Runtime Analysis of Whole-System Provenance

Thomas Pasquier[*]
University of Bristol

Xueyuan Han
Harvard University

Thomas Moyer
University of North Carolina at Charlotte

Adam Bates
University of Illinois at Urbana-Champaign

Olivier Hermant
MINES ParisTech
PSL Research University

David Eyers
University of Otago

Jean Bacon
University of Cambridge

Margo Seltzer
University of British Columbia

ABSTRACT

Identifying the root cause and impact of a system intrusion remains a foundational challenge in computer security. *Digital provenance* provides a detailed history of the flow of information within a computing system, connecting suspicious events to their root causes. Although existing provenance-based auditing techniques provide value in forensic analysis, they assume that such analysis takes place only retrospectively. Such post-hoc analysis is insufficient for realtime security applications; moreover, even for forensic tasks, prior provenance collection systems exhibited poor performance and scalability, jeopardizing the timeliness of query responses.

We present `CamQuery`, which provides inline, realtime provenance analysis, making it suitable for implementing security applications. `CamQuery` is a Linux Security Module that offers support for both userspace and in-kernel execution of analysis applications. We demonstrate the applicability of `CamQuery` to a variety of runtime security applications including data loss prevention, intrusion detection, and regulatory compliance. In evaluation, we demonstrate that `CamQuery` reduces the latency of realtime query mechanisms, while imposing minimal overheads on system execution. `CamQuery` thus enables the further deployment of provenance-based technologies to address central challenges in computer security.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; *Information flow control*; Intrusion detection systems;

KEYWORDS

Whole-system Provenance; Information Flow Tracking; Graph Processing; Linux Kernel

ACM Reference Format:
Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. 2018. Runtime Analysis of Whole-System Provenance. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243776

[*]Part of this work was completed at Harvard University and at the University of Cambridge.

1 INTRODUCTION

Timely investigation of system intrusions remains a notoriously difficult challenge [66, 94, 96]. While security monitoring tools provide an initial notification of foul play [13, 41, 86, 91, 95, 97], these indicators are rarely sufficient in and of themselves. Instead, crafting an appropriate response to a security incident often requires scouring terabytes of audit logs to determine an adversary's method of entry, how their reach spread through the system, and their ultimate mission objective. Such investigations not only require a human-in-the-loop, but are excruciatingly slow, at times requiring months of investigation and thousands of employee hours [56]. This delay between an event's occurrence and its diagnosis represents a tremendous window of opportunity for attackers – as they continue to exploit the system, defenders are still just getting their bearings.

Digital provenance (or *provenance* for short) refers to the data being used in a variety of ways to address the challenges of forensic audits. By parsing individual records into causal relationship graphs that describe a system's execution, provenance enables defenders to leverage the full historical context of a system and to reason about the interrelationships between different events and objects. With provenance, forensic investigations can trace back a given security indicator (e.g., a port scan) to the attacker's point of entry (e.g., a malicious email attachment) [53] and then trace forward from the entry point to determine what other actions the attacker has taken on the system.

Unfortunately, provenance-based auditing's growing popularity has uncovered significant limitations in its performance and scalability. Early efforts to integrate provenance querying into production systems indicated that, even for modestly small organisations (e.g., 150 workstations), forensic queries can take on the order of hours or days to complete [61]. In an actual attack scenario, where a timely incident response could make the difference between victory and defeat, such delays are unacceptable. Moreover, to date, provenance-aware systems have supported causal reasoning only as an after-the-fact forensic activity [54]; this is unfortunate, because provenance is also invaluable to a variety of runtime security tasks such as access control [76, 77], integrity measurement [92], and regulatory compliance [8, 15, 68, 81]. To date, the design of low latency mechanisms for realtime provenance analysis has not been given adequate consideration in the literature.

The goal of this work is to bridge the gap between runtime security monitoring and post-hoc forensic analysis. In support of this goal, we consider methods for the deep integration of provenance capture and analysis within the operating system. We introduce `CamQuery`, a framework that supports runtime analysis of provenance and thus enables its practical use for a variety of security applications. `CamQuery` pairs a runtime kernel-layer reference monitor – expanding and modifying `CamFlow` [79] – with a novel query module mechanism that enables runtime provenance analysis and even mediation of system events. `CamQuery` modules present a familiar vertex-centric API, as popularised by modern graph processing systems such as GraphChi [57] and GraphX [40]. In these vertex-centric platforms, full-graph analysis routines are expressed in terms of a small program that runs in parallel on every vertex (node) in the system. The graph-structured nature of provenance data makes this model a good fit and permits use of a familiar API. While these applications run directly over the live provenance stream, provenance can be simultaneously persisted to facilitate additional post-mortem and/or forensic analysis.

To demonstrate the generality of `CamQuery`, we consider several exemplar query applications in § 5. 1) a data loss prevention scheme [18] popular in provenance-security based community; 2) a provenance based intrusion detection scheme; 3) a mechanism to apply constraints on information flow; and 4) a provenance signature scheme. These case studies illustrate the rich space of design possibilities that are enabled through runtime provenance analysis. The source code for `CamQuery`, along with associated applications and datasets, is available at http://camflow.org.

