using standard relational models, PERM offers support for more sophisticated queries than other provenance-aware databases, and outperforms the Trio system by at least a factor of 30.

2.2.3 Automatic Provenance in Applications

Manual instrumentation of applications offers concise and semantically rich data provenance. Several development libraries have appeared in the literature to aid in instrumentation efforts, providing a standardized API through which to emit data provenance. Muniswamy-Reddy et al.'s *Disclosed Provenance API (DPAPI)* [42] and Macko and Seltzer's *Core Provenance Library (CPL)* [37] provide portable, multi-lingual libraries that application programmers could use to disclose provenance by defining provenance objects and describing the flows between those objects. CPL offers the advantages of avoiding version disconnect between files that are seemingly distinct to the operating system but are actually ancestors, integration between different provenance-aware applications due to a look-up function in the API, and reconciling different notions of provenance in a unified format.

DPAPI is a component of the PASSv2 project [42]. It's intended purpose is to create provenance-aware applications whose provenance can be layered on top of information collected by the PASS system, allowing system operators to reason holistically about activities at multiple system layers. Several exemplar applications for provenance layering as part of this effort: Kepler, Lynx, and a set of general-purpose Python wrappers. Similarly, the SPADE system offers support for provenance layering by exposing a named pipe and generic domain-specific language for application layer provenance disclosure [21].

Other work has sought out alternate deployment model to create provenance-aware applications at a lower cost and without developer cooperation. Hassan et al. present Sprov, a modified version of the `stdio` library that captures provenance for file I/O system calls at the application layer. By replacing the `glibc` library with the modified version, Sprov is able to record file provenance for all dynamically linked applications on the system. This system also provides integrity for provenance records through the introduction of a tamper-evident *provenance chain* primitive. By cryptographically binding time-ordered sequences of provenance records together for a given document, Sprov is able to prevent undetected rewrites of the document's history.

Recent efforts have also attempted to reconstitute application workflow provenance through analysis of system layer audit logs, requiring only minimally invasive and automated transformation of the monitored application. A significant consequence of provenance tracking at the operating system layer is *dependency explosion* – for long-lived processes, each new output from an application must conservatively be assumed to have been derived from all prior inputs, creating *false provenance*. The BEEP systems resolves this problem through analysis and transformation of binary executables [32]. Leveraging the insight that most long-lived processes are made up of an initialization phase, main work loop, and tear-down phase, BEEP procedurally identifies the main work loop in order to decompose the process into autonomous units of work. After this *execution partitioning (EP)* step, the system audit log can then be analyzed to build causal provenance graphs for the monitored applications. Ma et al. go on to adapt these techniques Windows and other proprietary software [35], where EP can be performed by perform regular ex-

pression analysis of audit logs in order to identify autonomous units of work. The LogGC system extends BEEP by introducing a garbage collection filtering mechanism to improve the forensic clarity of the causal graphs [33]; for example, if a process creates and makes use of a short-lived temporary file that no other process ever reads, this node contains no semantic value in the causal graph, and can therefore be pruned. These techniques should be able to be applied in tandem with Chapman et al's provenance factorization techniques that find common subtrees and manipulate them to reduce the provenance size [11]. Although these systems do not modify the operating system, by operating at the system call or audit logs levels, Sprov, LogGC, and BEEP provide provenance at a similar granularity to that of provenance-aware operating systems; they offer only limited insight into application layer semantics.

3 Ensuring the Trustworthiness of Provenance

Data provenance has proven to be of tremendous value in addressing a variety of security challenges. Provenance is most commonly employed as a forensic tool for performing offline event attribution. Pohly et al. [48], Ma et al. [36], and Lee et al. [32, 33] all demonstrate that their provenance-aware systems can be used to diagnose system intrusions such as malware and data exfiltration. Jones et al. [28] propose a technique to use provenance to aid in determining potential sources of information leaks. When a removable storage device or network connection is opened by a provenance-aware host, a "ghost object" is created representing the device. Files which are read by the user during the session are added as inputs to the ghost object. The ghost object is annotated with information to make it later identifiable, including device ID, user ID, process ID, and remote network information. If a leak occurs, the ghost objects (i.e., "transient provenance") can be used to limit the list of potential culprits.

In other work, data provenance has been shown to be a promising means of enforcing and verifying the realtime security of computer systems. Provenance-based access control schemes (PBAC) have been presented that leverage richer contextual information than traditional MAC labels [44, 45, 47]. By inspecting the ancestry of a data object, it is possibly to dynamically infer its present security context before applying an access control rule. A provenance-based approach provides a general method for handling arbitrary data fusions, obviating the need to exhaustively enumerate transitions between security contexts. Provenance has similarly been considered in mechanisms for facilitating regulatory compliance. Aldeco-Pérez and Moreau present a methodology for incorporating provenance into applications during their design such that they satisfy the auditing requirements of the UK Data Protection Act [1]. Bates et al. consider the challenges of managing data provenance between cloud deployments [4], and present a general distributed mechanism for the enforcement of regulatory policies such as ITAR [60] and HIPAA [10].

3.1 Security Challenges to Provenance Collection

Unfortunately, while the above work has shown that provenance is an invaluable capability when securing systems, less attention has been given to *securing* provenance-aware systems. Provenance itself is a ripe attack vector; adversaries may seek to tamper with provenance to hide evidence of their misdeeds, or to subvert another system component that uses provenance as an input. In light of this realization, it becomes clear that the integrity, authenticity, and completeness of provenance must be guaranteed before it can be put to use.

When provenance collection occurs in user-space, either through software or middleware, an implicit decision is made to fully trust user space applications. However, in a malicious environment, there is likely to be exploitable software bugs in provenance-aware applications that handle untrusted inputs. An attacker in control

of a compromised application could instruct it to disclose false provenance about its activities, or simply disable the mechanism responsible for provenance collection altogether. Examples of such vulnerabilities abound in the systems surveyed above. An attacker with root privilege could terminate the SPADE collection daemon [21] or modify environment variables to prevent Sprov from being linked [24]. In spite of the fact that BEEP and LogGC are intended to aid in Advanced Persistent Threat (APT) detection, a compromised application could violate BEEP's EP assumptions in order to cast doubt onto other system users [32, 33]. Due to the confinement problem that persists in commodity operating systems [31], extraordinary lengths would need to be taken to harden these systems from attack. As a result, the provenance collected by these mechanisms is only suitable for benign operating environments.

Capturing data provenance within the operating system is a more promising direction for secure data provenance under a realistic threat model. While PASS was designed for benign environments [8], its approach of instrumenting the kernel for provenance collection creates an opportunity to insulate the provenance capture agent from the dangers of user space. Hardening the kernel against attack, while a difficult problem, can be achieved through use of Mandatory Access Control (MAC) mechanisms and other trusted computing techniques. Unfortunately, as PASS instruments the VFS layer [43], it is unable to monitor a variety of explicit data flows within the system, such as shared memory and other forms of interprocess communication. Moreover, PASSv2, which supports layered provenance, does not have a defense against the untrustworthy provenance-aware applications discussed above. The efficient ProTracer system suffers from many of the same problems. While it records provenance outside of the file system, its instrumentation of the kernel is ad hoc and unverified, making it possible that explicit data flows are left unmonitored [36]. ProTracer also places trust in user space applications during a training phase that provides EP; like BEEP, this assumption could be violated by an active attacker, injecting uncertainty into the provenance record.

In response to these issues, and to facilitate data provenance's use in other security-critical tasks, concurrent work in 2010 began to advocate for the union of data provenance and trusted computing. Lyle and Martin [34] argue that trusted computing is useful and immediately applicable to the provenance domain. They sketch a service-oriented architecture that uses Trusted Platform Module (TPM) attestations to track the provenance of jobs in a smart grid. When a node receives a job, it hashes the job into one of the TPM's Platform Configuration Registers (PCRs) and extends the PCR with the hased result upon finishing the job. It then sends a full attestation up to and including the job result to a provenance store. This allows the grid to later produce a non-repudiable assertion of the software and hardware stack that produced the result. However, Lyle and Martin's proposal did not describe a complete provenance-aware operating system, as provenance proofs did not describe configuration files, environment variables, generated code, and load information, nor can this system explain *who* accessed a piece of data.

3.2 The Provenance Monitor Concept

To address the challenges discussed above, McDaniel et al. [38] developed the concept of a *provenance monitor*, where provenance authorities accept host-level provenance data from validated monitors to assemble a trustworthy provenance record. Subsequent users of the data obtain a provenance record that identifies not only the inputs, systems, and applications leading to a data item, but also evidence of the identity and validity of the recording instruments that observed its evolution. At the host level, the provenance monitor acts as the recording instrument that observes the operation of a system and securely records each data manipulation. The concept for a provenance monitor is based on the *reference monitor* proposed by Anderson (cite), which has become a cornerstone for evaluating systems security. The two concepts share the following three fundametal properties:

The host level provenance monitor should enforce the classic reference monitor guarantees of complete mediation of relevant operations, tamper-proofness of the monitor itself, and basic verification of correct operation. For the purpose of the provenance monitor, we define these as follows.

- **Complete Mediation.** A provenance monitor should *mediate* all provenance-relevant operations, whatever these may be for a given application. In other words, there should be no way by which the provenance monitor can be bypassed if an event is provenance-sensitive.
- **Tamperproof.** The provenance monitor must be isolated from the subjects operating on provenance-enhanced data, e.g. the OS kernel or storage device, and there should be no means by which the monitor can be modified or disabled by the activities of users on a system.
- bf Verifiable. Finally, the provenance monitor should be designed to allow for simple verification of its behavior, and optimally should be subject to formal verification to assure its trustworthy operation.

