Practical Whole-System Provenance Capture

7 authors, including:

Thomas Pasquier
University of Bristol
31 PUBLICATIONS **335** CITATIONS

SEE PROFILE

Xueyuan Han
Harvard University
9 PUBLICATIONS **23** CITATIONS

SEE PROFILE

Thomas Moyer
Massachusetts Institute of Technology
19 PUBLICATIONS **238** CITATIONS

SEE PROFILE

David Eyers
University of Otago
127 PUBLICATIONS **1,100** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project   CloudSafetyNet: End-to-end application security in the cloud explores the use of Information Flow Control to achieve greater security in cloud computing. View project

Project   OPERA Project View project

Practical Whole-System Provenance Capture

Thomas Pasquier
Harvard University
Cambridge, USA
tfjmp@seas.harvard.edu

Xueyuan Han
Harvard University
Cambridge, USA
hanx@g.harvard.edu

Mark Goldstein
Harvard University
Cambridge, USA
markgoldstein@g.harvard.edu

Thomas Moyer*
UNC Charlotte
Charlotte, USA
tmoyer2@uncc.edu

David Eyers
University of Otago
Dunedin, New Zealand
dme@cs.otago.ac.nz

Margo Seltzer
Harvard University
Cambridge, Massachusetts
margo@eecs.harvard.edu

Jean Bacon
University of Cambridge
Cambridge, United Kingdom
jean.bacon@cl.cam.ac.uk

ABSTRACT

Data provenance describes how data came to be in its present form. It includes data sources and the transformations that have been applied to them. Data provenance has many uses, from forensics and security to aiding the reproducibility of scientific experiments. We present CamFlow, a whole-system provenance capture mechanism that integrates easily into a PaaS offering. While there have been several prior whole-system provenance systems that captured a comprehensive, systemic and ubiquitous record of a system's behavior, none have been widely adopted. They either A) impose too much overhead, B) are designed for long-outdated kernel releases and are hard to port to current systems, C) generate too much data, or D) are designed for a single system. CamFlow addresses these shortcoming by: 1) leveraging the latest kernel design advances to achieve efficiency; 2) using a self-contained, easily maintainable implementation relying on a Linux Security Module, NetFilter, and other existing kernel facilities; 3) providing a mechanism to tailor the captured provenance data to the needs of the application; and 4) making it easy to integrate provenance across distributed systems. The provenance we capture is streamed and consumed by tenant-built auditor applications. We illustrate the usability of our implementation by describing three such applications: demonstrating compliance with data regulations; performing fault/intrusion detection; and implementing data loss prevention. We also show how CamFlow can be leveraged to capture meaningful provenance without modifying existing applications.

*Work was completed while author was a member of technical staff at MIT Lincoln Laboratory.

CCS CONCEPTS

• **General and reference** → **Design**; • **Security and privacy** → **Operating systems security**; *Information flow control*;

KEYWORDS

Data Provenance, Whole-system provenance, Linux Kernel

1 INTRODUCTION

Provenance was originally a formal set of documents describing the origin and ownership history of a work of art. These documents are used to guide the assessment of the authenticity and quality of the item. In a computing context, *data provenance* represents, in a formal manner, the relationships between data items (*entities*), transformations applied to those items (*activities*), and persons or organisations associated with the data and transformations (*agents*). It can be understood as the record of the origin and transformations of data within a system [20].

Data provenance has a wide range of applications, from facilitating the reproduction of scientific results [39] to verifying compliance with legal regulations (see § 2). For example, in earlier work [73], we discussed how provenance can be used to demonstrate compliance with the French data privacy agency guidelines in a cloud-connected smart home system; in other work [77] we proposed data provenance to audit compliance with privacy policies in the Internet of Things.

In this paper, we examine the deployment of Provenance as a Service in PaaS clouds, as illustrated in Fig. 1. Tenants' application instances (e.g., Docker containers) run on top of a cloud-provider-managed Operating System (OS). We include in this OS a provenance capture module that records all interactions between processes and kernel objects (i.e., files, sockets, IPC etc.). We also record network packet information, to track the exchange of data
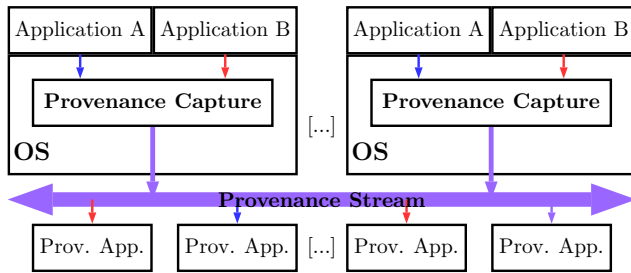
Figure 1: CamFlow cloud provenance architecture.

Figure 2: A W3C ProvDM compliant provenance graph.

across machines. We stream the captured provenance data to tenant-provided *provenance applications* (which can process and/or store the provenance). We focus on the capture mechanism and discuss four concrete provenance applications in § 7.

Two of us were deeply involved in the development of prior, similar provenance capture systems: PASS [66] and LPM [13]. In both cases, we struggled to keep abreast with current OS releases and ultimately declared the code unmaintainable. CamFlow achieves the goals of prior projects, but with a cleaner architecture that is more maintainable – we have already upgraded CamFlow several more times than was ever accomplished with the prior systems. The adoption of the Linux Security Modelu (LSM) architecture and support for NetFilters makes this architecturally possible. Ultimately, our goal is to have provenance integrated into the mainline Linux kernel. Thus, we developed a fully self-contained Linux kernel module, discussed in detail in § 4.

A second challenge in whole system provenance capture is the sheer volume of data, *"which [...] continues to grow indefinitely over time"* [13]. Recent work [12, 75] addresses this issue by limiting capture based on security properties, on the assumption that only sensitive objects are of interest. In practice, after the design stage, many provenance applications answer only a well-defined set of queries. We extend the "take what you need" concept [12] beyond the security context, with detailed policies that capture requirements to meet the exact needs of a provenance application.

The contributions of this work are: 1) a practical implementation of whole-system provenance that can easily be maintained and deployed; 2) a policy enforcement mechanism that finely tunes the provenance captured to meet application requirements; 3) an implementation that relies heavily on standards and current practices, interacting with widely used software solutions, designed in a modular fashion to interoperate with existing software; 4) a new, simpler approach to retrofit provenance into existing "provenance-unaware" applications; 5) a demonstration of several provenance applications; and 6) an extension of provenance capture to containers and shared memory.

2 THE SCOPE OF PROVENANCE

Data provenance – also called data *lineage*, or *pedigree* – was originally introduced to understand the origin of data in a database [18, 92]. Whole-system provenance [78] goes further, tracking file metadata and transient system 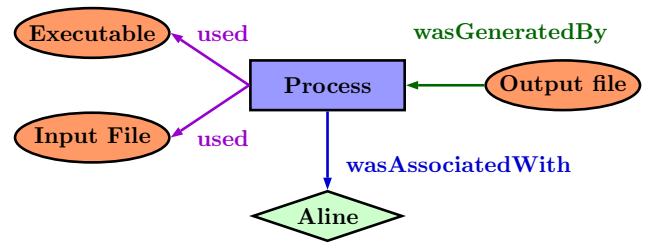objects. It gives a complete picture of a system from *initialisation* to *shutdown*. We begin by examining a number of illustrative provenance use cases.

Retrospective Security [7, 54, 79, 91] is the detection of security violations after execution, by determining whether an execution complied with a set of rules. We use provenance to detect such violations. In particular, we verify compliance with contractual or legal regulations [73]. We discuss a concrete implementation in § 7.1.

Reproducibility: Many have explored using provenance to guarantee the reproducibility of computational experiments or to explain the irreproducibility of such results [42, 61, 85]. It is increasingly common for such experiments to run on a PaaS platform. Creating a detailed record of the data ((e.g., input dataset, parameters etc.)) and software used (e.g., libraries, environment etc.) and the dependencies between them enables reproduction, by creation of an environment as close as possible to that of the original execution. Differential provenance techniques [23] can provide information to explain why two executions produce different results, which might otherwise be difficult to understand, e.g., in a long and complex scientific workflow.

Deletion and Policy Modification: When dealing with highly sensitive data it is important to be able to delete every document containing certain information, including derived documents. It is also necessary to propagate changes to the access policy of such information. For example, EU data protection law includes "secure delete" and the "right to be forgotten". This is achieved by capturing and recording the relationships between documents (and their system representations, e.g., as files). These complex relationships must be taken into account when deleting information or updating a policy.

Intrusion/Fault Detection: In cloud environments, several instances of the same system/application run in parallel. Through analysis of provenance data generated during those executions, we can build a model of normal behavior. By comparing the behavior of an executing task to this model, we can detect abnormal behavior. We discuss this use case in more detail in § 7.2.

Fault Injection: Fault injection is a technique commonly used to test applications' resistance to failure [70]. Colleagues at UCSC are using a CamFlow-generated provenance graph to record the execution of a distributed storage system to help them discover points of failure, leading to better fault-injection testing.

While CamFlow satisfies all of the above application requirements, in this paper, we focus on the construction of the capture mechanism itself. In § 7, we discuss some of the provenance applications that we have built using CamFlow.
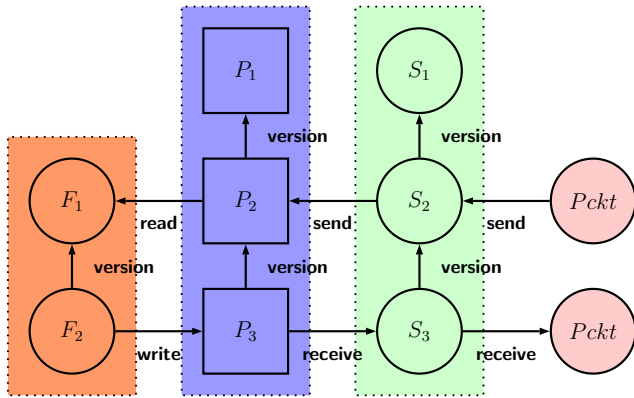
Figure 3: CamFlow-Provenance partial graph example. A process P reads information from a file F, sends the information over a socket S, and updates F based on information received through S.