This paper makes the following contributions:
`CamQuery`: We present the design and implementation of `CamQuery`, an analysis framework over a live provenance stream.
Whole-system provenance modelling: our work is the first to provide automated modelling of whole-system provenance through static analysis of the Linux kernel source code.
Exemplar Applications: We demonstrate `CamQuery`'s efficacy in security-related applications, such as preventing loss of sensitive data or assuring log integrity.
Performance Evaluation: We rigorously evaluate the performance of `CamQuery` to demonstrate its effectiveness in realistic operating environments.
Availability: We released an open-source implementation of `CamQuery`. Based on the Linux Security Modules framework, `CamQuery` is immediately deployable on millions of systems worldwide.

2 BACKGROUND

To provide context for the rest of the paper, we first introduce the concept of whole-system provenance and then outline some shortcomings of existing systems.

2.1 Whole-System Provenance

The W3C [19] defines provenance as a directed acyclic graph (DAG) where vertices represent *entities* (data), *activities* (transformations of data) and *agents* (persons or organisations), and edges represent relationships between those elements. Fig. 1 presents a simple example. In our context, *entities* are kernel objects, such as inodes, messages, and network packets; *activities* are tasks; and *agents* are users and groups.

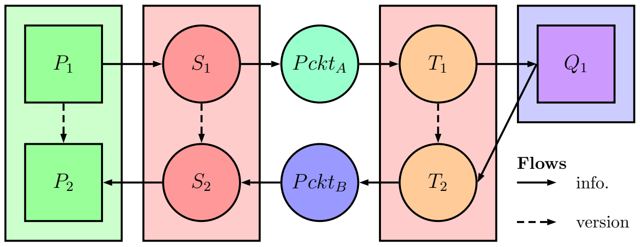

Figure 1: A simple provenance DAG: two processes (P and Q) exchange packets ($Pckt_A$ and $Pckt_B$) through their respective sockets (S and T).

In practice, it is impossible to represent a mutable process or file as a single vertex while simultaneously ensuring that the graph remains acyclic [21]. For example, in a naive representation, a process that both reads and writes a file immediately creates a cycle, because the process depends on the file (due to the read), and the file depends on the process (due to the write). Cycles are problematic. Edges in the provenance graph represent dependencies between the states of different objects and express causal relationships. Therefore, an object must depend only on the past (i.e., the state of an object cannot depend on a future state). The most commonly used cycle avoidance technique is to create multiple vertices per entity or activity [72], each representing a version or state of the corresponding object. We can see in Fig. 1 that new versions of the process P and sockets S and T are created as information flows through these objects.

Using provenance graphs, we can detect and provide attribution for malicious behaviour [43] or actively prevent attacks using provenance-based access control [76]. However, using provenance to prevent actions requires that provenance is *"complete and faithful to actual events"* [82]. Missing events could sever connections, resulting in failure to reveal an important information flow; errant provenance could falsely implicate a benign process. Pohly et al. [82] demonstrated that it was possible to satisfy such requirements by building provenance capture around the reference monitor concept [11] to mediate all events that should appear in the provenance graph. They called this approach *whole-system* provenance, which records events from system initialisation to shutdown.

2.2 Issues With Provenance Architectures

Existing provenance capture architectures were not designed with realtime support for security applications in mind. Therefore, unsurprisingly, they have some fundamental limitations. The traditional whole-system provenance capture stack, as first implemented in PASSv1 [72], is built of the following five layers:

- **the capture layer** records system events;
- **the collection layer** transports provenance information to where it may be used (e.g., using messaging middleware such as Kafka [2] or Flume [1]);
- **the storage layer** transforms system events into a provenance graph and persists it;

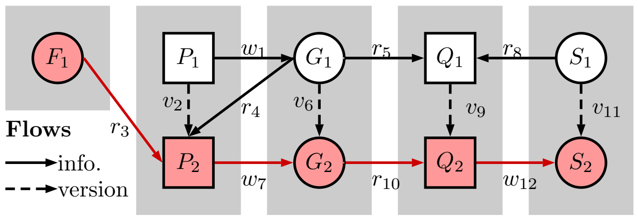

Figure 2: A demonstration of how path queries can be calculated through label propagation. The red (shaded) boxes indicate those vertices (with versions in subscripts) to which the "confidential" label is propagated. Confidential information flows from file F to socket S, through process P, file G, and process Q. r_i, v_i, and w_i stand for read, version, and write, respectively. The subscripts i represent event ordering.

- **the query layer** extracts provenance through queries relevant for a particular analysis;
- **the analysis layer** interprets the provenance in the context of an application.

The use of whole-system provenance for runtime security applications is a relatively recent phenomenon. Bates et al. [18] demonstrate provenance-based techniques to prevent loss of sensitive data in an enterprise, while Han et al. [43] use provenance to detect errant or malicious processes in a cloud environment. Both of their systems were built on top of the conventional stack described above. We argue that such an approach is suboptimal for provenance-based security applications, incurring latency penalties arising from the need to store data before querying or analysing it. Specifically, *when the goal of provenance analysis is mediation, delaying that analysis until after the data has been stored is impractical.* Therefore, while existing architectures may be appropriate for post-mortem forensic investigations, they are not ideal for runtime security applications. The goal of our work is to enable such applications through the introduction of vertex-centric, real-time analysis of streaming provenance.

3 RUNTIME ANALYSIS FRAMEWORK

In the previous section, we made the case for realtime analysis over the provenance data stream. We now present the design of CamQuery, a framework for enabling such analysis to support runtime provenance-based security applications.

3.1 Threat Model & Assumptions

We design CamQuery with consideration for an adversary that has gained remote access to a host. Once the adversary has gained access to the machine, they may engage in typical attacker behaviour such as installing malware, escalating their privilege level, or engaging in anti-forensic activities to hide evidence of their misdeeds. However, we make the common assumption that the adversary does not have physical access to the machine. Broadly, the goal of CamQuery is to securely facilitate the provenance-based analysis of the adversary's actions in real time.