The provenance monitor provides powerful guarantees for the secure and trustworthy collection of provenance. However, while the idea is seeminly simple in concept, its execution requires considerable design and implementation considerations. As we have seen above, a large amount of provenance related proposals, while pushing forth novel functionality and advancing the state of research, do not pass the provenance monitor criteria. Complete mediation and tamperproofness cannot be guaranteed if the mechanisms used to collect provenance are subject to compromise, and collecting sufficiently fine-grained provenance to ensure complete mediation is a challenge unaddressed by other systems discussed. The next two sections discuss recent attempts to satisfy the provenance monitor concept and detail the challenges and design decisions made to assure a practical and functional collection system.

4 High-Fidelity Whole Systems Provenance

As we described in the previous section, for a data provenance system to provide the holistic view of system operation required for such forensic applications, it must be complete and faithful to actual events. This property, which we call "fidelity," is necessary for drawing valid conclusions about system security. A missing entry in the provenance record could sever an important information flow, while a spurious entry could falsely implicate an innocuous process. The provenance monitor concept proides a strong conceptual framework for achieving trusttworthy proenance collection. In particular, this mechanism must provide complete mediation for events which should appear in the record.

Forensic investigation requires a definition of provenance which is broader than just file metadata. What is needed is a record of *whole-system provenance* which retains actions of processes, IPC mechanisms, and even the kernel. These "transient" system objects can be meaningful even without being an ancestor of any "persistent" object. The command-and-control daemon on Alice's server, for example, was significant because it was a *descendant* of the compromised process. If the provenance system had deemed it unworthy of inclusion in the record, she could not have traced the outgoing connections to the compromise.

To address these issues, Pohly et al. developed the Hi-Fi provenance system, designed to collect high-fidelity whole-system provenance. Hi-Fi was the first provenance system to collect a *complete* provenance record from early kernel initialization through system shutdown. Unlike existing provenance systems, it accounts for all kernel actions as well as application actions. Hi-Fi also collects *socket provenance*, creating a system-level provenance record that spans multiple hosts.

4.1 Design of Hi-Fi

Hi-Fi consists of three components: the provenance collector, the provenance log, and the provenance handler. An important difference between Hi-Fi and previous work is that rather than collecting events at the file system level, Hi-Fi ensures complete mediation by collecting events as a Linux Security Module (LSM) (cite). Because the collector is an LSM, it resides below the application layer in the operating system's kernel space, and is notified whenever a kernel object access is about to take place. When invoked, the collector constructs an entry describing the action and writes it to the provenance log. The log is a buffer which presents these entries to userspace as a file. The provenance handler can then access this file using the standard file API, process it, and store the provenance record. The handler used in our experiments simply copies the log data to a file on disk, but it is possible to implement a custom handler for any purpose, such as post-processing, graphical analysis, or storage on a remote host.

Such a construction allows for a far more robust adversarial model. Hi-Fi maintains the fidelity of provenance collection regardless of any compromise of the OS

user space by an adversary. This is a strictly stronger guarantee than those provided by any previous system-level provenance collection system. Compromises are possible against the kernel, but other techniques for protecting kernel integrity, including disk-level versioning [57] or a strong write-once read-many (WORM) storage system [55], can mitigate the effects of such compromises. Because provenance never changes after being written, a storage system with strong WORM guarantees is particularly well-suited to this task. For socket provenance, Hi-Fi guarantees that incoming data will be recorded accurately; to prevent on-the-wire tampering by an adversary, standard end-to-end protection such as IPsec should be used.

The responsibility of the provenance handler is to interpret, process, and store the provenance data after it is collected, and it should be flexible enough to support different needs. Hi-Fi decouples provenance handling from th collection process, allowing the handler to be implemented according to the system's needs.

For the purposes of recording provenance, each object which can appear in the log must be assigned an identifier which is unique for the lifetime of that object. Some objects, such as inodes, are already assigned a suitable identifier by the kernel. Others, such as sockets, require special treatment. For the rest, Hi-Fi generates a *provid*, a small integer which is reserved for the object until it is destroyed. These provids are managed in the same way as process identifiers to ensure that two objects cannot simultaneously have the same provid.

4.2 Handling of System-Level Objects

Collecting system-level provenance requires a clear model of system-level objects. For each object, Hi-Fi must first describe how data flows into, out of, and through it. Next, the LSM hooks for mediating data-manipulating objects must be identified (as listed in Table 1), or new hooks are placed if existing ones are insufficient.

Each entry in the provenance log describes a single action on a kernel object. This includes the type of action, the subject, the object, and any appropriate context.

The data-flow model includes transferring data between multiple systems or multiple boots of a system. Hi-fi must therefore identify each boot separately. To ensure that these identifiers do not collide, a random UUID is created at boot time, which is written to the provenance log so that subsequent events can be associated with the system on which they occur.

Within a Linux system, the only actors are processes (including threads), and the kernel. These actors store and manipulate data in their respective address spaces, and we treat them as black boxes for the purpose of provenance collection. Most data flows between processes use one of the objects described in subsequent sections. However, several actions are specific to processes: forking, program execution, and changing subjective credentials.

Since LSM is designed to include kernel actions, it does not represent actors using a PID or `task_struct` structure. Instead, LSM hooks receive a `cred` structure, which holds the user and group credentials associated with a process or kernel

Kernel object	LSM hook
Inode	`inode_init_security`
	`inode_free_security`
	`inode_link`
	`inode_unlink`
	`inode_rename`
	`inode_setattr`
	`inode_readlink`
	`inode_permission`
Open file	`file_mmap`
	`file_permission`
Program	`bprm_check_security`
	`bprm_committing_creds`
Credential	`cred_prepare`
	`cred_free`
	`cred_transfer`
	`task_fix_setuid`
Socket	`socket_sendmsg`
	`socket_post_recvmsg`
	`socket_sock_rcv_skb`
	`socket_dgram_append`
	`socket_dgram_post_recv`
	`unix_may_send`
Message queue	`msg_queue_msgsnd`
	`msg_queue_msgrcv`
Shared memory	`shm_shmat`

Table 1 LSM hooks used to collect provenance in Hi-Fi.

action. Whenever a process is forked or new credentials are applied, a new credential structure is created, allowing us to use these structures to represent individual system actors. As there is no identifier associated with these `cred` structures, we generate a provid to identify them.

Regular files are the simplest and most common means of storing data and sharing it between processes. Data enters a file when a process writes to it, and a copy of this data leaves the file when a process reads from it. Both reads and writes are mediated by a single LSM hook, which identifies the the actor, the open file descriptor, and whether the action is a read or a write. Logging file operations is then straightforward.

Choosing identifiers for files, however, requires considering that files differ from other system objects in that they are persistent, not only across reboots of a single system, but also across systems (like a file on a portable USB drive). Because of this, it must be possible to uniquely identify a file independent of any running system. In this case, already-existing identifiers can be used rather than generating new ones. Each file has an inode number which is unique within its filesystem, which can be combined with a UUID that identifies the filesystem itself to obtain a suitable identifier that will not change for the lifetime of the file. UUIDs are generated for most filesystems at creation.

Files can also be mapped into one or more processes' address spaces, where they are used directly through memory accesses. This differs significantly from normal reading and writing in that the kernel does not mediate accesses once the mapping is established. Hi-Fi only records the mapping when it occurs, along with the requested access mode (read, write, or both). This does not affect the notion of complete mediation if it is assumed that flows via memory-mapped files take place whenever possible.

Shared memory segments are managed and interpreted in the same way. POSIX shared memory is implemented using memory mapping, so it behaves as described above. XSI shared memory, though managed using different system calls and mediated by a different LSM hook, also behaves the same way, so our model treats them identically. In fact, since shared memory segments are implemented as files in a temporary filesystem, their identifiers can be chosen in the same way as file identifiers.

The remaining objects have stream or message semantics, and they are accessed sequentially. In these objects, data is stored in a queue by the writer and retrieved by the reader. The simplest such object is the pipe, or FIFO. Pipes have stream semantics and, like files, they are accessed using the `read` and `write` system calls. Since a pipe can have multiple writers or readers, it cannot be directly represented as a flow from one process to another. Instead, flow is split into two parts, modeling the data queue as an independent file-like object. In this way, a pipe behaves like a sequentially-accessed regular file. In fact, since named pipes are inodes within a regular filesystem, and unnamed pipes are inodes in the kernel's "pipefs" pseudo-filesystem, pipe identifiers can be chosed similarly to files.

Message queues are similar to pipes, with two major semantic differences: the data is organized into discrete messages instead of a single stream, and these messages can be delivered in a different order than that in which they are sent. However, because LSM handles messages individually, a unique identifier can be created for each, allowing reliable idenfitication of which process receives the message regardless of the order in which the messages are dequeued. Since individual messages have no natural identifier, a provid is generated for each.