3 PROVENANCE DATA MODEL

We extend the W3C PROV Data Model (PROV-DM) [15], which is used by most of the provenance community. A provenance graph represents *entities*, *activities* and *agents*. At the OS level, *entities* are typically kernel data objects: files, messages, packets etc., but also xattributes, inode attributes, exec parameters, network addresses etc. *Activities* are typically processes carrying out manipulations on entities. *Agents* are persons or organisations (e.g., users, groups etc.) on whose behalf the activities operate. In the provenance graph, all these elements are nodes, connected by edges representing different types of interactions. For example, Fig. 2 depicts that an output file *was generated by* a process that *was associated with* a user Aline, and *used* an executable and an input file.

CamFlow records how information is exchanged within a system through system calls. Some calls represent an exchange of information at a point in time e.g., `read`, `write`; others may create shared state, e.g., `mmap`. The former can easily be expressed within the PROV-DM model, the latter is more complex to model.

Previous implementations managed shared states (i.e., POSIX shared memory and mmap files, which both use the same underlying Linux kernel mechanism) by recording the action of "associating" the shared memory with a process (e.g., `shmget`, `mmap_file`). These shared states are either ignored during queries or make the queries significantly more complex. Indeed, information may flow implicitly (from the system perspective) between processes, without being represented in the provenance graph. Without a view of information flow within an application (we discuss how intra-application provenance can be appended to the graph in § 4.5), it is impossible to know if/when a process is writing to some "associated" shared memory at the system-call level. Therefore, we conservatively assume that any incoming or outgoing flow to/from the process implies an incoming or outgoing flow to/from the shared state.

Provenance graphs are acyclic. A central concept of a CamFlow graph is the *state-version* of kernel objects, which guarantees that the graph remains acyclic. A kernel object (e.g., an inode, a process etc.) is represented as a succession of nodes, which represent an

object changing state. We conservatively assume that any incoming information flow generates a new object state. Nodes associated with the same kernel object share the same object id, machine id, and boot id (network packets follow different rules described in § 4.2). Other node attributes may change, depending on the incoming information (e.g., `setattr` might modify an `inode`'s `mode`). Fig. 3 illustrates CamFlow versioning.

In Fig. 3, we group nodes belonging to the same *entity* or *activity* (F, P and S), based on shared attributes between those nodes (i.e., `boot_id`, `machine_id`, and `node_id`). In the cloud context, those groups can be further nested into larger groups representing individual machines or particular clusters of machines. For example, when trying to understand interactions between Docker containers [1], we can create groups based on namespaces (i.e., UTS, IPC, mount, PID and network namespaces), control groups, and/or security contexts. The combination of these process properties differentiate processes belonging to different Docker containers. Provenance applications determine their own meaningful groupings.

```
1  "ABAAAAAAACAe9wIAAAAAAE7aeaI+200UAAAAAAAAAAA=": {
2      "cf:id": "194334",
3      "prov:type": "fifo",
4      "cf:boot_id": 2725894734,
5      "cf:machine_id": 340646718,
6      "cf:version": 0,
7      "cf:date": "2017:01:03T16:43:30",
8      "cf:jiffies": "4297436711",
9      "cf:uid": 1000,
10     "cf:gid": 1000,
11     "cf:mode": "0x1180",
12     "cf:secctx": "unconfined_u:unconfined_r:
        unconfined_t:s0-s0:c0.c1023",
13     "cf:ino": 51964,
14     "cf:uuid": "32b7218a-01a0-c7c9-17b1-666f200b8912",
15     "prov:label": "[fifo] 0"
16 }
```

Listing 1: PROV-JSON formatted inode entry.

```
1  "QAAAAAAAQIANAAAAAAAAE7aeaI+200UAAAAAAAAAAA=": {
2      "cf:id": "13",
3      "prov:type": "write",
4      "cf:boot_id": 2725894734,
5      "cf:machine_id": 340646718,
6      "cf:date": "2017:01:03T16:43:30",
7      "cf:jiffies": "4297436711",
8      "prov:label": "write",
9      "cf:allowed": "true",
10     "prov:activity": "AQAAAAAAEAf9wIAAAAAAE7aeaI+200
        UAQAAAAAAAA=",
11     "prov:entity": "ABAAAAAAACAe9wIAAAAAAE7aeaI+200
        UAQAAAAAAAA=",
12     "cf:offset": "0"
13 }
```

Listing 2: PROV-JSON formatted write entry.

Listing 1 and Listing 2 show an inode and a write relationship in the provenance graph, in PROV-JSON [47] format.[1] We also developed a compact binary format.

4 CAPTURING SYSTEM PROVENANCE

CamFlow builds upon and learns from previous OS provenance capture mechanisms, namely PASS [65, 66], Hi-Fi [78] and LPM [13].

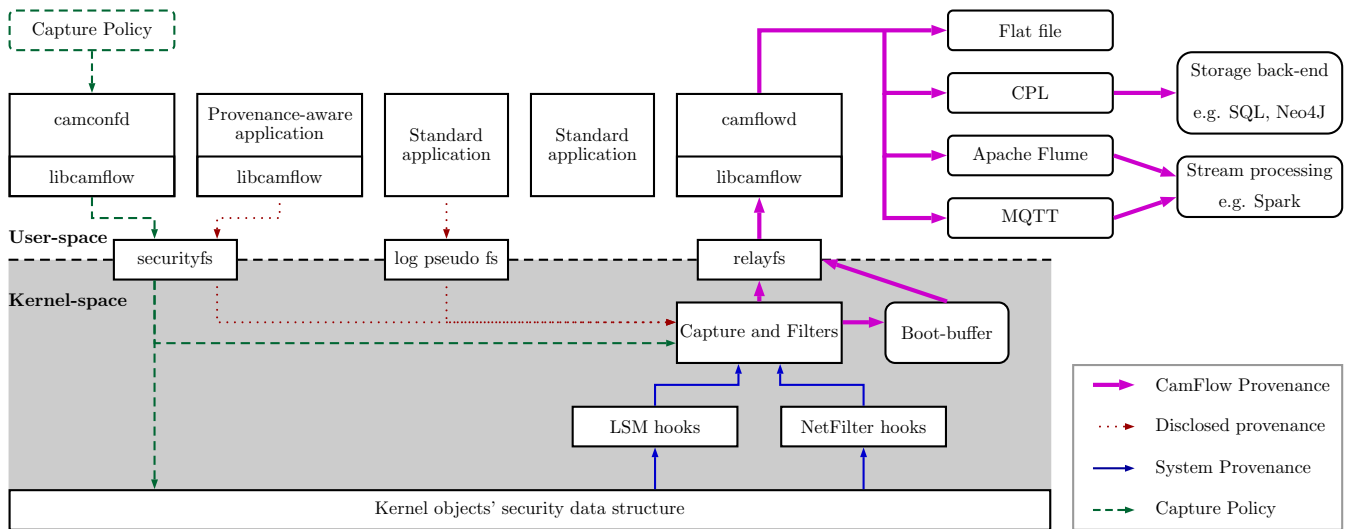[1]See https://github.com/CamFlow/CamFlow.github.io/wiki/JSON-output.

Figure 4: Architecture overview.

Our strategy for CamFlow is that it: 1) is easy to maintain, 2) uses existing kernel mechanisms whenever possible, 3) does not duplicate existing mechanisms, and 4) can be integrated into the mainline Linux kernel. Beyond being good practice, the first three requirements are essential to realize the last requirement (adoption in the mainline Linux kernel). We also incorporated requirements for usability and extendibility.

Fig. 4 gives an overview of the architecture. We use LSM and NetFilter (NF) hooks to capture provenance and `relayfs` [95] to transfer the data to user space where it can be stored or analyzed. Applications can augment system level provenance with application-specific details through a pseudofile interface. This is restricted to the owner of the capability `CAP_AUDIT_WRITE`, following Linux development recommendations to re-use existing capabilities, rather than from creating new ones. Applications owning the `CAP_AU-DIT_CONTROL` capability can collect configure CamFlow to capture a subset of the provenance (see § 5). We provide a library whose API abstracts interactions with `relayfs` and the pseudo-files we use to communicate configuration information.

4.1 Capture mechanism

The key to clean Linux integration is use of the LSM API. LSM implements two types of security hooks that are called 1) when a kernel object (e.g., `inodes`, `file`, `sockets` etc.) is allocated and 2) when a kernel object is accessed. The first allocates a security blob, containing information about the associated kernel object. The second is called when making access decisions during a system call (e.g., `read`, `write`, `mmap_file`).

There are two types of LSM: *major* and *minor* modules. *Major* modules use a security blob pointer (e.g., `cred->security`, `inode->i_security` etc.). Examples include `SELinux` [86] and `AppArmor` [14]. *Minor* security modules, such as `SMACK` [26] and `LoadPin` [27], do not use such pointers. Kernel version 4.1 allows a major module to be stacked with any number of minor modules. After Linux performs its normal access control checks, it calls LSM

hooks in a pre-defined order. If any LSM hook call fails, the system call immediately fails. We arrange for the CamFlow hook to be called last to avoid recording flows that are subsequently blocked by another LSM. CamFlow users the record of LSM events to build its provenance graph [78].