Trusted Computing Base (TCB): The TCB of CamQuery includes a capture mechanism to generate a provenance graph from system events and a query mechanism to process the provenance at runtime, which we discuss at greater length in the remainder of this section. Because any loaded kernel module is granted unrestricted access to kernel memory, we assume that the entire kernel is distributed and installed in a trusted state, which is a typical assumption in kernel-layer security mechanisms. This assumption is made more reasonable through the use of integrity measurement techniques such as remote attestation and module signatures. Protecting the capture mechanism from attackers who are able to alter kernel behaviour is an important but orthogonal issue that we discuss in § 7.
Secure Provenance Store: If we wish to store provenance for post-mortem forensic analysis, an adversary must not be able to corrupt it. We assume the availability of secure provenance storage, which can be achieved through a variety of known techniques. For example, Hasan et al. [45] present a hash-chain-based method for protecting provenance, while Bates et al. [18] secure provenance storage and transmission through the use of type enforcement. By layering these systems, it becomes possible to ensure full-stack trustworthy provenance.
Checkpointing: We assume that CamQuery is deployed on a host that does not leverage checkpointing. Checkpointing systems pose a challenge for *all* provenance systems, because restoring a checkpoint effectively moves a system backwards in time. As a demonstrative example of this problem, consider a policy to prevent conflicts of interest [22], e.g., a policy to prevent a user who has read the Coca-Cola recipe from also reading the Pepsi recipe. If checkpointing could be used to rollback the system to a state before the Coca-Cola recipe was read, an adversary could easily violate the policy.

3.2 Motivating Example

To identify the operational requirements of CamQuery, we ground our discussion in a prior example of provenance-based runtime security applications. Bates et al. [18] present a loss prevention scheme (LPS) that disallows confidential information to be sent to an external IP address by issuing provenance ancestry queries on all network transmissions. Because this application was implemented on a conventional provenance capture stack, query latency rapidly became the bottleneck – even when the user queried a relatively small graph stored in an in-memory database, the responses took upward of 21ms. Worse, because response latency grew linearly with the size of the graph, one would expect this application to quickly grow unusable under realistic conditions.

In contrast to ancestry queries, an alternative method of implementing LPS would be to propagate security labels along the provenance graph in realtime, as demonstrated in Fig. 2. Because each object will be associated with the correct security label at the enforcement point, graph traversal is no longer necessary and an authorization decision can be made in constant time. Note that while this approach is akin to taint tracking, a provenance-based approach allows for the expression of more complex queries than is possible in a conventional taint-tracking system. With a provenance-based approach, we can express subtle propagation constraints based on

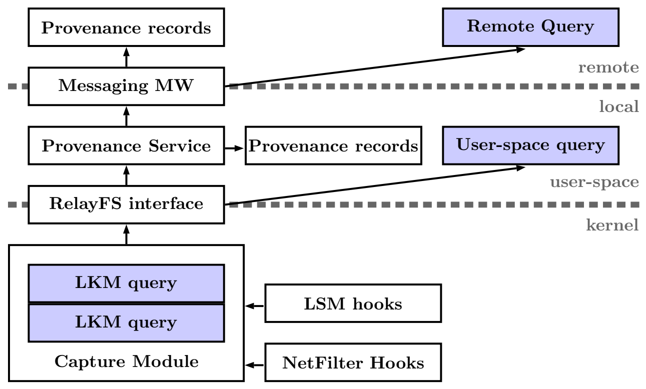

Figure 3: An overview of the `CamQuery` framework.

properties of the graph (we demonstrate this in **Example #1** in § 5). For example, Bates et al.'s LPS system propagates labels only along certain edge types, which is not possible in a data-centric taint-analysis system.

This approach to performing LPS can be generalized to a variety of other runtime security applications.[1] For example, in access control [76, 77], stream-based analysis can be used to express constraints on the properties of paths in the graph in a manner similar to computation tree logic (e.g., *all* paths *from* an external socket must not lead to disk *until* they have gone through an anti-virus process) [28]. Such constraints can be evaluated by building primitives above a value propagation algorithm. This allows, for example, policies such as declassification and path disjointedness to be built to enforce conflict-of-interest constraints [22]. With this in mind, the goal of `CamQuery` is to facilitate runtime security applications such as those considered above. In addition, as we show in § 5, the framework is sufficiently rich to be used, for example, to generate feature vectors in an intrusion detection setting.

3.3 Overview

Fig. 3 presents an overview of the `CamQuery` framework. `CamQuery` captures system events using `LSM` and `NetFilter` hooks; those events are transformed into a provenance graph within the `capture module` (i.e., `CamFlow`). The `capture module` feeds graph elements (i.e., edges and vertices) to stacked `CamQuery` queries, which are either built directly into the kernel or implemented as a loadable kernel module. The kernel transfers these graph elements to user space for 1) consumption by user space queries; 2) recording for post-hoc analysis; or 3) transmission to a remote machine. `CamQuery` embodies the design goal of ensuring a standard query implementation mechanism, independent of the three deployment options, discussed in § 4.