Sockets are the most complex form of inter-process communication handled by Hi-Fi but can be modeled very simply. As with pipes, a socket's receive queue can be represented an intermediary file between the sender and receiver. Sending data merely requires writing to this queue, and receiving data is reading from it. The details of network transfer are hidden by the socket abstraction. Stream sockets provide the simplest semantics with respect to data flow: they behave identically to pipes. Since stream sockets are necessarily connection-mode, all of the data sent over a stream socket will arrive in the same receive queue. Message-oriented sockets, on the other hand, do not necessarily have the same guarantees. They may be connection-mode or connectionless, reliable or unreliable, ordered or unordered. Each packet therefore needs a separate identifier, since it is unclear at which endpoint the message will arrive.

Socket identifiers must be chosen carefully. An identifier must never be re-used since since a datagram can have an arbitrarily long lifetime. The identifier should

also be associated with the originating host. Associating messages with a per-boot UUID addresses these requirements. By combining this UUID with an atomic counter, a sufficiently large number of identifiers can be generated.

4.3 Hi-Fi Implementation

Provenance Logging. As noted in Section 2, provenance collection has been noted to generate a large volume of data. Because of this, an efficient and reliable mechanism for making large quantities of kernel data available to user space is necessary. Other systems have accomplished this by using an expanded `printk` buffer [52], writing directly to on-disk log files [42], or using FUSE [56]. Each of these approaches has drawbacks, so Hi-Fi instead uses a Linux kernel object known as a *relay,* which is designed specifically to address this problem [63].

A relay is a kernel ring buffer made up of a set of preallocated sub-buffers. Once the relay has been initialized, the collector writes provenance data to it using the `relay_write` function. This data will appear in userspace as a regular file, which can be read by the provenance handler. Since the relay is backed by a buffer, it retains provenance data even when the handler is not running, as is the case during boot, or if the handler crashes and must be restarted. Since the number and size of the sub-buffers in the relay are specified when it is created, the relay has a fixed size. Although the collector can act accordingly if it is about to overwrite provenance which has not yet been processed by the handler, a better solution is allowing the relay's size parameters to be specified at boot time.

Early Boot Provenance. The Linux kernel's boot-time initialization process consists of setting up a number of subsystems in sequence. One of these subsystems is the VFS subsystem, which is responsible for managing filesystem operations and the kernel's in-memory filesystem caches. These caches are allocated as a part of VFS initialization. They are then used to cache filesystem information from disk, as well as to implement memory-backed "pseudo-filesystems" such as those used for pipes, anonymous memory mappings, temporary files, and relays.

The security subsystem, which loads and registers an LSM, is another part of this start-up sequence. This subsystem is initialized as early as possible, so that boot events are also subject to LSM mediation. In fact, the LSM is initialized *before* the VFS, which has a peculiar consequence for the relay we use to implement the provenance log. Since filesystem caches have not yet been allocated, the relay cannot be created when the LSM is initialized, which violated Hi-Fi's goal of fidelity. In response, Hi-Fi separates relay creation from the rest of the module's initialization and registers it as a callback in the kernel's generic `initcall` system. This allows it to be delayed until after the core subsystems such as VFS have been initialized. In the meantime, provenance data is stored in a small temporary buffer. Inspection of this early boot provenance reveals that a one-kilobyte buffer is sufficiently large to hold the provenance generated by the kernel during this period. Once the relay is created, temporary boot-provenance buffer is flushed of its contents and freed.

OS Integration. One important aspect of Hi-Fi's design is that the provenance handler must be kept running to consume provenance data as it is written to the log. Since the relay is backed by a buffer, it can retain a certain amount of data if the handler is inactive or happens to crash. It is important, though, that the handler is restarted in this case. Fortunately, this is a feature provided by the operating system's `init` process. By editing the configuration in `/etc/inittab`, we can specify that the handler should be started automatically at boot, as well as respawned if it should ever crash.

Provenance must also be collected and retained for as much of the operating system's shutdown process as possible. At shutdown time, the `init` process takes control of the system and executes a series of actions from a shutdown script. This script asks processes to terminate, forcefully terminates those which do not exit gracefully, unmounts filesystems, and eventually powers the system off. Since the provenance handler is a regular user space process, it is subject to this shutdown procedure as well. However, there is no particular order in which processes are terminated during the shutdown sequence, so it is possible that another process may outlive the handler and perform actions which generate provenance data.

In response, Hi-Fi handles the shutdown process similarly to a system crash. The provenance handler must be restarted, and this is accomplished by modifying the shutdown script to re-execute the handler after all other processes have been terminated before filesystems are unmounted. This special case requires a "one-shot" mode in the handler which, instead of forking to the background, exits after handling the data currently in the log. This allows it to handle any remaining shutdown provenance, then returns control to `init` to complete the shutdown process.

Bootstrapping Filesystem Provenance. Intuitively, a complete provenance record contains enough information to recreate the structure of an entire filesystem. This requires three things: a list of inodes, filesystem metadata for each inode, and a list of hard links (filenames) for each inode. Hi-Fi includes a hook corresponding to each of these items, to ensure all information appears in the provenance record starting from an empty filesystem. However, this is difficult to do in practice, as items may have been used elsewhere or provenance may be collected on an existing, populated filesystem. Furthermore, it is actually impossible to start with an empty filesystem. Without a root inode, which is created by the corresponding `mkfs` program, a filesystem cannot even be mounted. Unfortunately, `mkfs` does this by writing directly to a block device file, which does not generate the expected provenance data.

Therefore, provenance must be bootstrapped on a populated filesystem. To have a complete record for each file, a creation event for any pre-existing inodes must be generated. Hi-Fi implements a utility called `pbang` (for "provenance Big Bang") which does this by traversing the filesystem tree. For each new inode it encounters, it outputs an allocation entry for the inode, a metadata entry containing its attributes, and a link entry containing its filename and directory. For previously encountered inodes, it only outputs a new link entry. All of these entries are written to a file to complete the provenance record. A new filesystem is normally created using `mkfs`, then made provenance-aware by executing `pbang` immediately afterward.

Opaque Provenance. Early versions of Hi-Fi generated continuous streams of provenance even when no data was to be collected. Inspection of the provenance record showed that this data described the actions of the provenance handler itself. The handler would call the `read` function to retrieve data from the provenance log, which then triggered the `file_permission` LSM hook. The collector would record this action in the log, where the handler would again read it, triggering `file_permission`, and so on, creating a large amount of "feedback" in the provenance record. While technically correct behavior, this floods the provenance record with data which does not provide any additional insight into the system's operation. One option for solving this problem is to make the handler completely exempt from provenance collection. However, this could interfere with filesystem reconstruction. Instead, the handler is *provenance-opaque*, treated as a black box which only generates provenance data if it makes any significant changes to the filesystem.

To achieve this Hi-Fi informs the LSM which process is the provenance handler, by leveraging the LSM framework's integration with extended filesystem attributes. The provenance handler program is identified by setting an attribute called `security.hifi`. The "security" attribute namespace, which is reserved for attributes used by security modules, is protected from tampering by malicious users. When the program is executed, the `bprm_check_security` hook examines this property for the value "opaque" and sets a flag in the process's credentials indicating that it should be treated accordingly. In order to allow the handler to create new processes without reintroducing the original problem—for instance, if the handler is a shell script—this flag is propagated to any new credentials that the process creates.

Socket Provenance. Network socket behavior is designed to be both transparent and incrementally deployable. To allow interoperability with existing non-provenanced hosts, packet identifiers are placed in the IP Options header field. Two Netfilter hooks process packets at the network layer. The outgoing hook labels each packet with the correct identifier just before it encounters a routing decision, and the incoming hook reads this label just after the receiver decides the packet should be handled locally. Note that even packets sent to the loopback address will encounter both of these hooks.

In designing the log entries for socket provenance, Hi-Fi aims to make the reconstruction of information flows from multiple system logs as simple as possible. When the sender and receiver are on the same host, these entries should behave the same as reads and writes. When they are on different hosts, the only added requirement should be a partial ordering placing each send before all of its corresponding receives. Lamport clocks [30] would satisfy this requirement. However, the `socket_recvmsg` hook, which was designed for access control, executes before a process attempts to receive a message. This may occur before the corresponding `socket_sendmsg` hook is executed. To solve this, a `socket_post_recvmsg` hook is placed after the message arrives and before it is returned to the receiver; this hook generates the entry for receiving a message.

Support for TCP and UDP sockets is necessary to demonstrate provenance for both connection-mode and connectionless sockets, as well as both stream and

message-oriented sockets. Support for the other protocols and pseudo-protocols in the Linux IP stack, such as SCTP, ping, and raw sockets, can be implemented using similar techniques. For example, SCTP is a sequential packet protocol, which has connection-mode and message semantics.

TCP Sockets. TCP and other connection-mode sockets are complicated in that a connection involves three different sockets: the client socket, the listening server socket, and the server socket for an accepted connection. The first two are created in the same way as any other socket on the system: using the `socket` function, which calls the `socket_create` and `socket_post_create` LSM hooks. However, sockets for an accepted connection on the server side are created by a different sequence of events. When a listening socket receives a connection request, it creates a "mini-socket" instead of a full socket to handle the request. If the client completes the handshake, a new child socket is cloned from the listening socket, and the relevant information from the mini-socket (including our IP options) is copied into the child. In terms of LSM hooks, the `inet_conn_request` hook is called when a mini-socket is created, and the `inet_csk_clone` hook is called when it is converted into a full socket. On the client side, the `inet_conn_established` hook is called when the SYN+ACK packet is received from the server.