CamFlow needs a "provenance" blob pointer to store provenance information alongside kernel objects, so CamFlow is a major security module. Using CamFlow in conjunction with another major module, e.g., SELinux, requires that we modify some kernel data structures, e.g., `inode`, `cred`, to add an additional pointer. This is the one place we violate our "self-contained" philosophy. However, work is in progress [83] to allow stacking of *major* security modules (by modifying how security blobs are handled).

CamFlow hooks: 1) allocate a provenance-blob, if needed 2) consider filter/target constraints (see § 5), and 3) record provenance in the `relayfs` buffer. The user space provenance service then reads the provenance entries see § 4.4).

4.2 Cross-host Provenance Capture

CamFlow captures incoming packets through the `socket_sock_-rcv_skb` LSM hook. No equivalent hook exists for outgoing packets, so we implement a `NetFilter` hook to capture their provenance.

CamFlow represents a packet as an entity in the provenance graph. We construct packet identifiers from the immutable elements of the IP packet and immutable elements of the protocol (e.g., TCP or UDP). For example, a TCP packet's identifier is built from: IP packet ID, sender IP, receiver IP, sender port, receiver port, protocol, and TCP sequence number. We also record additional metadata such as the payload length. In most scenarios, the sending and receiving machines can match packets using this information. In other scenarios (e.g., NAT), some post-processing may be required to match packets (e.g., content match, temporal relationship, approximative packet length); the details of these techniques are beyond the scope of this paper. Additionally, CamFlow can optionally capture payloads.
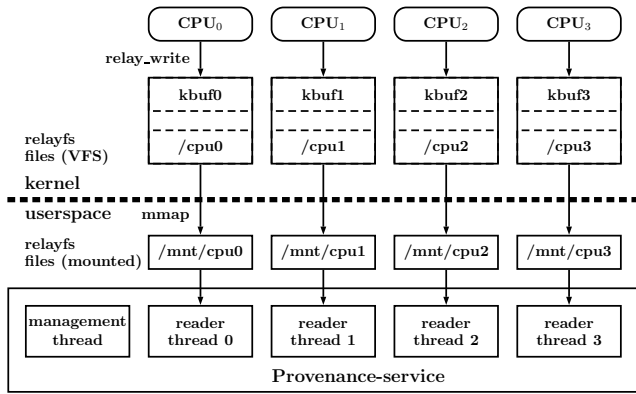
Figure 5: Transferring provenance data to user space through `relayfs`.

4.3 Hardware Root of Trust

In some circumstances it is necessary to be able to vouch for the integrity of provenance data and the provenance capture mechanism, e.g., when presented as evidence in a court of law. The software may not always be under the control of a trusted entity, and hardware-based trust certification will be required. Our architecture can support this by using Trusted Platform Module (TPM) [63] or virtual TPM [16] technologies that can be used to measure the integrity of a system. This integrity can be verified through Remote Attestation [52].

We need to assume that sufficient technical and non-technical measures are taken to ensure that the hardware has not been tampered with (e.g., see CISCO vs NSA [80]). We assume that the integrity of the low-level software stack is recorded and monitored at boot time through a Core Root of Trust Measurement (CRTM). The low-level software stack here includes: BIOS, boot loader code and configuration, options ROM, host platform configuration, etc. We assume that this integrity measurement is kept safe and cannot be tampered with (one of the features provided by TPM). Further, system integrity needs to be guaranteed at run time.

In the Linux world such the Integrity Measurement Architecture (IMA) [50, 52, 81] provies integrity guarantees and remote attestation. The CRTM measures hardware and kernel integrity. IMA is initialized before kernel modules are loaded and before any application is executed. IMA measures the integrity of kernel modules, applications executed by root, or libraries. Each binary is hashed to create an evidence trail. The integrity of this record is guaranteed by a cumulative hash, which is stored in TPM's Platform Configuration Registers. During remote attestation, this evidence trail is verified against the known good value. While not strictly part of CamFlow, use of IMA and remote attestation should be considered by cloud providers.

4.4 User Space Interface

While provenance capture must happen in the kernel, we relegate storage and processing to user space. CamFlow uses `relayfs` [95] to efficiently transfer data from kernel to user space as illustrated in Fig. 5. The CamFlow kernel module writes provenance records

into a ring buffer that is mapped to a pseudofile (this is handled by `relayfs`) that can be `read` or `mmapped` from user space. Before file system initialization, CamFlow records provenance into a "boot" buffer that is copied to `relayfs` as soon as it is initialized.

Once the file system is ready, `systemd` starts up a provenance service in user space. The service reads from the file that is mapped to the kernel buffer, and the `relayfs` framework deals with "freeing" space in the kernel ring buffer. The provenance service itself is marked as *opaque* to avoid the provenance service looping infinitely; the provenance capture mechanism ignores opaque entries, which do not appear in the captured data (see § 5).

We provide a user space library that deals with handling the `relayfs` files and multi-threading. The library also provides functions to serialize the efficient kernel binary encoding into PROV-JSON. We then stream the provenance data, in either binary or JSON format, to applications, as shown in Fig. 1. CamFlow can use any messaging middleware such as MQTT [10] or Apache Flume [46] to stream the provenance data.

4.5 Application-level Provenance

So far we have discussed how CamFlow records information flows through system calls, in terms of kernel objects. Other systems (e.g., database systems, workflow systems) record provenance in terms of different objects, e.g., tuples in a database system [19] or stages of processing in a workflow system [6, 28]. Muniswamy-Reddy et al. [65] demonstrated that many scenarios require associating objects from these different layers of abstraction to satisfy application requirements.

Consider scientific software reading several input files to generate several figures as output files. Application-level provenance is oblivious to files altered by other applications and can only provide an incomplete view of the system. On the other hand, system-level provenance, while providing a complete view of the system, will expose only coarse grained dependencies (i.e., every output will depend on all inputs read so far). Combining application- and system-level provenance allows us to describe fine-grained dependencies between inputs and outputs.

CamFlow provides an API for provenance-aware applications to disclose provenance to support the integration of application and system provenance. The API allows us to associate application provenance with system objects as long as a system level descriptor (e.g., a file descriptor) to that object is available to the application. CamFlow guarantees that such associations do not produce cycles, however, it is the responsibility of the application level provenance collection to ensure that there are no cycles within its subgraph. Further, CamFlow's underlying infrastructure manages identifiers and guarantees their uniqueness, system wide.

Application-level provenance associated with a kernel object is attached to a specific version of that object in the provenance graph. It is sometimes useful to view application-disclosed provenance as a subgraph contained within *activity* nodes, describing the internal working of that particular *activity*.

Modifying existing applications to be provenance-aware is not a trivial task [57, 65]. Ghoshal et al. [38] demonstrated that existing application log files can be transformed into a provenance graph. We provide the infrastructure for application to apply such approach.
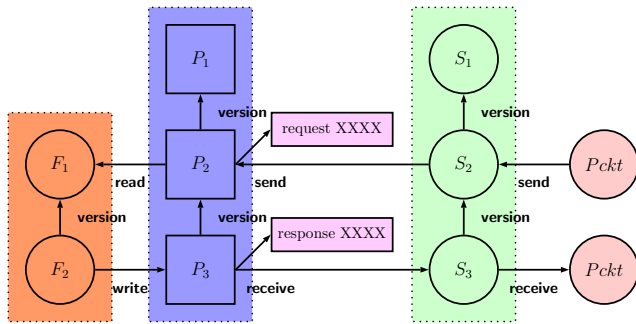
Figure 6: CamFlow-Provenance presented in Fig. 3 annotated with application logs.

We limit this approach to applications that already generate log files (e.g., `httpd`, `Postgres`). Placing these logs in a special pseudo-file automatically incorporates their log records into CamFlow's provenance graphs. For example, replacing `Error-Log "/var/log/httpd-error.log"` with `ErrorLog "/sys/kernel/security/provenance/log"` in the `httpd` configuration file enables CamFlow to incorporate web server operations. CamFlow creates an entity for each log record and attaches it to the appropriate version of activity that created it. The log record entities contain the text of the log message, thereby providing annotations for the provenance graph. This helps make the graph more human-readable. If desired, provenance applications can parse these log entries for further processing or interpretation.

Fig. 6 shows a log integrated into the provenance graph. In this example, the log entries might help a human user or application understand that the information exchanged with the socket corresponds to a client request/response interaction. In § 7.4, we use this feature to insert httpd logs into a system provenance graph.

5 TAILORING PROVENANCE CAPTURE

A major concern about whole-system provenance capture is that the size of the provenance can ultimately dwarf the "primary" data. Further, provenance application execution time is often a function of the size of the provenance graph. One approach is to limit provenance capture to "objects of interest", based on a particular security policy [12, 75, 76]. While recording interactions only between security-sensitive objects may make sense in some contexts, in others, for example, a scientific workflow, it probably does not make sense. We devised a more complex capture policy that allows end users to define the scope of capture based on their precise needs, greatly reducing storage overhead. Users choose between whole-system provenance and selective provenance. Within selective provenance, they can specify filters on nodes and edges, specific programs or directories to capture, whether capture should propagate from a specified process, and what network activity should be captured. Users specify these capture policies either through a command line interface or in a policy configuration file, like the one shown in Listing 3.

In whole-system provenance mode, CamFlow captures all objects that have not been declared opaque (line 7). In selective mode (line 5),

```
1  [provenance]
2  ;unique identifier for the machine, use hostid if set to 0
3  machine_id=0
4  ;enable provenance capture
5  enabled=true
6  ;record provenance of all kernel object
7  all=false
8  ;set opaque file
9  opaque=/usr/bin/ps
10 ;set tracked file
11 ;track=/home/thomas/test.o
12 ;propagate=/home/thomas/test.o
13 node_filter=directory
14 node_filter=inode_unknown
15 node_filter=char
16 relation_filter=sh_read
17 relation_filter=sh_write
18 propagate_node_filter=directory
19 propagate_node_filter=char
20 propagate_node_filter=inode_unknown
21
22 [ipv4−egress]
23 ;propagate=0.0.0.0/0:80
24 ;propagate=0.0.0.0/0:404
25 ;record exchanged with local server
26 ;record=127.0.0.1/32:80
27
28 [user]
29 ;track=thomas
30
31 [group]
32 ;propagate=docker
```

Listing 3: Capture policy example.