3.4 Provenance Monitor

Like prior kernel-layer provenance capture systems (e.g., LPM [18], HiFi [82]), `CamFlow` introduces a *provenance monitor* in the kernel. A provenance monitor is a provenance capture mechanism that satisfies the reference monitor concept [10], possessing the properties of complete mediation, tamperproofness, and verifiability. The relevance of these guarantees in the context of provenance capture is that they ensure that the provenance history is complete and accurate, even in the presence of an active attacker. While past provenance monitors generally only denied system accesses if they were unable to generate a new record of the access (e.g., out of memory), `CamQuery` exposes a general mechanism for system mediation, allowing security applications to authorize or deny new access based on the provenance history of the concerned principals. Further details are in § 4.2.

[1]We return to the subject of example provenance-based security applications in § 5.

3.5 `CamQuery` API

`CamQuery` provides an API, inspired by graph-processing frameworks such as GraphChi [57] and GraphX [40], enabling straightforward implementation of value propagation applications. A query application consists of three functions:

(1) **init:** called upon query initiation to initialise the query's variables;
(2) **out_edge(v, e):** called on every outgoing edge e of vertex v;
(3) **in_edge(e, v):** called on every incoming edge e of vertex v.

`CamQuery` invokes `out_edge` and `in_edge` in a manner guaranteeing that edges are processed according to the partial order implied in paths in the graph and in topological order of the vertices.

`CamQuery` calls the developer-defined `out_edge` and `in_edge` functions with two parameters containing `edge` and `node` data structures. These structures expose attributes of the underlying kernel objects they represent (e.g., inode, process, shared memory), allowing the developer to reference or modify the objects associated with the new system event. For example, the data structure representing a process vertex contains information such as UID, GID, namespaces, security context, system and user time, memory consumption, etc.; in turn, the edge data structure contains information such as offset, flags, mode, etc.. There are around two dozen vertex types, e.g., path, network addresses, network packet, and shared memory states (complete list online [7]). Similarly, there are over three dozen different edge types covering families of system calls (complete list online [6]). By specifying conditional constraints on the processing of vertex/edge labels and values, developers can express specific, complex queries.

In addition to the manipulation of the provenance objects and existing kernel objects, `CamQuery` also provides functions that allow developers to associate new labels or values with edges and vertices (e.g., `add_label`, `add_ptr`). Listing 1 shows a query that implements a loss-prevention scheme, which we describe at greater length in § 5. Associating labels with graph elements allows developers to easily implement, in a few lines of code, mechanisms such as taint tracking, information flow control, or access control. Futhermore, using data structure association it is possible to build more complex graph analytics. For example, we show in § 5 how to associate complex data structures with kernel objects and perform inlined computation while traversing the graph. From that, we can compute, at runtime, feature vectors used to perform intrusion detection.

`CamQuery` explicitly decouples the graph analysis implementation from the underlying kernel infrastructure. The goal is to allow

```c
#define KERNEL_QUERY
#include "include/camquery.h"

static label_t confidential;

static void init( void ){
  confidential = get_label("confidential");
}

static int out_edge(union prov_msg* node, union prov_msg*
      edge){
  switch (edge_type(edge)){
    case RL_WRITE:
    case RL_READ:
    case RL_SND:
    case RL_RCV:
    case RL_VERSION:
    case RL_VERSION_PROCESS:
    case RL_CLONE:
      if ( has_label(node, confidential) )
        add_label(edge, confidential);
  }
  return 0;
}

static int in_edge(union prov_msg* edge, union prov_msg*
      node){
  if ( has_label(edge, confidential) ) {
    add_label(node, confidential);
    if( node_type(node) == ENT_INODE_SOCKET )
      return PROVENANCE_RAISE_WARNING;
  }
  return 0;
}

QUERY_NAME("My Example Query");
QUERY_DESCRIPTION("An example query");
QUERY_AUTHOR("John Doe");
QUERY_VERSION("0.1");
QUERY_LICSENSE("GPL");
register_query(init, in_edge, out_edge);
```

Listing 1: CamQuery query in C.

development of new provenance modules with a minimum of engineering effort. For example, traditional taint tracking or information flow control implementations require extensive engineering effort [55, 85], while it is possible to implement these applications in CamQuery using only a few dozen lines of code.

4 IMPLEMENTATION

We have implemented CamQuery for Linux 4.14.15 and validated its use on Fedora 27. The work presented here is fully implemented, used in multiple research projects, and is available online on GitHub (https://github.com/CamFlow) under a GPL v2 license.

4.1 Capture Mechanism

We built CamQuery on top of the CamFlow provenance capture system [3, 79, 80], our actively-maintained provenance monitor built as a stackable Linux Security Module (LSM) [69]. Compared to other existing capture techniques [34, 72], an LSM-based approach ensures that CamFlow can observe and mediate all information flows between processes and kernel objects [27, 31, 36, 51] (see § 4.2 for further discussion).

Recording exact interactions between shared states (e.g., mmap files, shmem, etc.) is challenging. CamFlow records those interactions by conservatively assuming that information always flows between processes and shared states. We represent shared states as entities. In the provenance graph, we add a relation from a process to the associated shared states when it receives information (e.g., reading a file), and a relation from the associated shared states to the process when it sends information (e.g., writing a file). We track shared memory by parsing through the memory data structure (mm_struct) associated with each task. Additionally, we extended CamFlow to track provenance at the thread level rather than the process level. Note that CamFlow is the first whole-system provenance capture mechanism to do so. Process memory is represented as a shared state between threads in the provenance graph. We made these changes on top of the original design of CamFlow to obtain more accurate provenance and consequently more accurate results in security applications such as intrusion backtracking [53]. However, conservatively assuming the existence of information flows can lead to false positives. We discuss this limitation and its potential solutions in § 7.