Hi-Fi must treat the TCP handshake with care, since there are two different sockets participating on the server side. A unique identifier is created for the mini-socket in the `inet_conn_request` hook, and this identifier is later copied directly into the child socket. The client must then be certain to remember the correct identifier, namely, the one associated with the child socket. The first packet that the client receives (the SYN+ACK) will carry the IP options from the listening parent socket. To keep this from overriding the child socket, the `inet_conn_established` hook clears the saved identifier so that it is later replaced by the correct one.

UDP Sockets. Since UDP sockets are connectionless, we an LSM hook must assign a different identifier to each datagram. In addition, this hook must run in process context to record the identifier of the process which is sending or receiving. The only existing LSM socket hook with datagram granularity is the `sock_rcv_skb` hook, but it is run as part of an interrupt when a datagram arrives, not in process context. The remaining LSM hooks are placed with socket granularity; therefore, two additional hooks are placed to mediate datagram communication. If the file descriptor of the receiving socket is shared between processes, they can all receive the same datagram by using the `MSG_PEEK` flag. In fact, multiple processes can also contribute data when *sending* a single datagram by using the `MSG_MORE` flag or the `UDP_CORK` socket option. Because of this, placing send and receive hooks for UDP is a very subtle task.

Since each datagram is considered to be an independent entity, the crucial points to mediate are the addition of data to the datagram and the reading of data from it. The Linux IP implementation includes a function which is called from process context to append data to an outgoing socket buffer. This function is called each time a process adds data to a corked datagram, as well as in the normal case where a single process constructs a datagram and immediately sends it. This makes it an ideal candidate for the placement of the send hook, which we call

`socket_dgram_append`. Since this hook is placed in network-layer code, it can be applied to any message-oriented protocol and not just UDP.

The receive hook is placed in protocol-agnostic code, for similar flexibility. The core networking code provides a function which retrieves the next datagram from a socket's receive queue. UDP and other message-oriented protocols use this function when receiving, and it is called once for each process that receives a given datagram. This is an ideal location for the message-oriented receive hook, so the `socket_dgram_post_recv` hook is placed in this function.

4.4 Limitations of Hi-Fi

Hi-Fi represents a significant step forward in provenance collection, being the first system to consider design with regard to the provenance monitor concept. The complexity of design and implmentation attest to the goals of complete mediation of provenance. However, it fails to address other security challengs idenfified in this chapter.

Hi-Fi does not completely satisfy the provenance monitor concept; enabling Hi-Fi blocks the installation of other LSM's, such as SELinux or Tomoyo, effectively preventing the installation of a mandatory access control (MAC) policy that could otherwise be used to protect the kernel. This leaves the entire system, including Hi-Fi's trusted computing base, vulnerable to attack, and Hi-Fi is therefore not tamperproof. Hi-Fi is also vulnerable to network attacks. Hi-Fi embeds an identifier into each IP packet transmitted by the host, which the recipient host to later use the identify to query the sender for the provenance of the packet. However, because these identifiers are not cryptographically secured, an attacker in the network can strip the provenance identifiers off of packets in transit, violating the forensic validity of Hi-Fi's provenance in distributed environments. Finally, Hi-Fi does not provide support for provenance-aware applications. Provenance layering is vital to obtaining a comprehensive view of system activity; however, rather than providing an insecure disclosure mechanism like PASSv2 [42], Hi-Fi does not offer layering support at all, meaning that its provenance is not complete in its observations of relevant operations.

5 Linux Provenance Modules

As we have shown in this chapter, the application of data 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 [52, 43], operating systems [48], access controls [44, 47], and distributed systems [4, 65, 67]. 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. Moreover, each of these proposals has conceptualized provenance in different ways, indicating that a one-size-fits-all solution to provenance collection is unlikely to meet the needs of all of these audiences

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 [43, 48]. Unfortunately, this introduces further security and interoperability problems; in order to enable provenance-aware systems, users currently need to disable their MAC policy [48], instrument applications [24, 65], gamble on experimental storage formats [43], or sacrifice other critical system functionality. These issues point to a pressing need for a dedicated platform for provenance development.

The **Linux Provenance Modules (LPM)** project [5] is an attempt to unify the operational needs of the disparate provenance communities through the design and implementation of a generalized framework for the development of automated, whole-system provenance collection on the Linux operating system. LPM extends and generalizes the Hi-Fi approach to kernel layer provenance collection with consideration for a variety of automated provenance systems that have been proposed in the literature. The framework is designed in such a way to allow for experimentation with new provenance collection mechanisms, and permits interoperability with other security mechanisms.

5.1 Augmenting Whole-System Provenance

The LPM project provides an explicit definition for the term *whole-system provenance* introduced in the Hi-Fi work. that is broad enough to accommodate the needs of a variety of existing provenance projects. To arrive at a definition, four past proposals were inspected that collect broadly scoped provenance: SPADE [21], LineageFS [52], PASS [43], and Hi-Fi [48]. **SPADE** provenance is structured around primitive operations of system activities with data inputs and outputs. It instruments file and process system calls, and associates each call to a process ID (PID), user identifier, and network address. **LineageFS** uses a similar definition, associating process IDs with the file descriptors that the process reads and writes. **PASS** associates a process's output with references to all input files and the command line and

process environment of the process; it also appends out-of-band knowledge such as OS and hardware descriptions, and random number generator seeds, if provided. In each of these systems, networking and IPC activity is primarily reflected in the provenance record through manipulation of the underlying file descriptors. **Hi-Fi** takes an even broader approach to provenance, treating non-persistent objects such as memory, IPC, and network packets as principal objects.

In all instances, provenance-aware systems are exclusively concerned with operations on *controlled data types*, which are identified by Zhang et al. as files, inodes, superblocks, socket buffers, IPC messages, IPC message queue, semaphores, and shared memory [64]. Because controlled data types represent a superset of the objects tracked by system layer provenance mechanisms, LPM defines whole-system provenance as *a complete description of agents (users, groups) controlling activities (processes) interacting with controlled data types during system execution.*

LPM also extends the provenance monitor concept to incorporate two additional properties that are required by the collection mechanism to support trustworthy provenance:

- **Authenticated Channel**. In distributed environments, provenance-aware systems must provide a means of assuring authenticity and integrity of provenance as it is communicated over open networks [4, 38, 48, 65]. LPM does not seek to provide a complete distributed provenance solution, but we wish to provide the required building blocks within the host for such a system to exist. LPM must therefore be able to monitor every network message that is sent or received by the host, and reliably explain these messages to other provenance-aware hosts in the network.
- **Attested Disclosure**. Layered provenance, where additional metadata is disclosed from higher operational layers, is a desirable feature in provenance-aware systems, as applications are able to report workflow semantics that are invisible to the operating system [42]. LPM must provide a gateway for upgrading low integrity user space disclosures before logging them in the high integrity provenance record. This is consistent with the Clark-Wilson Integrity model for upgrading or discarding low integrity inputs [13].

5.2 Threat Model

LPM is designed to securely collect provenance in the face of an adversary that has gained remote access to a provenance-aware host or network. Once inside the system, the attacker may attempt to remove provenance records, insert spurious information into those records, or find gaps in the provenance monitor's ability to record information flows. A network attacker may also attempt to forge or strip provenance from data in transit. Because captured provenance can be put to use in other applications, the adversary's goal may even be to target the provenance monitor itself. The implications and methods of such an attack are domain-specific. For example:

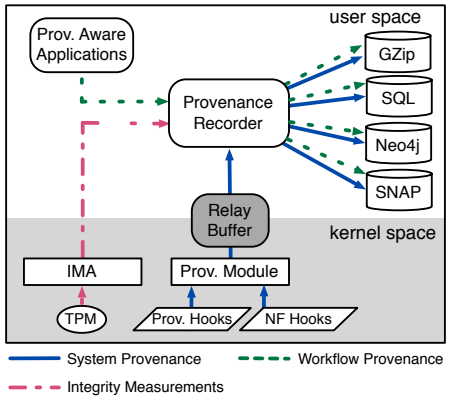

Fig. 1 Diagram of the LPM Framework. Kernel hooks report provenance to a recorder in userspace, which uses one of several storage back-ends. The recorder is also responsible for evaluating the integrity of workflow provenance prior to storing it.

- **Scientific Computing**: An adversary may wish to manipulate provenance in order to commit fraud, or to inject uncertainty into records to trigger a "Climategate"-like controversy [50].
- **Access Control**: When used to mediate access decisions [4, 44, 45, 47], an attacker could tamper with provenance in order to gain unauthorized access, or to perform a denial-of-service attack on other users by artificially escalating the security level of data objects.
- **Networks**: Provenance metadata can also be associated with packets in order to better understand network events in distributed systems [2, 65, 67]. Coordinating multiple compromised hosts, an attacker may attempt to send *unauthenticated* messages to avoid provenance generation and to perform data exfiltration.