CamFlow captures the provenance of only non-opaque specified targets; line 9, `track=/home/thomas/myexperiment.o`, tells CamFlow to track any information flow to/from the specified file and any process resulting from the execution of programs built from it. Line 10 tells CamFlow to track any object that interacts with the file/process and any object they may subsequently interact with. Similarly, line 27, `propagate=0.0.0.0/0:0`, indicates that we want to track any information flow to/from a listening socket on the machine.

We can also specify opaque or tracked objects based on criteria, such as: 1) pathname (line 9), 2) network address (line 27), 3) LSM security context[2] (not shown), 4) control group,[3] (not shown), and 5) user and group ID. We frequently expand this list based on feedback from users. Fig. 7 illustrates how CamFlow implements the capture policy for the `open` system call.

The mechanism to *mark* targets varies depending on the targeted attribute. For example, exact path matches are handled through security extended attributes, while network interface based policies are handled in `connect` and `bind` LSM hooks. The policy engine does not reevaluate the policy for every event, but rather verifies the mark associated with kernel objects and the provenance data structure. Some policies (e.g., user id) are regularly reevaluated, for example, on `setuid` events.

---

[2]These are dependent on the major LSM running on a machine. The security contexts defined by SELinux or AppArmor are different. Defining capture policy based on these, therefore requires an understanding of the MAC mechanism in place.
[3]Control groups or `cgroups` are the mechanism used in Linux to control, for example, the resource access of containers.
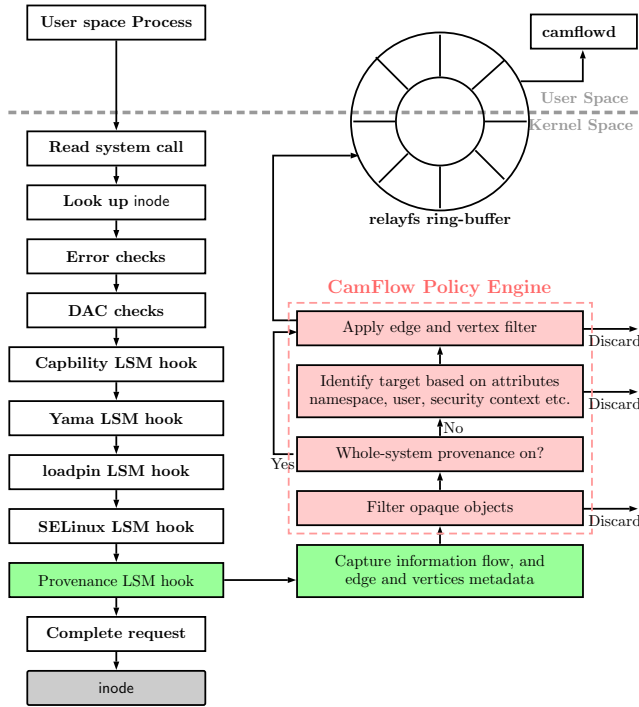
Figure 7: Executing an `open` system call. In green the capture mechanism described in § 4, in pink the provenance tailoring mechanism described in § 5.

Excluding edges or node types from the graph: Line 12 (14) illustrates how to exclude specific node (edge) types. In this case, line 12 tells CamFlow not to capture operations on directories, while line 14 says to ignore read permission checks. This is the last stage in an event capture, as illustrated in Fig. 7. For every edge recorded, CamFlow compares the types of its endpoints to that specified in the filter, discarding any edge that produced a match.

Propagating tracking: While targeting a single individual or group of objects may be interesting, in many cases the user wants to track all information emanating from a given source. The `propagate` setting tells CamFlow to track interactions with any object that received information from a tracked object; this automatically tracks the receiving object in a manner similar taint tracking [32] works. It is also possible to define types of relations or nodes through which tracking should not propagate (e.g., `propagate_node_-filter=directory`). Filtered nodes or relation types do not propagate tracking. For example, when tracking content dissemination (see § 7.3) it is possible to set the tracking propagation to never be applied to `directories`, or not to occur on permissions-checking operations (e.g., when opening a file), but only on explicit `read` and `write` operations.

Taint tracking: Tracking propagation can be coupled with tainting (with a taint being an arbitrary 64 bit integer), following the same policy-specified restrictions. Kernel objects can hold an arbitrary number of taints. This is useful when provenance is used for policy enforcement, requiring a quick decision to be made upon accessing an object (see § 7.3).

Note that "tailoring" provenance capture means that the provenance is no longer guaranteed to be *complete*. Indeed, the *raison d'être* for tailoring is to exclude part of the provenance from the record. Thus, it is imperative that a user understand the *minimum* and *sufficient* information required for a particular application to be *correct*.

We can, for example, make the distinction between *data provenance* [20] and *information flow audit* [75]. The two are similar, but differ in their objectives. The first is concerned with where data comes from; that is, the *integrity* of the data (e.g., verifying properties of data used for a scientific results). The second is concerned with where the data have gone; that is, the *confidentiality* of the data (e.g., monitoring usage of personal data in a system). In terms of *completeness*, assuming an object or objects of interest, this means that for data provenance (integrity), all information flows **to** the object must be recorded; and for infomation flow audit (confidentiality), all information flows **from** the object must be recorded. The tracking propagation mechanism, for example, is appropriate to handle *confidentiality*, but not necessarily *integrity*. It is important to define the *objective* of the capture and the *adversary* (if any) when tailoring the capture.

6 EVALUATION

The goal of our evaluation is to answer the following questions: 1) How does CamFlow improve maintainability relative to prior work? 2) What is the impact of CamFlow on PaaS application performance? 3) How much does selective provenance capture reduce the amount of data generated? We follow this quantitative evaluation with more qualitative evaluation of usability in § 7.

6.1 Maintainability

Provenance system	Headers	Codes	Total	LoC
(kernel version)	Number of files			
PASS (v2.6.27.46)	18	69	87	5100
LPM (v2.6.32 RHEL 6.4.357)	13	61	74	2294
CamFlow (v4.9.5)	8	1	9	2428

Table 1: Number of files and lines of code (LoC) modified in various whole-system provenance solutions. The LoC results for PASS and LPM are as reported in [66] and [13] respectively. The number of files modified measurement is based on the last source code available from the PASS and Hi-Fi/LPM teams and for CamFlow v0.2.1.

It is difficult to objectively measure the maintainability of a code base. PASS was one of the first whole-system provenance capture mechanisms, but it required massive changes to the Linux kernel, and it was unmaintainable, almost from day one. We never simply upgraded PASS across Linux kernel minor versions, let alone major versions, because it was such a massive undertaking. It took us over two years and several dozen person months to develop the second version of PASS [65], which included a transition from Linux 2.4.29 to 2.6.23, i.e., two minor Linux revisions. Our estimate is that the effort was divided in roughly three equal parts: re-architecting how we interacted with the kernel, adding new features, and changing kernel releases. It should be apparent that maintaining a self-contained

provenance capture module will be significantly easier than maintaining PASS, and the ease with which we have adapted to new Linux releases supports this. Indeed, an explicit requirement of the LSM framework [62] is to limit the impact of new security features on the kernel architecture, thereby improving the maintanability of the Linux kernel. CamFlow selected the LSM framework for precisely this reason. Further as LSM is a major security feature of the Linux kernel to our knowledge most Linux distribution ship with an LSM (e.g., SELinux for Android >=4.3), it is practically guaranteed that future releases will continue to support the framework with minimal modification.

Table 1 provides some metrics to quantify the pervasiveness required for PASS, LPM, and CamFlow. We calculated the numbers using `diff` between "vanilla" Linux and provenance-enabled Linux for the same version, ignoring newly added files.

Kernel version	Code	Header	CamFlow	Total	Conflict
	In number of files modified			In number of lines modified	
4.9.6 → 4.10-rc6	0	1	0	234	0
4.9.5 → 4.9.6	0	0	0	0	0
4.9.4 → 4.9.5	0	0	0	0	0
4.9.2 → 4.9.4	0	0	0	0	0
4.9 → 4.9.2	0	0	0	0	0
4.4.38 → 4.9	1	4	3	1194 (17)	8
4.4.36 → 4.4.38	0	0	0	0	0
4.4.35 → 4.4.36	0	0	0	0	0
4.4.34 → 4.4.35	0	0	1	2 (2)	0
4.4.32 → 4.4.34	0	0	0	0	0

Table 2: Modification required when porting CamFlow to new kernel versions. The number of lines modified includes comments. The numbers in parentheses are the number of lines of CamFlow-specific code changed.

Neither PASS nor LPM were ever ported to newer kernel releases, even though both groups would have liked to do so. In contrast, we have already maintained CamFlow across multiple Linux kernel major revisions with only a few hours of effort. The authors of PASS and LPM state that transitioning to newer kernel releases would take many man months. Table 2 quantifies the CamFlow porting effort. We obtained these results by going through the updates in our git repository from November 21, 2016 to February 2, 2017. We considered only changes relating to version updates and ignored feature development. The most significant upgrade over this period was from version 4.4.38 to 4.9. However, we modified 8 lines of code in `/security/security.c` relating to changes in the LSM framework. Most of these patches were applied without any conflict. We continue to maintain CamFlow, see Appendix A for the most recent supported version and a full commit history. (Neither PASS nor LPM have any such commit history available.)