To support runtime analysis, further changes to CamFlow were necessary. Existing provenance capture mechanisms, including past versions of CamFlow, do not directly generate graph elements in the kernel but instead generate logs of events that are processed in user space as part of the storage layer [18, 34, 71, 72, 82]. We extended CamFlow to generate the graph directly at the point of capture for two reasons: 1) event ordering is easier, as opposed to previous systems' complex computations to reconstruct kernel states and event orderings in user space; 2) more importantly, event ordering is made a precondition of the graph analysis in kernel space.

We modified the capture mechanism to embed limited provenance metadata alongside kernel objects to perform cycle avoidance in the kernel [71, 72]. The cycle avoidance algorithm is entirely based on local properties of a node (i.e., information about incoming and outgoing information flows) and does not require maintenance of any global state. Fundamentally, we create a new version any time an object that sent information receives new information. This guideline guarantees global acyclicity and avoids the creation of a new state of an object that depends on the future.

Finally, we modified CamFlow to publish graph components (i.e., edges and vertices) as the system executes, while providing the following two partial ordering properties: 1) all incoming edges to a vertex are published before any outgoing ones; 2) edges and vertices along a path are published in order. CamQuery processes edges and vertices as they are published.

4.2 Ensuring Completeness and Accuracy

The design and implementation of CamQuery extend the guarantees of past provenance monitors to support runtime provenance analysis. The introduction of a query mechanism, which is described below, can be used to further restrict system access. The standard mechanisms used to secure the deployment of past provenance monitors are applicable to our system. It naturally follows that CamQuery possesses the same security properties as do past provenance

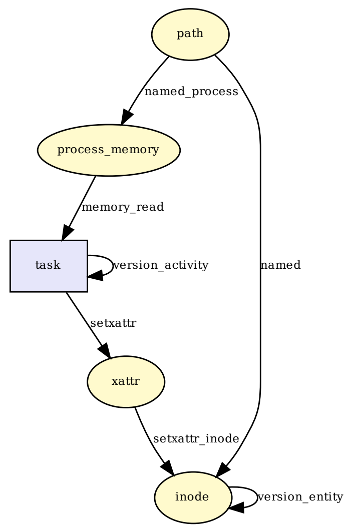

Figure 4: Provenance model for the `inode_post_setxattr` hook.

monitors, Therefore, we omit a complete security analysis, and instead refer interested readers to the work of Bates et al. [18] for a detailed analysis of the security properties of provenance monitors.

Past provenance monitor implementations (e.g., Hi-Fi [82] and LPM [18]) derive security properties from the guarantees provided by the formal verification of LSM placement [27, 31, 51, 99], ensuring that they capture all interactions between kernel objects. We extend this prior assessment of provenance completeness and accuracy:

Completeness: We want to ensure that all flows of information between kernel objects are properly recorded. The LSM framework [69] was originally implemented to support Mandatory Access Control (MAC) schemes but not information flow tracking. Recent work by Georget et al. [35, 36] demonstrated, through static analysis of the kernel code base, that the LSM framework is applicable to information flow tracking, and that by adding a small number of LSM hooks, it was possible to properly intercept all information flows between kernel objects. Building on their work, we maintain a patch [5] to the LSM framework that allows `CamFlow`, and by extension `CamQuery`, to provide stronger guarantees than do previous whole-system provenance capture mechanisms.

Accuracy: We also provide accuracy guarantees for the recorded provenance. We automatically analyse kernel source code to model the provenance generated by any `CamFlow`-supported LSM hook (see Fig. 4 and Fig. 5 for an example of such a model). We then manually verify that all models meet our expectations.[2] Finally, through static analysis, we identify the LSM hooks associated with each system call and generate the associated provenance model,

[2]Unfortunately, manual verification currently requires significant knowledge of the Linux kernel.

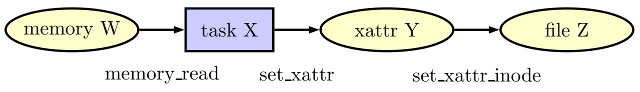

Figure 5: A whole-system provenance subgraph representing a valid instance of the model shown in Fig. 4.

which we again manually verify. This process is embedded in our continuous integration testing, with results automatically updated in our Git repository [4] so that as the capture mechanism and the underlying kernel evolve, we ensure the accuracy of our provenance capture. We welcome meaningful scrutiny by third parties. We believe this work is the first attempt towards formalisation of whole-system provenance.

We continue work on automated and formal analysis of whole-system provenance capture. Our future plans include combining static analysis techniques with dynamic provenance model benchmarking, as described in e.g., Chan et al. [23]. Although we currently just assume a correct implementation of the ordering properties described in § 4.1, our goal is to formalise these as well.

4.3 `CamQuery` Query Configurations

Depending on the security and performance requirements of a deployment, it may not always be practical to embed analysis applications in the kernel. For example, computationally expensive analysis may affect system stability, or a proprietary analysis tool may need to be run on a separate host from the capture point. Therefore, our implementation supports a variety of different configuration options that enable built-in kernel level analysis, loadable kernel module analysis, local user-level analysis, and remote user-level analysis on a machine subscribed to the provenance stream. While all of the deployment options run the same code, only the in-kernel implementations can *prevent* policy violations; like previous systems, the user space and remote implementations can only *detect* violations after the fact.

Kernel-Based Configurations: `CamQuery` implements in-kernel queries using either Loadable Kernel Modules (LKMs) or directly linked objects. LKMs are dynamically loaded object files that run in kernel space and have access to a subset of the kernel API. Directly linked objects allow for shipping queries as part of the kernel.