LPM defines a provenance trusted computing base (TCB) to be the kernel mechanisms, provenance recorder, and storage back-ends responsible for the collection and management of provenance. *Provenance-aware applications are not considered part of the TCB.*

5.3 Design of LPM

An overview of the LPM architecture is shown in Figure 1. The LPM patch places a set of *provenance hooks* around the kernel; a *provenance module* then registers to control these hooks, and also registers several Netfilter hooks; the module then observes system events and transmits information via a relay buffer to a *provenance recorder* in user space that interfaces with a datastore. The recorder also accepts

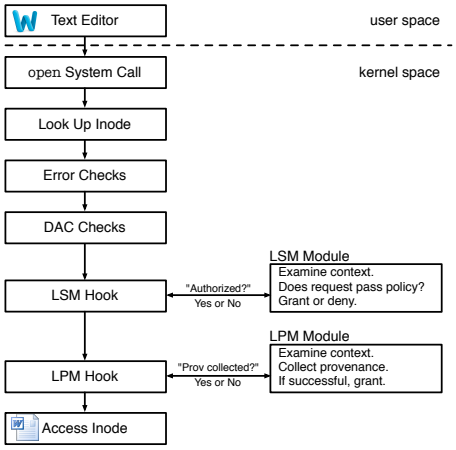

Fig. 2 Hook Architecture for the `open` system call. Provenance is collected *after* DAC and LSM checks, ensuring that it accurately reflects system activity. LPM will only deny the operation if it fails to generate provenance for the event.

disclosed provenance from applications after verifying their correctness using the Integrity Measurements Architecture (IMA) [51].

5.3.1 Provenance Hooks

The LPM patch introduces a set of hook functions in the Linux kernel. These hooks behave similarly to the LSM framework's security hooks in that they facilitate modularity, and default to taking no action unless a module is enabled. Each provenance hook is placed directly beneath a corresponding security hook. The return value of the security hook is checked prior to calling the provenance hook, thus assuring that the requested activity has been authorized prior to provenance capture. A workflow for the hook architecture is depicted in Figure 2. The LPM patch places over 170 provenance hooks, one for each of the LSM authorization hooks. In addition to the hooks that correspond to existing security hooks, LPM also supports a hook introduced by Hi-Fi that is necessary to preserve Lamport timestamps on network messages [30].

5.3.2 Netfilter Hooks

LPM uses Netfilter hooks to implement a cryptographic message commitment protocol. In Hi-Fi, provenance-aware hosts communicated by embedding a provenance sequence number in the IP options field [49] of each outbound packet. This approach allowed Hi-Fi to communicate as normal with hosts that were not provenance-

aware, but unfortunately was not secure against a network adversary. In LPM, provenance sequence numbers are replaced with Digital Signature Algorithm (DSA) signatures, which are space-efficient enough to embed in the IP Options field. LPM implements full DSA support in the Linux kernel by creating signing routines to use with the existing DSA verification function. DSA signing and verification occurs in the NetFilter `inet_local_out` and `inet_local_in` hooks. In `inet_local_out`, LPM signs over the immutable fields of the IP header, as well as the IP payload. In `inet_local_in`, LPM checks for the presence of a signature, then verifies the signature against a configurable list of public keys. If the signature fails, the packet is dropped before it reaches the recipient application, thus ensuring that there are no breaks in the continuity of the provenance log. The key store for provenance-aware hosts is obtained by a PKI and transmitted to the kernel during the boot process by writing to `securityfs`. LPM registers the Netfilter hooks with the highest priority levels, such that signing occurs just before transmission (i.e., after all other IPTables operations), and signature verification occurs just after the packet enters the interface (i.e., before all other IPTables operations).

5.3.3 Workflow Provenance

To support layered provenance while preserving our security goals, LPM requires a means of evaluating the integrity of user space provenance disclosures. To accomplish this, LPM Provenance Recorders make use of the Linux Integrity Measurement Architecture (IMA) [51]. IMA computes a cryptographic hash of each binary before execution, extends the measurement into a TPM Platform Control Register (PCR), and stores the measurement in kernel memory. This set of measurements can be used by the Recorder to make a decision about the integrity of the a Provenance-Aware Application (PAA) prior to accepting the disclosed provenance. When a PAA wishes to disclose provenance, it opens a new UNIX domain socket to send the provenance data to the Provenance Recorder. The Recorder uses its own UNIX domain socket to recover the process's pid, then uses the `/proc` filesystem to find the full path of the binary, then uses this information to look up the PAA in the IMA measurement list. The disclosed provenance is recorded only if the signature of PAA matches a known-good cryptographic hash.

A demonstration of this functionality is shown in Figure 3 for the popular ImageMagick utility [1]. ImageMagick contains a batch conversion tool for image reformatting, `mogrify`. Shown in Figure 3, `mogrify` reads and writes multiple files during execution, leading to an *overtainting* problem – at the kernel layer, LPM is forced to conservatively assume that all outputs were derived from all inputs, creating false dependencies in the provenance record. To address this, we extended the Provmon protocol to support a new message, `provmsg_imagemagick_convert`, which links an input file directly to its output file. When the recorder receives this message, it first checks the list of IMA measurements to confirm that ImageMag-

[1] See http://www.imagemagick.org

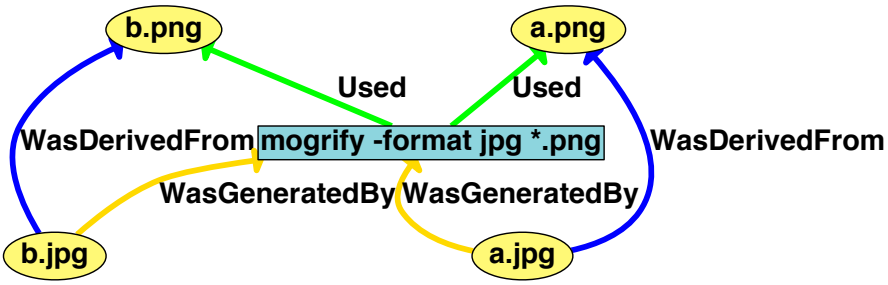

Fig. 3 A provenance graph of image conversion. Here, workflow provenance (*WasDerivedFrom*) encodes a relationship that more accurately identifies the output files' dependencies compared to only using kernel layer observations (*Used*, *WasGeneratedBy*).

ick is in a good state. If successful, it then annotates the existing provenance graph, connecting the appropriate input and output objects with *WasDerivedFrom* relationships. LPM presents a minimally modified version of ImageMagick that upports layered provenance at no additional cost over other provenance-aware systems [21, 43], and does so in a manner that provides assurance of the integrity of the provenance log.

5.4 Deploying LPM

We now demonstrate how we used LPM in the deployment of a secure provenance-aware system. We configured LPM to run on a physical machine with a Trusted Platform Module (TPM). The TPM provides a root of trust that allows for a measured boot of the system. The TPM also provides the basis for remote attestations to prove that LPM was in a known hardware and software configuration. The BIOS's core root of trust for measurement (CRTM) bootstraps a series of code measurements prior to the execution of each platform component. Once booted, the kernel then measures the code for user space components (e.g., provenance recorder) before launching them, through the use of the Linux Integrity Measurement Architecture (IMA)[51]. The result is then extended into TPM PCRs, which forms a verifiable chain of trust that shows the integrity of the system via a digital signature over the measurements. A remote verifier can use this chain to determine the current state of the system using TPM attestation.

We configured the system with Intel's Trusted Boot, which provides a secure boot mechanism, preventing system from booting into the environment where critical components (e.g., the BIOS, boot loader and the kernel) are modified. Intel tboot relies on the Intel TXT extensions to provide a secure execution environment. Additionally, we compiled support for IMA into the provenance-aware kernel, which is necessary in order for the LPM Recorder to be able to measure the integrity of provenance-aware applications.

After booting into the provenance-aware kernel, the runtime integrity of the TCB must also be assured. To protect the runtime integrity of the kernel, we deploy a Mandatory Access Control (MAC) policy, as implemented by Linux Security Modules. On our prototype deployments, we enabled SELinux's MLS policy, the security of which was formally modeled by Hicks et al. [25]. Refining the SELinux policy to prevent Access Vector Cache (AVC) denials on LPM components required minimal effort; the only denial we encountered was when using the PostgreSQL recorder, which was quickly remedied with the `audit2allow` tool. Preserving the integrity of LPM's user space components, such as the provenance recorder, was as simple as creating a new policy module. We created a policy module to protect the LPM recorder and storage back-end using the `sepolicy` utility. Uncompiled, the policy module was only 135 lines.

6 Analyzing the Security of Provenance Monitors

In this section, we briefly consider metrics for evaluating the provenance monitor solutions that we have discussed in this chapter, specifically Hi-Fi and LPM. We consider their evaluative metrics from both a coverage and performance perspective.

6.1 Completeness Analysis of Hi-Fi

Hi-Fi demonstrated that malware could be observed throughout a variety of elements of a malicious worm's life-cycle. For brevity, we do not discuss full simulation results, which are further discussed in [48].

6.1.1 Recording Malicious Behavior

Our first task is to show that the data collected by Hi-Fi is of sufficient fidelity to be used in a security context. We focus our investigation on detecting the activity of network-borne malware. A typical worm consists of several parts. First, an exploit allows it to execute code on a remote host. This code can be a dropper, which serves to retrieve and execute the desired payload, or it can be the payload itself. A payload can then consist of any number of different actions to perform on an infected system, such as exfiltrating data or installing a backdoor. Finally, the malware spreads to other hosts and begins the cycle again.

For our experiment, we chose to implement a malware generator which would allow us to test different droppers and payloads quickly and safely. The generator is similar in design to the Metasploit Framework [39], in that you can choose an exploit, dropper, and payload to create a custom attack. However, our tool also includes a set of choices for generating malware which automatically spreads from one host to another; this allows us to demonstrate what socket provenance can record about the flow of information between systems. The malware behaviors that we implement and test are drawn from Symantec's technical descriptions of actual Linux malware[58].