6.2 Performance

We next discuss how CamFlow affects system performance. We selected standard benchmarks that can be easily reproduced and provide meaningful references to prior work. We ran the benchmarks on a bare metal machine with 6GiB of RAM and an Intel i7-4510U CPU. (We use a bare metal machine rather than a cloud platform, such as EC2, to reduce uncontrolled variables.) See Appendix A for details on obtaining the code, reproducing these results and accessing

Figure 8: System call overhead decomposition.

the raw results presented here, following suggestions from Collberg et. al [24].

We run each test on three different kernel configurations. The *vanilla* kernel configuration is our baseline and corresponds to an unmodified Linux kernel. The *whole* configuration corresponds to CamFlow recording whole-system provenance. The *selective* configuration corresponds to CamFlow with whole-system provenance off, but selective capture on. We use CamFlow v0.2.1 [71] with Linux 4.9.5. Comparing the CamFlow configurations to the baseline reveals the overhead imposed by CamFlow's provenance capture mechanism.

Test Type	vanilla	whole	overhead	selective	overhead
Process tests, times in μs, smaller is better					
NULL call	0.07	0.06	0%	0.05	0%
NULL I/O	0.09	0.14	56%	0.14	56%
stat	0.39	0.78	100%	0.50	28%
open/close file	0.88	1.59	80%	1.04	18%
signal install	0.10	0.10	0%	0.10	0%
signal handle	0.66	0.67	2%	0.66	0%
fork process	110	116	6%	112	2%
exec process	287	296	3%	289	<1%
shell process	730	771	6%	740	1%
File and memory latencies in μs, smaller is better					
file create (0k)	5.05	5.32	5%	5.12	1%
file delete (0k)	3.42	3.78	11%	3.79	11%
file create (10k)	9.81	11.41	16%	10.9	11%
file delete (10k)	5.70	6.12	7%	6.08	6%

Table 3: LMbench measurements.

Microbenchmarks: We used LMbench [60] to benchmark the impact of CamFlow's provenance capture on system call performance. Table 3 presents a subset of the results. Due to space constraints, we selected the subset of greatest relevance; the complete results are available online.

The performance overhead can be decomposed into two parts as illustrated in Fig. 8. The first part is present in the *selective* and *whole* results. This part concerns the allocation of the provenance blob associated with kernel objects (see § 4) and the application of the policies described in § 5. The second part is present in the *whole* overhead only and corresponds to the recording of the provenance.

It is important to note that a system call does not necessarily correspond to a single event in the provenance graph. On a file open, for example, both `read_perm` and `write_perm` provenance events may be generated as appropriate for the directory and subdirectories in the path and for the file itself, in addition to an event for the `open` on the file. Further, if the file did not exist prior to the `open`, CamFlow generates a `create` event. Tailoring the types of provenance events that must be recorded (see § 5) can further

improve performance. Additionally, assuming a relatively constant time for provenance data allocation and copy into the relay buffer, the overhead appears proportionally large due to the short execution time of most system calls. For example, `stat` is fast, but must record `perm_read`, `read` and `get_attr` provenance relations, producing large *relative* overhead, which in *absolute* terms is smaller than that for `exec` ($0.39\mu s$ vs $9\mu s$ for whole provenance), which requires several provenance object allocations and a larger number of relations.

Test Type	vanilla	whole	overhead	selective	overhead
Execution time in seconds, smaller is better					
unpack	11.16	11.36	2%	11.24	<1%
build	433	442	2%	429	0%
4kB to 1MB file, 10 subdirectories,					
4k5 simultaneous transactions, 1M5 transactions					
postmark	71	79	11%	75s	6%

Table 4: Standard provenance-system macrobenchmark results.

Macrobenchmarks: We present two sets of macrobenchmarks that provide context for the overheads measured in the microbenchmarks. First, we used the Phoronix test suite [55] to illustrate single machine performance, running kernel *unpack* and *build* operations. Second, we ran the Postmark benchmark [51], simulating the operation of an email server. These are the common benchmarks used in the system provenance literature. Next, we ran a collection of benchmarks frequently used in cloud environments, also available via the Phoronix test suite. Table 4 shows the results of the single system benchmarks, and Table 5 shows the results for the cloud-based benchmarks.

Test Type	off	whole	overhead
Request/Operation per second (the higher the better)			
apache	8235	7269	12%
redis (LPOP)	1442627	1120935	22%
redis (SADD)	1144405	1068877	7%
redis (LPUSH)	998077	961849	4%
redis (GET)	1386329	1219830	12%
redis (SET)	1053978	1012158	4%
memcache (ADD)	18297	16658	9%
memcache (GET)	37826	32548	14%
memcache (SET)	17762	16825	5%
memcache (APPEND)	19393	17172	11%
memcache (DELETE)	38245	32517	15%
php	240889	239069	<1%
Execution time (ms) (the lower the better)			
pybench	3515	3525	<1%

Table 5: Extended macrobenchmark results.

The single system benchmarks (Table 4) show that CamFlow adds minimal overhead for the kernel unpack and build benchmarks and as much as 11% overhead to Postmark for whole-system capture. These results compare favorably to prior systems and the option to use selective capture provides a better performing solution.

We ran two configurations for the cloud-based benchmark: *off* includes CamFlow in the kernel build but does not capture any data, representing tenants who do not use the provenance capture feature, and *whole* runs in whole-system capture mode. This time, the overheads are larger, with a maximum of 22% for redis LPOP.

Discussion: CamFlow's performance is the same order of magnitude as that reported by PASS [66], Hi-Fi [78], its successor LPM [13], and SPADE [36], and slightly better in most cases. However, these results are difficult to compare—the variation in experimental setup is significant e.g., from a 500MHz Intel Pentium 3 with 768 MiB RAM for PASS to a dual Intel Xeon CPU machine for LPM, or e.g., kernel version from 2.4.29 for PASS to 4.9.5 for CamFlow. Further the provenance "coverage" tends to become more extensive in each successive publication. Comparing to the latest published work—LPM—CamFlow includes additional information such as `iattr` and `xattr`, `security context`, `control group`, `overlayfs` etc. While this provides a more complete picture, it also increases the amount of provenance recorded, which produces more overhead.

6.3 Generated Data Volume

CamFlow is a capture mechanism, so we do not attempt to evaluate provenance storage strategy, but rather the impact of selective capture on the amount of data generated. As prior systems [13, 36, 66, 78] have shown, storage and the accompanying query time increase over time. We discuss an approach to address increasing query time for some scenarios in § 7.3.

We currently write provenance in a bloated, but standard format (PROV-JSON). Colleagues working with CamFlow have achieved a storage size of 8% of the equivalent PROV-JSON graph size through compression techniques [89]. Others have reported on alternative ways to reduce storage requirements through compression [22, 93, 94], but such concerns are orthogonal to the work presented here. Regardless of the storage techniques employed, the size of the graph grows over time, and this makes query time grow proportionally.

The CamFlow capture mechanism addresses storage issues in the following ways: 1) it gives the storage system great flexibility through a clear separation of concerns; and 2) it allows for configurable capture, limiting the volume of data. The separation of concerns is application specific and out of scope, however, previous work by Moyer et al. [64] discusses handling provenance storage in a cloud environment. In the rest of this section we evaluate the impact of selective capture.

We selected a simple workload to ease reproducibility; we record the provenance generated by:
`wget www.google.com`
We have three experimental setups: 1) provenance off (baseline); 2) ignore directory nodes and permission edges (`perm_read`, `perm_write` and `perm_exec`) (selective); and 3) record everything (whole). Packet payloads are *not* recorded in any of these setups. Note that there are 39 edge types and 24 node types in the current implementation. The data recorded when the provenance is off corresponds to the machine/boot description (<1kB). The output rate is 12.45kB/iteration for *whole* and 9.98kB/iteration for *selective*; tailoring capture reduces the data rate by 20%. This illustrates the trade-off between between completeness and utility (see discussion

in § 5). In scenarios where, for example, only a set of applications or the actions of a user are of interest, the quantity of data captured can be several orders of magnitude smaller relative to PASS or LPM. As discussed in § 5, it is possible express in detail what part of the system is/is not of interest. This trade-off needs to be viewed in relation to the provenance application; we discuss a concrete example in § 7.3.

6.4 Completeness and expressiveness

CamFlow extends the functionality of previous provenance systems, while retaining the same features. Notably, we introduce a new completeness trade-off against performance, storage, and I/O bandwidth. In other words, we allow users to decide between capturing the entirety of supported system events (as PASS or LPM did) and capturing a subset of those events based of some particular well defined needs (see § 5). Further, this same feature has proved helpful to new users when developing provenance applications (see § 7). The tailoring feature has been used to help explain system behavior and to help new developers get started. A typical workflow with CamFlow is: 1) start capturing a single or small group of processes and their interactions; 2) develop a small prototype provenance application; and 3) move to an architecture able to handle scale issues introduced by whole-system provenance. This has proved extremely beneficial as the accumulation rate and amount of provenance data can be daunting. We and our users believe this to be important for increasing provenance adoption.

A remaining challenge is to provide a full comparison of CamFlow against alternative implementations: i.e., *is the graph as complete and expressive as when using an alternative approach?* To our knowledge, no study of such a nature has been completed. Our experience tells us that CamFlow captures as much or more information compared to PASS or LPM, but we have not performed any formal evaluation. Indeed, quantitatively measuring provenance completeness or expressivity are complex, ongoing topics of research in the provenance community [21]. Chan et al. [21] propose running a system-call benchmark and comparing the "quality" of the provenance captured. However, one could argue that "quality" is a hard metric to evaluate. An alternative approach is more formal and, as discussed in § 4.1, relies on the guarantees provided by LSM hooks that are present on the path between any kernel object and a process. However, recent research has demonstrated that this coverage may not capture every information flow [37], as the LSM framework was originally conceived to implement MAC policy. We derived from this publication [37] a patch to the LSM infrastructure [72]. This patch is used by most recent versions of CamFlow (>=0.3.3). In addition to demonstrating that provenance is generated for every event, it must also be shown that the generated graph is "correct" and "well connected". An empirical or formal demonstration of the completeness of the provenance captured is beyond the scope of this paper.