Loading a query LKM invokes the `register_query` function, which in turn invokes the `init` function. After registration, the kernel invokes `out_edge` and `in_edge` whenever `CamFlow` records a new event. Given the partial ordering property of our collection, a vertex v will receive all values propagated through an in-edge before `out_edge` runs on its outgoing edges. If several queries are loaded, they execute sequentially in their load order. These functions are actually executed before the actions they describe, because they are executed from LSM framework hooks designed to implement Mandatory Access Control schemes. This enables `CamQuery` to prevent policy violations rather than merely detecting them.

`CamQuery` maintains approximately 20 bytes of provenance state for kernel objects, e.g., `inode`, `cred`. By associating provenance with the kernel objects themselves, queries have access to the kernel objects, granting them visibility into kernel states. Provenance for long-lived kernel states, such as `inodes`, persists across reboots through the use of extended attributes.

While the focus of this paper is enabling runtime query and analysis, we observe that our framework creates opportunities at other layers of the provenance stack as well. For example, we were able to rewrite `CamFlow`'s optional selective capture mechanism [79] using `CamQuery` to reflect the modular nature of this component. This mechanism makes it possible to limit provenance captured to a process, an object, or characteristics of the provenance graph, e.g., recording the actions of only those processes belonging to a specific SELinux context e.g., to track the actions of an `httpd` server.

User space Configurations: The user space implementations of `CamQuery` operate similarly to the kernel one. Rather than producing an LKM, user-level queries produce a service that reads provenance records from either `relayfs` or a messaging middleware. Queries process the stream by placing records into a sorted in-memory edge list and a persistent vertex map.

The `CamQuery` capture mechanism writes records to per-core `relayfs` files that are read in per-core batches, producing a collection of partially-ordered edge lists that are not necessarily totally ordered. To facilitate ordered processing of edges, a user space utility performs a merge of the per-core lists as follows – for an out-edge e received at time t, all in-edges must have been received by $t + T$, where T is the QoS threshold. At regular time intervals, the query processes all the edges satisfying $t < now - T$.

Rather than using timestamps to order edges, we use edge IDs; the capture mechanism guarantees that edge ID ordering respects the ordering properties described in § 4.1. In a similar manner, we use provenance DAG causality relationships on network packets to produce a partial order across machines. We then merge the per-core edge lists and the network packets to produce a sorted edge list. The query processes each edge sequentially by invoking the `in_edge` and `out_edge` functions.

In addition to maintaining a list of edges, a user space query maintains a map of vertices. We discard an edge after it is processed; we discard a vertex either after processing an edge referencing a new version of the vertex or after terminating events specific to the object (e.g., a network packet will not be referenced after it has been received or a process will not be referenced after it has been terminated). We show in § 6 that, in practice, this represents a small memory footprint.

Vertex garbage collection relies on the semantics of system events. We therefore record events relating to the life cycle of long-lived objects (e.g., representing in the graph process kernel data structures being freed). These events are not necessarily pertinent to the tracking of information flows, but greatly help with garbage collection. If the framework were to be applied to other types of provenance (e.g., Spark provenance [50]), the garbage collection algorithm would require different domain knowledge.

Converting the code in Listing 1 to its user space equivalent is trivial. We modify Line 1 to reflect the proper target, currently one of `MW_QUERY` or `RELAY_QUERY`, indicating from where the service will obtain data (a middleware-provided data stream or `relayfs`, respectively). We add two more query attributes after line 39. `QUERY_MSG` specifies the messaging middleware broker address and topic. Note that although the kernel transmits information to `relayfs` before executing the action corresponding to the query, as we do not control when user processes are scheduled, we cannot guarantee that the query service has an opportunity to process the provenance before the corresponding action is taken. As such, the user level implementation, and by extension the distributed one, can guarantee only violation detection, not prevention.

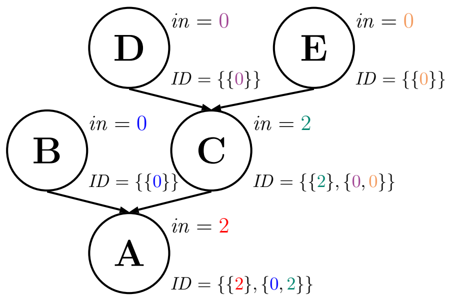

Figure 6: Calculating vertices' structural identity (Depth=1).

Algorithm 1 Encoding Structure Identity (pseudo code).

```
1: function OUT_EDGE(vertex, edge)
2:     Calculate DTW between its own ID and parent ID
3:     Publish feature vector
4:     Write to the edge its own ID
5: function IN_EDGE(edge, vertex)
6:     Increment vertex in-degree counter
7:     Read and save parent ID
8:     Merge parent ID to build own ID
```

Discussion: The different guarantees available from different `CamQuery` configurations provide a rich set of trade-offs. While in-kernel queries have access to the underlying kernel data structures and can prevent events from occurring, they incur overhead on every system call. § 6 illustrates this power/performance trade-off. On the other hand, user space queries can perform runtime monitoring only, raising alerts relatively quickly, but not quickly enough to prevent events from occurring. However, such queries can build on existing libraries to e.g., perform log analysis [79]. Additionally, the scheduler is responsible for scheduling user space queries, so it can more easily adjust to system workload as shown in § 6.