To collect provenance data, we prepare three virtual machines on a common subnet, all of which are running Hi-Fi. The attacker generates the malware on machine A and infects machine B by exploiting an insecure network daemon. The malware then spreads automatically from machine B to machine C. For each of the malicious behaviors we wish to test, we generate a corresponding piece of malware on machine A and launch it. Once C has been infected, we retrieve the provenance logs from all three machines for examination.

Each malware behavior that we test appears in some form in the provenance record. In each case, after filtering the log to view only the vulnerable daemon and its descendants, the behavior is clear enough to be found by manual inspection. Below we describe each behavior and how it appears in the provenance record.

6.1.2 Persistence and Stealth

Frequently, the first action a piece of malware takes is to ensure that it will continue to run for as long as possible. In order to persist after the host is restarted, the malware must write itself to disk in such a way that it will be run when the system boots. The most straightforward way to do this on a Linux system is to infect one of the startup scripts run by the `init` process. Our simulated malware has the ability to modify `rc.local`, as the Kaiten trojan does. This shows up clearly in the provenance log:

```
[6fe] write B:/etc/rc.local
```

In this case, the process with provid 0x6fe has modified `rc.local` on B's root filesystem. Persistent malware can also add cron jobs or infect system binaries to ensure that it is executed again after a reboot. Examples of this behavior are found in the Sorso and Adore worms. In our experiment, these behaviors result in similar log entries:

```
[701] write B:/bin/ps
```

for an infected binary, and

```
[710] write B:/var/spool/cron/root.new
[710] link B:/var/spool/cron/root.new to
           B:/var/spool/cron/root
[710] unlink B:/var/spool/cron/root.new
```

for an added cron job.

Some malware is even more clever in its approach to persistence. The Svat virus, for instance, creates a new C header file and places it early in the default include path. By doing this, it affects the code of any program which is subsequently compiled on that machine. We include this behavior in our experiment as well, and it appears simply as:

```
[707] write B:/usr/local/include/stdio.h
```

6.1.3 Remote Control

Once the malware has established itself as a persistent part of the system, the next step is to execute a payload. This commonly includes installing a backdoor which allows the attacker to control the system remotely. The simplest way to do this is to create a new root-level user on the system, which the attacker can then use to log in. Because of the way UNIX-like operating systems store their account databases, this is done by creating a new user with a UID of 0, making it equivalent to the root user. This is what the Zab trojan does, and when we implement this behavior, it is clear to see that the account databases are being modified:

```
[706] link (new) to B:/etc/passwd+
[706] write B:/etc/passwd+
```

```
[706] link B:/etc/passwd+ to B:/etc/passwd
[706] unlink B:/etc/passwd+
[706] link (new) to B:/etc/shadow+
[706] write B:/etc/shadow+
[706] link B:/etc/shadow+ to B:/etc/shadow
[706] unlink B:/etc/shadow+
```

A similar backdoor technique is to open a port which listens for connections and provides the attacker with a remote shell. This approach is used by many pieces of malware, including the Plupii and Millen worms. Our experiment shows that the provenance record includes the shell's network communication as well as the attacker's activity:

```
[744] exec B:/bin/bash -i
[744] socksend B:173
[744] sockrecv unknown
[744] socksend B:173
[751] exec B:/bin/cat /etc/shadow
[751] read B:/etc/shadow
[751] socksend B:173
[744] socksend B:173
[744] sockrecv unknown
[744] socksend B:173
[744] link (new) to B:/testfile
[744] write B:/testfile
```

Here, the attacker uses the remote shell to view `/etc/shadow` and to write a new file in the root directory. Since the attacker's system is unlikely to be running a trusted instance of Hi-Fi, we see "unknown" socket entries, which indicate data received from an unprovenanced host. Remote shells can also be implemented as "reverse shells," which connect from the infected host back to the attacker. Our tests on a reverse shell, such as the one in the Jac.8759 virus, show results identical to a normal shell.

6.1.4 Exfiltration

Another common payload activity is data exfiltration, where the malware reads information from a file containing password hashes, credit card numbers, or other sensitive information and sends this information to the attacker. Our simulation for this behavior reads the `/etc/shadow` file and forwards it in one of two ways. In the first test, we upload the file to a web server using HTTP, and in the second, we write it directly to a remote port. Both methods result in the same log entries:

```
[85f] read B:/etc/shadow
[85f] socksend B:1ae
```

Emailing the information to the attacker, as is done by the Adore worm, would create a similar record.

6.1.5 Spread

Our experiment also models three different mechanisms used by malware to spread to newly infected hosts. The first and simplest is used when the entire payload can be sent using the initial exploit. In this case, there does not need to be a separate dropper, and the resulting provenance log is the following (indentation is used to distinguish the two hosts):

```
[807] read A:/home/evil/payload
[807] socksend A:153
    [684] sockrecv A:153
    [684] write B:/tmp/payload
```

The payload is then executed, and the malicious behavior it implements appears in subsequent log entries.

Another mechanism, used by the Plupii and Sorso worms, is to fetch the payload from a remote web server. We assume the web server is unprovenanced, so the log once again contains "unknown" entries:

```
[7ff] read A:/home/evil/dropper
[7ff] socksend A:15b
    [685] sockrecv A:15b
    [685] write B:/tmp/dropper
    [6ef] socksend B:149
    [6ef] sockrecv unknown
    [6ef] write B:/tmp/payload
```

If the web server were a provenanced host, this log would contain host and socket IDs in the `sockrecv` entry corresponding to a `socksend` on the server.

Finally, to illustrate the spread of malware across several hosts, we tested a "relay" dropper which uses a randomly-chosen port to transfer the payload from each infected host to the next. The combined log of our three hosts shows this process:

```
[83f] read A:/home/evil/dropper
[83f] socksend A:159
    [691] sockrecv A:159
    [691] write B:/tmp/dropper
    [6f5] exec B:/tmp/dropper
[844] read A:/home/evil/payload
[844] socksend A:15b
    [6fc] sockrecv A:15b
    [6fc] write B:/tmp/payload
    [74e] read B:/tmp/dropper
    [74e] socksend B:169
        [682] sockrecv B:169
        [682] write C:/tmp/dropper
        [6e6] exec C:/tmp/dropper
    [750] read B:/tmp/payload
    [750] socksend B:16b
        [6ed] sockrecv B:16b
        [6ed] write C:/tmp/payload
```

Here we can see the attacker transferring both the dropper and the payload to the first victim using two different sockets. This victim then sends the dropper and the payload to the next host in the same fashion.

6.2 Security Analysis of LPM

We now turn our focus to LPM, which provides additional features for demonstraitng the provenance monitor concept beyond what Hi-Fi enforces. We demonstrate that LPM meets all of the required security goals for trustworthy whole-system provenance. In this analysis, we consider an LPM deployment on a physical machine that was enabled with the Provmon module, whihc mirrors the functionality of Hi-Fi.

Complete. We defined whole-system provenance as a complete description of agents (users, groups) controlling activities (processes) interacting with controlled data types during system execution (§ 5.1). LPM attempts to track these system objects through the placement of provenance hooks (§5.3.1), which directly follow each LSM authorization hook. The LSM's complete mediation property has been formally verified [15, 64]; in other words, there is an authorization hook prior to every security-sensitive operation. Because every interaction with a controlled data type is considered security-sensitive, we know that a provenance hook resides on all control paths to the provenance-sensitive operations. LPM is therefore capable of collecting complete provenance on the host.

It is important to note that, as a consequence of placing provenance hooks beneath authorization hooks, LPM is unable to record failed access attempts. However, inserting the provenance layer beneath the security layer ensures accuracy of the provenance record. Moreover, failed authorizations are a different kind of metadata than provenance because they do not describe processed data; this information is better handled at the security layer, e.g., by the SELinux Access Vector Cache (AVC) Log.

Tamperproof. The runtime integrity of the LPM trusted computing base is assured via the SELinux MLS policy, and we have written a policy module that protects the LPM user space components. Therefore, the only way to disable LPM would be to reboot the system into a different kernel; this action can be disallowed through secure boot techniques and is detectable by remote hosts via TPM attestation.

Verifiable. While we have not conducted an independent formal verification of LPM, our argument for its correctness is as follows. A provenance hook follows each LSM authorization hook in the kernel. The correctness of LSM hook placement has been verified through both static and dynamic analysis techniques [15, 19, 27]. Because an authorization hook exists on the path of every sensitive operation to controlled data types, and LPM introduces a provenance hook behind each authorization hook, LPM inherits LSM's formal assurance of complete mediation over controlled data types. This is sufficient to ensure that LPM can collect the provenance of ev-

ery sensitive operation on controlled data types in the kernel (i.e., whole-system provenance).

Authenticated Channel. Through use of Netfilter hooks [59], LPM embeds a DSA signature in every outbound network packet. Signing occurs immediately prior to transmission, and verification occurs immediately after reception, making it impossible for an adversary-controlled application running in user space to interfere. For both transmission and reception, the signature is invisible to user space. Signatures are removed from the packets before delivery, and LPM feigns ignorance that the options field has been set if `get_options` is called. Hence, LPM can enforce that all applications participate in the commitment protocol.