7 EXAMPLE APPLICATIONS

We proposed that PaaS providers offer provenance as a service to their tenants. In § 3, we discussed how our model expands on previous provenance capture implementations, notably by recording information relevant to Docker containers. In § 4, we showed how provenance can be captured and streamed in OS or distributed systems. In § 5, we showed how the captured provenance can be tailored to meet the tenant application requirements. Finally, in § 6 we showed that this can be achieved with tolerable performance impact. We now discuss, through examples, how cloud tenants can subscribe to the provenance data stream and develop provenance applications. We explore four such applications: two from from ongoing work at Harvard (§ 7.1 and § 7.2), and two where we implement examples from prior work, taking advantage of CamFlow's unique features (§ 7.3 and § 7.4).

7.1 Demonstrable Compliance

The manipulation of personal data in widely distributed systems, especially the cloud, has become a subject of public concern. As a result, legislators around the world are passing laws to address issues around manipulation and privacy of personal data. In Europe, data protection laws [2, 4] regulate and control all flows of personal data, in essence, information identifiable to individuals. These flows must be for specific legitimate purposes, with various safeguards for individuals and responsibilities on those who hold, manage and operate on personal data. In other jurisdictions, most notably the United States, which does not have such omnibus data protection laws, there are sector-specific restrictions on personal data. These include areas such as health [3] and finance [5], as well as general Fair Information Practice Principles (FIPP) [25], which embody principles such as notice, choice, access, accuracy, data minimization, security, and accountability. Further, there is pressure for the involved actors to provide transparency in how they handle data and to demonstrate compliance with the requirements.

In recent work [74], we developed an application that used CamFlow provenance to continuously monitor compliance with such regulations. information flow constraints that translate into properties of paths in a provenance graph can express many of these regulations. An *auditor* can subscribe to a (distributed) system's provenance stream (as described in § 4) and verify that the stream respects the specified constraints, using a label propagation mechanism. Developers write these constraints using a framework similar to familiar graph processing tools such as GraphChi [53] or GraphX [41]. The example application uses structural and semantic properties of provenance graphs to reduce overhead, claiming CPU overhead of about 5% and storage of only a few hundred megabytes of memory. In parallel, the system retains the complete provenance record as forensic evidence, using cloud-scale storage such as Apache Accumulo [64]. When the *auditor* detects a regulation violation, users can query and analyze the forensic provenance to identify the cause of the violation and take action, for example, by fixing the system or notifying affected users as mandated by HIPAA [3].

7.2 Intrusion Detection

The FRAPpuccino fault/intrusion detection system uses the capture mechanism presented here to identify errant or malicious instances of a PaaS application [44, 45]. An earlier system, pH [87, 88], recorded system call patterns of security-sensitive applications to detect abnormal behavior. We hypothesised that provenance data would be a better source of information, leading to more accurate results. We

experimented with reported real world vulnerabilities, and preliminary results indicate that we are able to detect attacks with higher accuracy.

The system consists of two stages of operation: 1) a training stage where we build a per-application behavior model and 2) a deployment stage where we monitor deployed applications. During the first stage, many instances of the target application run in parallel on the PaaS platform, and we assume most of them behave correctly. We use the provenance graph generated to build a model of normal behavior. In deployment, we analyze the provenance stream in sliding windows, comparing the current execution to the model generated during the training stage. If the stream does not fit the model, we determine if there is a real intrusion or fault. In the case of a false positive, we retrain the model incorporating the new data. In the case of a true positive, we can analyze the forensic provenance to identify the root cause.

The prototype successfully detected known vulnerabilities, such as a `wget` arbitrary file upload [40] that can be leveraged for privilege escalation, a tcpdump crash [84] through malicious packets, and an out of memory exception in a Ruby server that breaks garbage collection [35].

This is ongoing work. We plan to analyze streamed provenance data on Data-Intensive Scalable Computing systems such as Apache Spark, to expedite the process and make our intrusion detection system scalable. We will also extend the prototype to include the retraining process, to further improve accuracy in monitoring complex applications. Work remains to evaluate the effectiveness of the approach in discovering new vulnerabilities.

7.3 Data Loss Prevention

Bates et al. describe how to leverage whole-system provenance capture to build Provenance-Based Data Loss Prevention (PB-DLP) [13]. PB-DLP issues a provenance query that follows well-defined edge types to determine data dependencies on data before it leaves the system. If the dependencies include any sensitive information, PB-DLP aborts the transfer. Bates et al. acknowledge that the provenance graph grows without bound, creating increasingly long query times.

Our first approach to this use case leveraged the knowledge that policy enforcement queries need only consider a set of well-defined edge types. As query time is dependent on the overall graph size, we tailor the capture policy, restricting capture to only those edges of the specified type(s). This significantly reduces the volume of data collected.

The second approach uses CamFlow's taint mechanism, introduced in § 5. Using taint propagation, we can replace a potentially costly graph query with a much less expensive taint check when data is about to leave the system. Propagation can be tuned taint to apply only to specific types of node and/or edge. As a result, we can produce the same behavior produced by the (expensive) query of Bates et al. [13] with a time-constant check. Further, we note that taint tracking has been proved effective in similar scenarios [32]. In conjunction with tracking taint using this method, we can also choose to store either whole or selective provenance as forensic evidence. This allows the use of "post-mortem" queries to explore the chain of events leading to the disclosure attempt. This approach

reduces computational overhead, while simultaneously providing forensic evidence.

7.4 Retrofitting existing applications

Bates et al. also retrofitted provenance into a classic three-tier web application [11]. They use LPM to capture whole-system provenance and modify the web server to relate HTML and database queries. A proxy between the web server and the database collects SQL queries, which are transformed into application-level provenance for the database. It is therefore possible to link an HTML response to database entries. Before responding to a query, the system issues a provenance query on the combined application and system data to check for leakage of sensitive data, which it then prevents. This architecture avoids having to modify the database, but still requires modifications to the server and adds an additional component (the proxy) into the system. With CamFlow we can achieve the same result without modifying the existing three-tier web application, as follows:

1) We track (and optionally propagate tracking from) both server and database executables. 2) We configure the server's `ErrorLog` as a provenance log file, as described in § 4.5 and set the `LogLevel` to collect sufficient information to relate an HTML request to an SQL query. 3) We capture network packets (e.g., by adding the line `record=DB_IP/32:DB_PORT` in the capture policy), which allows us to capture the SQL queries, without using an intermediate proxy.

8 RELATED WORK

Most provenance capture systems work at the application level, e.g. [8, 57, 59, 69] or at the workflow level [17, 29]. These systems provide an incomplete record of what happened, which renders some use cases impossible (see § 4.5). System level provenance provides information about the interactions *between* programs, which is the key to supporting more complex use cases. One of the first attempts at system-level provenance is the Lineage File System [82] that used system call interposition in a modified Linux Kernel, while a user space process read captured provenance out of a printk buffer and stored the data in a SQL database. PASS captures provenance at the Virtual File System (VFS) layers. PASSv1 [66] provides functionality to capture processes' I/O interactions, while PASSv2 [65] introduced a disclosed provenance API to allow the integration of provenance across semantic levels. The main issue for these systems was the difficulty of maintaining and extending provenance capture over time, since it was spread among various places in the kernel. CamFlow is a strictly self-contained implementation that uses standard kernel features and is extremely easy to extend and maintain.

The next generation of system-level provenance used the Linux Security Framework (LSM) to capture provenance. The LSM framework provides a guarantee of completeness [31, 34, 49] of the provenance captured; Hi-Fi [78] was the first such approach. Hi-Fi did not support security module stacking (i.e., SELinux or AppArmor cannot run alongside provenance capture), rendering the system it monitors vulnerable. Further, Hi-Fi did not provide any API for disclosed provenance. Linux Provenance Module (LPM) [13] addressed those limitations. LPM provides an API for disclosed provenance and

duplicates the LSM framework to allow the stacking of provenance and MAC policy.

CamFlow leverages recent advances of the LSM framework [30, 83] to allow stacking of LSMs without feature duplication, thereby further increasing maintainability and extensibility. A second objective of LPM was to provide stronger integrity guarantees: leveraging TPM and the Linux Integrity Measurement Architecture [50, 81] and using cryptographic techniques to protect packet-level provenance taint. We can do the former, while the latter tends to create compatibility issues. We believe in a clearer separation of concerns, leaving packet integrity guarantees to mechanisms such as IPSec [33].

An alternative implementation of whole-system provenance is interposition between system calls and libraries in user space, as in OPUS [9]. An argument in favour of such systems is that modifications to existing libraries are more likely to be adopted than modifications to the kernel. However, for this approach to work, **all** applications in the system need to be built against, or dynamically linked to, provenance-aware libraries, replacing existing libraries. Further, the guarantees of completeness provided are much weaker than those provided by an LSM-based implementation. SPADE [36] mixes this approach, with the Linux Audit infractustructure; the LSM-based approach to provenance provides a more detailed record and stronger completeness guarantees.

Another system-call based approach that uses both logging and taint tracking is ProTracer [58]. ProTracer relies on kernel tracepoints to intercept system calls and capture provenance. The system relies on taint propagation when the events being traced are ephemeral (e.g., inter-process communication) and full provenance capture for events that are determined to be permanent. One concern with system call-based approaches is the lack of guarantees of completeness in the provenance record. Consider for example the `read` and `pread64` system calls. Both read from a file descriptor, the difference being that the latter can do so with an offset. A system needs to properly record both types of read system calls, and should probably ensure that the record for `pread64` is the same as the record of an `lseek` and subsequent `read` call. Ensuring such consistency increases the complexity of the capture (or post-processing). LSM-based approaches are guaranteed to capture every event that is deemed security-sensitive and focus on the objects being accessed, instead of the actions being carried out on those objects.