5 EXAMPLE APPLICATIONS

We designed `CamQuery` to enable development of important security and compliance applications, such as intrusion detection [43, 44], enforcement of software licenses, and compliance with data regulation [78, 81]. During development, we implemented several algorithms inspired by the literature to validate the suitability of the framework. In this way, we ensured that we could implement meaningful provenance analysis at runtime in widely different use cases. We provide the examples below to illustrate the range of applications that can be implemented with `CamQuery`.

Example #1: Data Loss Prevention. We first demonstrate how the framework works with a relatively simple graph processing algorithm implementing the loss prevention scheme of Bates et al. [18], which prevents sensitive data from leaving a system (as discussed in § 3.2 and shown in Fig. 2).

Listing 1 shows the implementation of this application. The query contains four main components: the `init` function (lines 6–8), the `out_edge` function (lines 10–23), the `in_edge` function (lines 25–32), and a set of query attribute statements (lines 34–39). Before query registration, `CamQuery` executes the query attribute statements to set the query's properties exactly once. Then, during registration, it calls the `init` function exactly once. Subsequently, `CamQuery` invokes `in_edge` and `out_edge` for every edge in every active query.

The LPS scheme considers only certain flows of information as meaningful in the context of the policy. Therefore, it propagates labels (lines 19–20) only over the relevant flows (line 11–18), raising a warning if the label ever reaches a socket (lines 26–29). In more complex scenarios, developers can maintain global states within a query or associate more complex data structures with edges or vertices. Notably, we emphasise that Listing 1 contains all of the required runtime logic for a label-based loss-prevention system, demonstrating the efficiency with which security applications can be expressed in `CamQuery`. Outside of this application, our LPS scheme assumes only: 1) a labelling state that tags sensitive information sources with the *confidential* label, 2) that correctness requires handling explicit information flow only, not side channels, and 3) that sensitive information that reaches a system exit point (e.g., a socket) raises a warning.

We can design more complex algorithms around programmable label propagation. An example extension uses label propagation to detect abnormal behaviour in a system. For example, one can easily use `CamQuery` to track the origin of executables and sensitive data as previously labelled. An indicative abnormal system behaviour might be an executable that did not originate from a trusted repository manipulating sensitive data. Once a potentially harmful pattern is detected, techniques such as intrusion backtracking [53] can be used to manually assess the situation. Other more sophisticated, automated techniques are also available; we refer interested readers to the work by Eshete et al. [30], which describes, in more depth, use cases of provenance for label-based intrusion detection techniques.

Example #2: Intrusion Detection. Recent work explores how to improve the efficacy of Intrusion Detection Systems (IDS) using provenance [43]. With this work as inspiration, we show how to implement anomaly detection using `CamQuery`. Provenance-based intrusion detection is still a nascent field that has not yet been demonstrated to be robust against a realistic active adversary; we use it merely as a demonstration of `CamQuery`'s ability to allow for the construction of complex feature vectors.

Our proposed approach to provenance-based intrusion detection is based on unsupervised learning techniques. Our goal is to learn how the system normally behaves, build a model of such behaviour, and detect large deviations from the model. We generate provenance graphs from the executions of our system in a controlled environment under normal conditions. As in previous work [43, 44], we capture provenance data during multiple runs of a cloud application under a representative workload and build a model of normal behaviour.

Our example IDS uses a replicator neural network [47] (RNN, also known as an autoencoder) to detect anomalies in a graph. An RNN consists of an encoder and a decoder. The encoder performs compression of the feature vector. The decoder then reconstructs the input feature vector from the compressed vector. The objective of training is to minimise the distance between the input of the encoder and the output of the decoder. RNNs are often used for outlier detection, as they often have difficulty reconstructing feature vectors that diverge significantly from the training dataset. In our case, we leverage this behavior to detect abnormal structures in the provenance graph. Using `CamQuery`, we construct a feature vector for every vertex, which is composed of the following three parts: 1) vertex attributes (e.g., vertex type, security context, UID, namespace, etc.); 2) changes of some attributes over time (e.g., UID, memory or CPU usage for processes, etc.); and 3) the structural identity [84] of the vertex, which represents the graph structure surrounding the vertex.

Vulnerability ID	Detection rate	False positive
`MariaDB` race condition exploit [37]	100%	0%
`MySQL` root privilege escalation [38]	50%	0%
`Nagios` core root privilege escalation [39]	90%	0%

Table 1: Preliminary results for our `CamQuery` IDS mecahnism.

Structural identity is a vectorisation of the graph neighborhood, which represents the context in which a vertex exists, and is critical for anomaly identification in outlier detection [49] and intrusion detection [24]. We define a neighborhood as the $n - ancestry$ of a vertex, because descendants are unknown when we generate feature vectors at runtime. The structural identity is built from ancestor in-degrees. For each vertex, we maintain a list, L, of length $n+1$. Let i be the 0-based index of each element of this list. L_0 is the in-degree of the vertex itself and $L_i, i > 0$ is an in-degree sequence, a list consisting of the in-degrees of the ith generation ancestors. Thus, a vertex with two parents, each of which has no ancestors, is associated with the following list: $\{\{2\}, \{0, 0\}\}$; Fig. 6 shows a concrete example.

Following Ribeiro et al. [84], we use Dynamic Time Warping (DTW), a technique for calculating the similarity between two temporal sequences [20], to calculate the distance between two *in-degree sequences*. We then populate the feature vector of a vertex with each of the DTW distances between a vertex and its ancestry. This set of distance is the structural identity of the vertex.