Prior to implementing our own message commitment protocol in the kernel, we investigated a variety of existing secure protocols. The integrity and authenticity of provenance identifiers could also be protected via IPsec [29], SSL tunneling,[2] or other forms of encapsulation [2, 65]. We elected to move forward with our approach because 1) it ensures the monitoring of all *all* processes and network events, including non-IP packets, 2) it does not change the number of packets sent or received, ensuring that our provenance mechanism is minimally invasive to the rest of the Linux network stack, and 3) it preserves compatibility with non-LPM hosts. An alternative to DSA signing would be HMAC [6], which offers better performance but requires pairwise keying and sacrifices the non-repudiation policy; BLS, which approaches the theoretical maximum security parameter per byte of signature [7]; or online/offline signature schemes [9, 16, 20, 53].

Authenticated Disclosures. We make use of IMA to protect the channel between LPM and provenance-aware applications wishing to disclose provenance. IMA is able to prove to the provenance recorder that the application was unmodified at the time it was loaded into memory, at which point the recorder can accept the provenance disclosure into the official record. If the application is known to be correct (e.g., through formal verification), this is sufficient to establish the runtime integrity of the application. However, if the application is compromised after execution, this approach is unable to protect against provenance forgery.

A separate consideration for all of the above security properties are Denial of Service (DoS) attacks. *DoS attacks on LPM do not break its security properties*. If an attacker launches a resource exhaustion attack in order to prevent provenance from being collected, all kernel operations will be disallowed and the host will cease to function. If a network attacker tampers with a packet's provenance identifier, the packet will not be delivered to the recipient application. In all cases, the provenance record remains an accurate reflection of system events.

[2] See http://docs.oracle.com/cd/E23823_01/html/816-5175/kssl-5.html

7 Current and Future Challenges to Provenance for Forensics

In this chapter, we have discussed the provenance monitor approach to secure and trustworthy collection of data provenance, which can be an extraordinary source of metadata for forensics investigators. The ability to use provenance for this goal is predicated on its complete collection in an environment that cannot be tampered. As we discussed through our exploration of the Hi-Fi and Linux Provenance Modules system, the goals of a provenance monitor can be seen as a superset of reference monitor goals because of the need for integration of layers and the notion of attested disclosure, which are properties unique to the provenance environment.

A common limitation shared by provenance collection systems, including not only Hi-Fi and LPM but also proposals such as SPADE and PASS, is that provenance collection at the operating system layer demands large amounts of storage. For example, in short-lived benchmark trials, each of these systems generated gigabytes of provenance over the course of just a few minutes [21, 43]. There are some promising methods of reducing the costs of collection. Ma et al.'s ProTracer system offers dramatic improvement in storage cost by making use of a hybrid audit-taint model for provenance collection [36]. ProTracer only flushes new provenance records to disk when system writes occur (e.g., file write, packet transmission); on system reads, ProTracer propagates a taint label between kernel objects in memory. By leveraging this approach along with other garbage collection techniques [33, 32], ProTracer reduces the burden of provenance storage to just tens of megabytes per day. Additionally, Bates et al. [3] considered that much of the provenance collected by high-fidelity systems is simply *uninteresting*; in other words, it is the collection of data that does not provide new infomration essenial to system reconstruction or forensic analysis, for example. By focusing on information deemed important through its inclusion in the system's trusted computing base as inferred by its mandatory access policy, it is possible to identify the subset of processes and applications critical to enforcing the system's security goals. By focusing on these systems, the amount of data that needs to be collected can be reduced by over 90%. Such an approach can be complementary to other proposals for data transformation to assure the efficient storage of provenance metadata [11] and the use of techniques such as provenance deduplication [62, 61].

Extending provenance beyond a single host to distributed systems also poses a considerable challenge. In distributed environments, provenance-aware hosts must attest the integrity of one another before sharing provenance metadata [34], or in layered provenance systems where there is no means to attest provenance disclosures [42]. Kernel-based provenance mechanisms [43, 48] and sketches for trusted provenance architectures [34, 38] fall short of providing a fully provenance-aware system for distributed, malicious environments. Complicating matters further, data provenance is conceptualized in dramatically different ways throughout the literature, such that any solution to provenance security would need to be general enough to support the needs of a variety of diverse communities. Extending provenance monitors into these environments can provide a wealth of new information to the forensics investigator but must be carefully designed and implemented.

While we focus on the collection of provenance in this chapter, it is also important to be able to efficiently query the provenance once it is Provenance queries regarding transitive causes/effects of a *single* system state or event can be answered by a recursive procedure that retrieves relevant portions of a provenance graph [66, 65]. While such queries are useful in many applications, e.g., to find root causes of a detected policy violation, further research is necessary into efficient query languages to allow system operators to perform more complex queries that can identify user-specified subgraphs from the collected provenance in a manner that is easily usable and that facilitate inference of analytics.

To conclude, provenance represents a powerful new means for gathering data about a system for a forensics investigator. Being able to establish the context within which data was created and generating a chain of custody describing how the data came to take its current form can provide vast new capabilities. However, as the systems discussed in this chapter demonstrate, ensuring that provenance is securely collected is a challenging task. Future systems can build from existing work to addresses the challenges we outlined above in order to bring the promises of provenance to practical reality.

Acknowledgements This work draws in part from [38], [48], and [5]. We would like to thank our co-authors of those works, including Patrick McDaniel, Thomas Moyer, Stephen McLaughlin, Erez Zadok, Marianne Winslett, and Radu Sion, as well as reviewers of those original papers who provided us with valuable feedback. This work is supported in part by the U.S. National Science Foundation under grants CNS-1540216, CNS-1540217, and CNS-1540128.

References

1. Aldeco-Pérez, R., Moreau, L.: Provenance-based Auditing of Private Data Use. In: Proceedings of the 2008 International Conference on Visions of Computer Science: BCS International Academic Conference, VoCS'08, pp. 141–152. British Computer Society, Swinton, UK, UK (2008)
2. Bates, A., Butler, K., Haeberlen, A., Sherr, M., Zhou, W.: Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. SENT (2014)
3. Bates, A., Butler, K.R.B., Moyer, T.: 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 (2015)
4. Bates, A., Mood, B., Valafar, M., Butler, K.: 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, pp. 277–284. ACM, New York, NY, USA (2013). DOI 10.1145/2435349.2435389
5. Bates, A., Tian, D., Butler, K.R.B., Moyer, T.: Trustworthy Whole-System provenance for the linux kernel. In: Proceedings of the 2015 USENIX Security Symposium (Security'15). Washington, DC USA (2015)
6. Bellare, M., Canetti, R., Krawczyk, H.: Keyed Hash Functions and Message Authentication. In: Proceedings of Crypto'96, *LNCS*, vol. 1109, pp. 1–15 (1996)
7. Boneh, D., Lynn, B., Shacham, H.: Short Signatures from the Weil Pairing. In: C. Boyd (ed.) Advances in Cryptology – ASIACRYPT 2001 (2001)
8. Carata, L., Akoush, S., Balakrishnan, N., Bytheway, T., Sohan, R., Seltzer, M., Hopper, A.: A primer on provenance. Commun. ACM **57**(5), 52–60 (2014). DOI 10.1145/2596628. URL http://doi.acm.org/10.1145/2596628
9. Catalano, D., Di Raimondo, M., Fiore, D., Gennaro, R.: Off-line/On-line Signatures: Theoretical Aspects and Experimental Results. In: PKC'08: Proceedings of the Practice and theory in public key cryptography, 11th international conference on Public key cryptography, pp. 101–120. Springer-Verlag, Berlin, Heidelberg (2008)
10. Centers for Medicare & Medicaid Services: The Health Insurance Portability and Accountability Act of 1996 (HIPAA) (1996). URL http://www.cms.hhs.gov/hipaa/
11. Chapman, A., Jagadish, H., Ramanan, P.: Efficient Provenance Storage. In: Proceedings of the 2008 ACM Special Interest Group on Management of Data Conference, SIGMOD'08 (2008)
12. Chiticariu, L., Tan, W.C., Vijayvargiya, G.: DBNotes: A Post-it System for Relational Databases Based on Provenance
13. Clark, D.D., Wilson, D.R.: A Comparison of Commercial and Military Computer Security Policies. In: Proceedings of the IEEE Symposium on Security and Privacy. Oakland, CA, USA (1987)
14. Department of Homeland Security: A Roadmap for Cybersecurity Research (2009)
15. Edwards, A., Jaeger, T., Zhang, X.: Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS'02 (2002)
16. Even, S., Goldreich, O., Micali, S.: On-line/off-line Digital Signatures. In: Proceedings on Advances in cryptology, CRYPTO '89, pp. 263–275. Springer-Verlag New York, Inc., New York, NY, USA (1989). URL http://portal.acm.org/citation.cfm?id=118209.118233