One of the main hurdles of system-level provenance capture is the sheer amount of data generated [13, 64]. One approach to this issue is to improve provenance ingest to storage [64] and provenance query [90]. A second approach is to reduce the amount of provenance data captured. This reduction might be based on security policies [12, 75], limiting capture to only those elements considered *important* or *sensitive*. This is of limited interest in contexts such as scientific workflows, where data of interest may not be associated with any security policy. We address these issues by providing administrators and system designers with selective capture in a way consistent with the provenance data-model.

Using system-level provenance in a cloud computing context has been proposed in the past [56, 67, 68, 96]. To our knowledge, the work presented here is the first open-source maintained implementation with demonstrated use cases (§ 7). The main focus of our work has been to develop a practical system that can be used and

expanded upon by the community. There is application-level provenance for a cloud computational framework such as MapReduce [6] or Spark [43, 48]. We see these as complementary; they could be incorporated in CamFlow provenance capture through the mechanism discussed in § 4.4.

9 CONCLUSION

CamFlow is a flexible, efficient and easy to use provenance capture mechanism for Linux. Linux is widely used in cloud service offerings and our modular architecture, using LSM, means that data provenance can be included without difficulty as part of a cloud OS in PaaS clouds.

We take advantage of developments in Linux kernel architecture to create a modular, easily maintainable and deployable provenance capture mechanism. The ability to finely tune the provenance being captured to meet each application's requirements helps to create a scalable system. Our standards-compliant approach further enhances the broad applicability and ease of adoption of CamFlow. Further, we demonstrated the low impact on performance and the wide applicability of our approach.

ACKNOWLEDGMENTS

REFERENCES

[1] [n. d.]. Docker. ([n. d.]). https://www.docker.com.
[2] [n. d.]. General Data Protection Regulation. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2016.119.01.0001.01.ENG&toc=OJ:L:2016:119:TOC. ([n. d.]).
[3] [n. d.]. Health Insurance Portability and Accountability Act. https://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm. ([n. d.]).
[4] [n. d.]. OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data. ([n. d.]). http://www.oecd.org/internet/ieconomy/oecdguidelinesontheprotectionofprivacyandtransborderflowsofpersonaldata.htm.
[5] [n. d.]. The Sarbanes-Oxley Act of 2002. http://www.soxlaw.com/. ([n. d.]).
[6] Sherif Akoush, Ripduman Sohan, and Andy Hopper. 2013. Hadoopprov: Towards provenance as a first class citizen in mapreduce. In *Workshop on the Theory and Practice of Provenance*. USENIX.
[7] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. 2016. Correct Audit Logging: Theory and Practice. In *International Conference on Principles of Security and Trust (POST'16)*. Springer.
[8] Elaine Angelino, Daniel Yamins, and Margo Seltzer. 2010. StarFlow: A script-centric data analysis environment. In *International Provenance and Annotation Workshop*. Springer, 236–250.
[9] Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. 2013. OPUS: A Lightweight System for Observational Provenance in User Space.. In *Workshop on the Theory and Practice of Provenance*. USENIX.
[10] Andrew Banks and Rahul Gupta. 2014. MQTT Version 3.1. 1. *OASIS Standard* (2014).
[11] Adam Bates, Kevin Butler, Alin Dobra, Brad Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. 2016. Retrofitting Applications with Provenance-Based Security Monitoring. *arXiv preprint arXiv:1609.00266* (2016).
[12] Adam Bates, Kevin RB Butler, and Thomas Moyer. 2015. Take only what you need: leveraging mandatory access control policy to reduce provenance storage costs. In *Workshop on Theory and Practice of Provenance*. USENIX, 7–7.
[13] Adam Bates, Dave Tian, Kevin Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *Security Symposium*. USENIX.
[14] Mick Bauer. 2006. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal* 2006, 148 (2006), 13.
[15] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker,

Simon Miles, James Myers, Satya Sahoo, Luc Moreau, and Paolo et al. Missier. 2013. *Prov-DM: The PROV Data Model*. Technical Report. World Wide Web Consortium (W3C). https://www.w3.org/TR/prov-dm/.

[16] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. 2006. vTPM: Virtualizing the Trusted Platform Module. In *Security Symposium*. USENIX, 305–320.

[17] Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. 2012. Declarative rules for inferring fine-grained data provenance from scientific workflow execution traces. In *International Provenance and Annotation Workshop*. Springer, 82–96.

[18] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *International Conference on Database Theory*. Springer, 316–330.

[19] Peter Buneman and Wang-Chiew Tan. 2007. Provenance in databases. In *SIGMOD international conference on Management of data*. ACM, 1171–1173.

[20] Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Selter, and Andy Hopper. 2014. A primer on provenance. *Commun. ACM* 57, 5 (2014), 52–60.

[21] Sheung Chi Chan, Ashish Gehani, James Cheney, Ripduman Sohan, and Hassaan Irshad. 2017. Expressiveness Benchmarking for System-Level Provenance. In *Workshop on the Theory and Practice of Provenance (TaPP 2017)*. USENIX.

[22] Adriane P Chapman, Hosagrahar V Jagadish, and Prakash Ramanan. 2008. Efficient provenance storage. In *International Conference on Management of Data*. ACM, 993–1006.

[23] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 115–128.

[24] Christian Collberg and Todd A Proebsting. 2016. Repeatability in computer systems research. *Commun. ACM* 59, 3 (2016), 62–69.

[25] Federal Trade Commission. 2007. Fair information practice principles. (2007).

[26] Jonathan Corbet. 2007. SMACK meets the One True Security Module. (2007). https://lwn.net/Articles/252562/.

[27] Jonathan Corbet. 2016. The LoadPin security module. (2016). https://lwn.net/Articles/682302/.

[28] Daniel Crawl, Jianwu Wang, and Ilkay Altintas. 2011. Provenance for mapreduce-based data-intensive workflows. In *Workshop on Workflows in Support of Large-scale Science*. ACM, 21–30.

[29] Susan B Davidson, Sarah Cohen Boulakia, Anat Eyal, Bertram Ludäscher, Timothy M McPhillips, Shawn Bowers, Manish Kumar Anand, and Juliana Freire. 2007. Provenance in Scientific Workflow Systems. *IEEE Data Eng. Bull.* 30, 4 (2007), 44–50.

[30] Jake Edge. 2015. Progress in security module stacking. *Linux Weekly News* (2015).

[31] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. 2002. Runtime verification of authorization hook placement for the Linux Security Modules framework. In *Conference on Computer and Communications Security*. ACM, 225–234.

[32] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[33] S Frankel and S Krishnan. 2011. *RFC 6071: IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. Technical Report.

[34] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. 2005. Automatic placement of authorization hooks in the Linux security modules framework. In *Conference on Computer and Communications Security*. ACM, 330–339.

[35] Alexey Gaziev. 2015. How Ruby 2.2 can cause an out-of-memory server crash. (2015). https://evilmartians.com/chronicles/ruby-2_2-oom.

[36] Ashish Gehani and Dawood Tariq. 2012. SPADE: support for provenance auditing in distributed environments. In *International Middleware Conference*. ACM/IFIP/USENIX, 101–120.

[37] Laurent Georget, Mathieu Jaume, Frédéric Tronel, Guillaume Piolle, and Valérie Viet Triem Tong. 2017. Verifying the reliability of operating system-level information flow control systems in linux. In *International Workshop on Formal Methods in Software Engineering (FormaliSE 2017)*. IEEE, 10–16.

[38] Devarshi Ghoshal and Beth Plale. 2013. Provenance from log files: a BigData problem. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 290–297.

[39] Jeremy Goecks, Anton Nekrutenko, and James Taylor. 2010. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology* 11, 8 (2010), R86.

[40] Dawid Golunski. 2016. Wget Arbitrary File Upload Vulnerability Exploit. (2016). https://legalhackers.com/advisories/Wget-Arbitrary-File-Upload-Vulnerability-Exploit.html.

[41] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI*, Vol. 14. 599–613.

[42] Mark Greenwood, CA Goble, Robert D Stevens, Jun Zhao, Matthew Addis, Darren Marvin, Luc Moreau, and Tom Oinn. 2003. Provenance of e-science experiments-experience from bioinformatics. In *Proceedings of UK e-Science All Hands Meeting 2003*. 223–226.

[43] Muhammad Ali Gulzar, Xueyuan Han, Matteo Interlandi, Shaghayegh Mardani, Sai Deep Tetali, TD Millstein, and Miryung Kim. 2016. Interactive debugging for big data analytics. In *Workshop on Hot Topics in Cloud Computing (HotCloud'16)*. USENIX, 20–21.

[44] Xueyuan Han. 2017. michael-hahn/frap: v1.1.1. (2017). DOI:10.5281/zenodo.571444, https://github.com/michael-hahn/frap.

[45] Xueyuan Han, Thomas Pasquier, Mark Goldstein, and Margo Seltzer. 2017. FRAPpuccino: Fault-detection through Runtime Analysis of Provenance. In *Workshop on Hot Topics in Cloud Computing (HotCloud'17)*. USENIX.

[46] Steve Hoffman. 2013. *Apache Flume: Distributed Log Collection for Hadoop*. Packt Publishing Ltd.

[47] Trung Dong Huynh, Michael Jewell, Amir Keshavarz, Danius Michaelides, Huanjia Yang, and Luc Moreau. 2013. *The PROV-JSON Serialization: a JSON Representation for the PROV Data Model*. Technical Report. World Wide Web Consortium (W3C). https://www.w3.org/Submission/prov-json/.