17. Foster, I.T., Vöckler, J.S., Wilde, M., Zhao, Y.: Chimera: AVirtual Data System for Representing, Querying, and Automating Data Derivation. In: Proceedings of the 14th Conference on Scientific and Statistical Database Management, SSDBM'02 (2002)
18. Frew, J., Bose, R.: Earth System Science Workbench: A Data Management Infrastructure for Earth Science Products. In: Proceedings of the 13th International Conference on Scientific and Statistical Database Management, pp. 180–189. IEEE Computer Society (2001)
19. Ganapathy, V., Jaeger, T., Jha, S.: Automatic placement of authorization hooks in the linux security modules framework. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS '05, pp. 330–339. ACM, New York, NY, USA (2005). DOI 10.1145/1102120.1102164
20. Gao, C.z., Yao, Z.a.: A Further Improved Online/Offline Signature Scheme. Fundam. Inf. **91**, 523–532 (2009). URL http://portal.acm.org/citation.cfm?id=1551775.1551780
21. Gehani, A., Tariq, D.: SPADE: Support for Provenance Auditing in Distributed Environments. In: Proceedings of the 13th International Middleware Conference, Middleware '12 (2012)
22. Glavic, B., Alonso, G.: Perm: Processing Provenance and Data on the Same Data Model Through Query Rewriting. In: Proceedings of the 25th IEEE International Conference on Data Engineering, ICDE '09 (2009)
23. Hall, E.: The Arnolfini Betrothal: Medieval Marriage and the Enigma of Van Eyck's Double Portrait. University of California Press, Berekely, CA (1994)
24. Hasan, R., Sion, R., Winslett, M.: 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. San Francisco, CA, USA (2009)
25. Hicks, B., Rueda, S., St.Clair, L., Jaeger, T., McDaniel, P.: A Logical Specification and Analysis for SELinux MLS Policy. ACM Trans. Inf. Syst. Secur. **13**(3), 26:1–26:31 (2010). DOI 10.1145/1805874.1805982
26. Holland, D.A., Bruan, U., Maclean, D., Muniswamy-Reddy, K.K., Seltzer, M.I.: Choosing a Data Model and Query Language for Provenance. IPAW'08 (2008)
27. Jaeger, T., Edwards, A., Zhang, X.: Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework. ACM Trans. Inf. Syst. Secur. **7**(2), 175–205 (2004). DOI 10.1145/996943.996944
28. Jones, S.N., Strong, C.R., Long, D.D.E., Miller, E.L.: Tracking Emigrant Data via Transient Provenance. In: 3rd Workshop on the Theory and Practice of Provenance, TAPP'11 (2011)
29. Kent, S., Atkinson, R.: RFC 2406: IP Encapsulating Security Payload (ESP) (1998)
30. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM **21**(7), 558–565 (1978). DOI 10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563
31. Lampson, B.W.: A Note on the Confinement Problem. Communications of the ACM **16**(10), 613–615 (1973)
32. Lee, K.H., Zhang, X., Xu, D.: High Accuracy Attack Provenance via Binary-based Execution Partition. In: Proceedings of the 20th ISOC Network and Distributed System Security Symposium, NDSS (2013)
33. Lee, K.H., Zhang, X., Xu, D.: LogGC: Garbage Collecting Audit Log. In: Proceedings of the 2013 ACM Conference on Computer and Communications Security, CCS (2013)
34. Lyle, J., Martin, A.: Trusted Computing and Provenance: Better Together. In: 2nd Workshop on the Theory and Practice of Provenance, TaPP'10 (2010)
35. Ma, S., Lee, K.H., Kim, C.H., Rhee, J., Zhang, X., Xu, D.: Accurate, low cost and instrumentation-free security audit logging for windows. In: Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC 2015, pp. 401–410. ACM (2015). DOI 10.1145/2818000.2818039
36. Ma, S., Zhang, X., Xu, D.: ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In: Proceedings of the 23rd ISOC Network and Distributed System Security Symposium, NDSS (2016)
37. Macko, P., Seltzer, M.: A General-purpose Provenance Library. In: 4th Workshop on the Theory and Practice of Provenance, TaPP'12 (2012)

38. McDaniel, P., Butler, K., McLaughlin, S., Sion, R., Zadok, E., Winslett, M.: Towards a Secure and Efficient System for End-to-End Provenance. In: Proceedings of the 2nd conference on Theory and practice of provenance. USENIX Association, San Jose, CA, USA (2010)
39. Metasploit Project. http://www.metasploit.com
40. Moreau, L., Groth, P., Miles, S., Vazquez-Salceda, J., Ibbotson, J., Jiang, S., Munroe, S., Rana, O., Schreiber, A., Tan, V., Varga, L.: The provenance of electronic data. Commun. ACM **51**(4), 52–58 (2008). DOI http://doi.acm.org/10.1145/1330311.1330323
41. Mouallem, P., Barreto, R., Klasky, S., Podhorszki, N., Vouk, M.: Tracking Files in the Kepler Provenance Framework. In: SSDBM 2009: Proceedings of the 21st International Conference on Scientific and Statistical Database Management (2009)
42. Muniswamy-Reddy, K.K., Braun, U., Holland, D.A., Macko, P., Maclean, D., Margo, D., Seltzer, M., Smogor, R.: Layering in Provenance Systems. In: Proceedings of the 2009 Conference on USENIX Annual Technical Conference, ATC'09 (2009)
43. Muniswamy-Reddy, K.K., Holland, D.A., Braun, U., Seltzer, M.: Provenance-aware Storage Systems. In: Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference, Proceedings of the 2006 Conference on USENIX Annual Technical Conference (2006)
44. Nguyen, D., Park, J., Sandhu, R.: 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, pp. 4–4. USENIX Association, Berkeley, CA, USA (2012)
45. Ni, Q., Xu, S., Bertino, E., Sandhu, R., Han, W.: An Access Control Language for a General Provenance Model. In: Secure Data Management (2009)
46. Pancerella, C., Hewson, J., Koegler, W., Leahy, D., Lee, M., Rahn, L., Yang, C., Myers, J.D., Didier, B., McCoy, R., Schuchardt, K., Stephan, E., Windus, T., Amin, K., Bittner, S., Lansing, C., Minkoff, M., Nijsure, S., von Laszewski, G., Pinzon, R., Ruscic, B., Wagner, A., Wang, B., Pitz, W., Ho, Y.L., Montoya, D., Xu, L., Allison, T.C., Green Jr., W.H., Frenklach, M.: Metadata in the Collaboratory for Multi-Scale Chemical Science. In: Proceedings of the 2003 international conference on Dublin Core and metadata applications: supporting communities of discourse and practice—metadata research & applications, pp. 13:1–13:9. Dublin Core Metadata Initiative (2003)
47. Park, J., Nguyen, D., Sandhu, R.: A Provenance-Based Access Control Model. In: Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST), pp. 137–144 (2012). DOI 10.1109/PST.2012.6297930
48. Pohly, D.J., McLaughlin, S., McDaniel, P., Butler, K.: Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In: Proceedings of the 2012 Annual Computer Security Applications Conference, ACSAC '12. Orlando, FL, USA (2012)
49. Postel, J.: RFC 791: Internet protocol (1981)
50. Revkin, A.C.: Hacked E-mail is New Fodder for Climate Dispute. New York Times **20** (2009)
51. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: Proceedings of the 13th USENIX Security Symposium. San Diego, CA, USA (2004)
52. Sar, C., Cao, P.: Lineage file system. Online at http://crypto.stanford.edu/cao/lineage.html (2005)
53. Shamir, A., Tauman, Y.: Improved Online/Offline Signature Schemes. In: Advances in Cryptology — CRYPTO 2001 (2001)
54. Silva, C.T., Anderson, E.W., Santos, E., Freire, J.: Using VisTrails and Provenance for Teaching Scientific Visualization. Comput. Graph. Forum () **30**(1), 75–84 (2011)
55. Sion, R.: Strong worm. In: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems (2008)
56. Spillane, R.P., Sears, R., Yalamanchili, C., Gaikwad, S., Chinni, M., Zadok, E.: Story Book: An efficient extensible provenance framework. In: First Workshop on the Theory and Practice of Provenance. USENIX (2009)
57. Sundararaman, S., Sivathanu, G., Zadok, E.: Selective versioning in a secure disk system. In: Proceedings of the 17th conference on Security symposium (2008)

58. Symantec: Symantec Security Response. http://www.symantec.com/security_response (2015)
59. The Netfilter Core Team: The Netfilter Project: Packet Mangling for Linux 2.4. http://www.netfilter.org/ (1999). URL http://crypto.stanford.edu/ cao/lineage.html
60. U.S. Code: 22 U.S. Code § 2778 - Control of arms exports and imports (1976). URL https://www.law.cornell.edu/uscode/text/22/2778
61. Xie, Y., Feng, D., Tan, Z., Chen, L., Muniswamy-Reddy, K.K., Li, Y., Long, D.D.: A Hybrid Approach for Efficient Provenance Storage. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM '12 (2012)
62. Xie, Y., Muniswamy-Reddy, K.K., Long, D.D.E., Amer, A., Feng, D., Tan, Z.: Compressing Provenance Graphs. In: Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance (2011)
63. Zanussi, T., Yaghmour, K., Wisniewski, R., Moore, R., Dagenais, M.: relayfs: An efficient unified approach for transmitting data from kernel to user space. In: Proceedings of the 2003 Linux Symposium, Ottawa, ON, Canada, pp. 494–506 (2003)
64. Zhang, X., Edwards, A., Jaeger, T.: Using CQUAL for Static Analysis of Authorization Hook Placement. In: Proceedings of the 11th USENIX Security Symposium (2002)
65. Zhou, W., Fei, Q., Narayan, A., Haeberlen, A., Loo, B.T., Sherr, M.: Secure Network Provenance. In: ACM Symposium on Operating Systems Principles (SOSP) (2011)
66. Zhou, W., Mapara, S., Ren, Y., Haeberlen, A., Ives, Z., Loo, B.T., Sherr, M.: Distributed time-aware provenance. In: Proc. VLDB (2013)
67. Zhou, W., Sherr, M., Tao, T., Li, X., Loo, B.T., Mao, Y.: Efficient Querying and Maintenance of Network Provenance at Internet-Scale. In: Proceedings of the 2010 ACM SIGMOD International Conference on Measurement of Data (2010)