[48] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data provenance support in spark. *Proceedings of the VLDB Endowment* 9, 3 (2015), 216–227.

[49] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. 2004. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Transactions on Information and System Security (TISSEC)* 7, 2 (2004), 175–205.

[50] Trent Jaeger, Reiner Sailer, and Umesh Shankar. 2006. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*. ACM, 19–28.

[51] Jeffrey Katcher. 1997. *Postmark: A new file system benchmark*. Technical Report. Technical Report TR3022, Network Appliance.

[52] Chongkyung Kil, Emre Can Sezer, Ahmed M Azab, Peng Ning, and Xiaolan Zhang. 2009. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *Dependable Systems & Networks (DSN'09)*. IEEE, 115–124.

[53] Aapo Kyrola, Guy E Blelloch, Carlos Guestrin, et al. 2012. GraphChi: Large-Scale Graph Computation on Just a PC.. In *OSDI*, Vol. 12. 31–46.

[54] Butler W Lampson. 2004. Computer security in the real world. *Computer* 37, 6 (2004), 37–46.

[55] Michael Larabel and M Tippett. [n. d.]. Phoronix test suite. ([n. d.]). http://www.phoronix-test-suite.com.

[56] Brian Lee, Abir Awad, and Mirna Awad. 2015. Towards secure provenance in the cloud: a survey. In *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*. IEEE, 577–582.

[57] Barbara Lerner and Emery Boose. 2014. RDataTracker: collecting provenance in an interactive scripting environment. In *Workshop on the Theory and Practice of Provenance (TaPP 2014)*. USENIX.

[58] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://www.internetsociety.org/sites/default/files/blogs-media/protracer-towards-practical-provenance-tracing-alternating-logging-tainting.pdf

[59] Peter Macko and Margo Seltzer. 2012. A General-Purpose Provenance Library. In *Workshop on the Theory and Practice of Provenance (TaPP 2012)*. USENIX.

[60] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[61] Simon Miles, Paul Groth, Miguel Branco, and Luc Moreau. 2007. The requirements of using provenance in e-science experiments. *Journal of Grid Computing* 5, 1 (2007), 1–25.

[62] James Morris, Stephen Smalley, and Greg Kroah-Hartman. 2002. Linux security modules: General security support for the linux kernel. In *Security Symposium*. USENIX.

[63] Thomas Morris. 2011. Trusted Platform Module. In *Encyclopedia of Cryptography and Security*. Springer, 1332–1335.

[64] Thomas Moyer and Vijay Gadepally. 2016. High-throughput Ingest of Provenance Records into Accumulo. In *High Performance Extreme Computing*. IEEE.

[65] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A Holland, Peter Macko, Diana L MacLean, Daniel W Margo, Margo I Seltzer, and Robin Smogor. 2009. Layering in Provenance Systems.. In *USENIX Annual technical conference*. USENIX.

[66] Kiran-Kumar Muniswamy-Reddy, David Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-Aware Storage Systems. In *USENIX Annual Technical Conference, General Track*. 43–56.

[67] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo I Seltzer. 2009. Making a Cloud Provenance-Aware.. In *Workshop on the Theory and Practice of Provenance (TaPP 2012)*. USENIX.

[68] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo I Seltzer. 2010. Provenance for the Cloud.. In *Conference on File and Storage Technologies (FAST'10)*,

Vol. 10. USENIX, 15–14.

[69] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2014. noWorkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*. Springer, 71–83.

[70] Thomas Naughton, Wesley Bland, Geoffroy Vallee, Christian Engelmann, and Stephen L Scott. 2009. Fault injection framework for system resilience evaluation: fake faults for finding future failures. In *Proceedings of the 2009 workshop on Resiliency in high performance*. ACM, 23–28.

[71] Thomas Pasquier. 2017. CamFlow/camflow-dev: v0.2.1. (2017). DOI:10.5281/zenodo.571427, https://github.com/CamFlow/camflow-dev.

[72] Thomas Pasquier. 2017. CamFlow/information-flow-patch. (2017). https://doi.org/10.5281/zenodo.826436

[73] Thomas Pasquier and David Eyers. 2016. Information Flow Audit for Transparency and Compliance in the Handling of Personal Data. In *International Workshop on Legal and Technical Issues in Cloud Computing (CLaw'16)*. IEEE.

[74] Thomas Pasquier and Xueyuan Han. 2017. CamFlow/camflow-query: v0.1.0. (2017). DOI:10.5281/zenodo.571433, https://github.com/CamFlow/camflow-query.

[75] Thomas Pasquier, Jatinder Singh, Jean Bacon, and David Eyers. 2016. Information Flow Audit for PaaS clouds. In *International Conference on Cloud Engineering (IC2E)*. IEEE.

[76] Thomas Pasquier, Jatinder Singh, David Eyers, and Jean Bacon. 2015. CamFlow: Managed data-sharing for cloud services. *IEEE Transactions on Cloud Computing* (2015).

[77] Thomas Pasquier, Jatinder Singh, Julia Powles, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Data provenance to audit compliance with privacy policy in the Internet of Things. *Pervasive and Ubiquitous Computing, Special Issue on Privacy and IoT* (2017).

[78] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: Collecting High-Fidelity whole-system provenance. In *Annual Computer Security Applications Conference*. ACM, 259–268.

[79] Dean Povey. 1999. Optimistic security: a new access control paradigm. In *Workshop on New security paradigms*. ACM, 40–45.

[80] Ira Rubinstein and Joris Van Hoboken. 2014. Privacy and security in the cloud: some realism about technical solutions to transnational surveillance in the post-Snowden era. (2014).

[81] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture.. In *USENIX Security Symposium*, Vol. 13. 223–238.

[82] Can Sar and Cao Pei. 2005. Lineage File System. (2005). http://crypto.stanford.edu/~cao/lineage.

[83] Casey Schaufler. 2016. LSM: Stacking for major security modules. (2016). https://lwn.net/Articles/697259/.

[84] David Silveiro. 2016. TCPDump 4.5.1 - Crash (DoS). (2016). https://www.exploit-db.com/exploits/39875/.

[85] Yogesh L Simmhan, Beth Plale, and Dennis Gannon. 2005. A survey of data provenance in e-science. *ACM Sigmod Record* 34, 3 (2005), 31–36.

[86] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1, 43 (2001), 139.

[87] Anil Somayaji and Stephanie Forrest. 2000. Automated Response Using System-Call Delay.. In *Security Symposium*. USENIX, 185–197.

[88] Anil Buntwal Somayaji. 2002. *Operating system stability and security through process homeostasis*. Ph.D. Dissertation. The University of New Mexico.

[89] Lily Tsai and Aaron Bembenek. [n. d.]. prov-compress. ([n. d.]). https://github.com/aaronbembenek/prov-compress.

[90] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*. ACM, 42.

[91] Daniel J Weitzner. 2007. Beyond secrecy: New privacy protection strategies for open information spaces. *IEEE Internet Computing* 11, 5 (2007), 96–95.

[92] Allison Woodruff and Michael Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In *International Conference on Data Engineering*. IEEE, 91–102.

[93] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell DE Long. 2012. A hybrid approach for efficient provenance storage. In *International Conference on Information and Knowledge Management*. ACM, 1752–1756.

[94] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Darrell DE Long, Ahmed Amer, Dan Feng, and Zhipeng Tan. 2011. Compressing Provenance Graphs.. In *Workshop on the Theory and Practice of Provenance*. USENIX.

[95] Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, and Michel Dagenais. 2003. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Linux Symposium*. 494.

[96] Olive Qing Zhang, Markus Kirchberg, Ryan KL Ko, and Bu Sung Lee. 2011. How to track your data: The case for cloud computing provenance. In *International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 446–453.

A AVAILABILITY

The work presented in this paper is open-source and available for download at http://camflow.org under a GPL-3.0 license. Detailed installation instructions are available online.[a]

A.1 Running CamFlow

To get started with the system presented in this paper install Vagrant[b] and VirtualBox[c] then run the following commands:

```
1  git clone https://github.com/CamFlow/vagrant.git
2  cd ./vagrant/basic-fedora
3  vagrant plugin install vagrant-vbguest
4  vagrant up
5  # testing the installation
6  vagrant ssh
7  # check installed version against CamFlow head
8  camflow -v
9  uname -r
10 # check services
11 cat /tmp/audit.log # audit service logs
12 cat /tmp/camflow.clg # configuration logs
```

Listing 4: Running CamFlow demo in Vagrant.

Alternatively, CamFlow can be installed and kept up to date via the package manager. The packages are distributed via packagecloud[d] and available for Fedora distributions. Installation on a Fedora machine, do as follow:

```
1  curl -s https://packagecloud.io/install/repositories/
      camflow/provenance/script.rpm.sh.rpm | sudo bash
2  sudo dnf install camflow
```

Listing 5: Installing CamFlow on Fedora.

A.2 Repeating Evaluation Results

The instructions for reproducing the results presented in § 6 are online.[e] We assume that a Linux kernel running CamFlow has been installed. To run the various tests please follow the instructions:

```
1  git clone https://github.com/CamFlow/benchmark.git
2  cd benchmark
3  make prepare
4  # choose the config to run the tests under
5  make <whole/selective/off>
6  make run
7  # for more accurate results you may want
8  # to run test individually and reboot
9  # after each. Run for example:
10 make run_lmbench
```

Listing 6: Reproducing evaluation results.

NOTES

[a]https://github.com/CamFlow/CamFlow.github.io/wiki/Getting-Started
[b]https://www.vagrantup.com/
[c]https://www.virtualbox.org/
[d]https://packagecloud.io/camflow
[e]https://github.com/CamFlow/benchmark