`CamQuery` propagates in-degree sequences along each path of a provenance graph. Using the `out_edge` function, each vertex passes its in-degree sequence to its descendants. A child vertex receives sequences from all of its parent vertices and updates its own sequences using the `in_edge` function. Algorithm 1 illustrates this. When the `out_edge` function runs, the vertex contains enough information to calculate its structural identity.

Table 1 shows some preliminary results of the intrusion detection scheme. We generate training data by executing unexploited instances of each vulnerable application. We then test the IDS by running a collection of normal and abnormal application executions. While a full-fledged evaluation of our IDS mechanism is beyond the scope of this paper, we measure the computational cost of feature vector generation in § 6.

Example #3: Information Flow. `CamQuery` can execute single-pass algorithms that rely on value propagation along paths in the

Q1) **path existence**
$\exists p : A \Rightarrow B$;
Q2) **existence of a vertex on all paths between two vertices**
$\forall p : A \Rightarrow B, \exists v \in p, v \neq A$ AND $v \neq B$;
Q3) **absence of a vertex on all paths between two vertices**
$\forall p : A \Rightarrow B, v \notin p$;
Q4) **path disjointedness**
$\forall v \in p, v \notin p'$;
Q5) **constraints on properties and types in a path**
$\forall v \in p$, if v_{type} = T then $P(v)$, for a specified property P and type T.

Figure 7: `CamQuery` can be used to query a variety of information flow properties. Here, we denote a path from vertex A to vertex B as $p : A \Rightarrow B$.

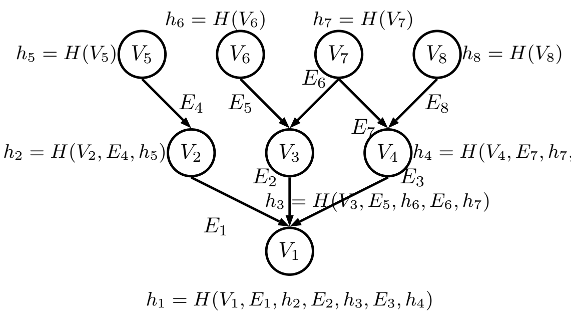

Figure 8: `CamQuery` can be used to assure integrity by generating a signed provenance graph.

graph. For example, we implemented the simple primitives summarised in Figure 7. Each implementation required just a few dozen lines of C code. The data loss prevention scheme introduced in Example #1, for example, tests for path existence (Q1).

Using these queries, `CamQuery` can aid in the enforcment or auditing of regulatory compliance. The Sarbanes-Oxley act (SOX) applies to publicly held US corporations. The intent of the law is to establish security controls and accountability of personnel to protect against data tampering to hide fraud. While the law itself does not specifically address computing systems, every major corporation today relies heavily on computers to process financial data and report to the Securities and Exchange Commission (SEC). Specifically, Sections 302 and 404 detail the required safeguards for data to ensure accuracy in financial reporting and required disclosures. To be SOX compliant, an organization must carefully consider and have policies for data creation, publishing, retention, access, distribution, and lifecycle.

We consider here just three of the cases mentioned above. The first control is data access (Section 302.4.B), which requires that companies have controls in place to track accesses to data and ensure that company officers are aware of all relevant data. The provenance records kept as forensic evidence ensure full compliance with the requirement to track data access. A report detailing all the data entities appearing in the captured provenance could inform company officers of the “relevant” data. Additionally, corporations could instantiate policies to detect accesses that do not comply with the act.

The second control we consider is data creation and the ability for a reporting officer to attest that the reported information is valid. This requires that data must not be tampered with before reports are created and filed with the SEC. We can write `CamQuery` policies that restrict data access to only those users and activities involved in report generation. There are multiple ways to express this, one of which would be to label activities that are known to be acceptable, then write policies that verify that all activities between data generation and the SEC filing are labelled as such. This is a query of type Q5. An alternative is to label all unacceptable techniques, e.g., using a text editor on the data, and check that no such activities appear in the path between the data and SEC filing. This is a Q3 type of query.

Sarbanes-Oxley Title V deals with analyst conflicts of interest. It requires financial analysts to disclose conflicts of interest, ensuring that investors are not being misled by the biases of a financial analyst. These conflicts of interest can be avoided using separation of concerns policies [22] that create information barriers preventing the exchange of information that would produce a conflict of interest. As a specific example, consider a financial analyst who is working with one company (Company A) as part of a hostile takeover of another company (Company B). Information concerning the takeover must not be transmitted to the brokerage department that could use the information to alter customer investments to increase profits for the financial company. This is a Q4 type of query.

Example #4: Graph Integrity. To ensure the integrity of our provenance graph, we implemented the directed acyclic graph signature scheme proposed by Aldeco-Pérez et al. [9]. This technique is often cited in the literature as a solution to provenance integrity.

The system generates a chain of hashes according to the graph structure, as shown in Fig. 8. The capture mechanism then signs these hashes. The analysis engine can re-calculate the hashes for a graph to verify that they correspond to the signed value. An advantage of this scheme is efficient verification, as it is not necessary to verify the entire graph to verify vertex V.

We leverage a kernel keyring infrastructure for key management (we took inspiration from eCryptfs [42]) and the cryptographic API to perform related operations. The resulting solution is a heavyweight, in-kernel query in the evaluation in § 6. While it was easy to implement graph signing in `CamQuery`, unsurprisingly, creating signatures on every system call incurs significant overhead, even when the cryptographic algorithm itself is relatively lightweight. Our measurements suggest that the provenance graph signature scheme [9] is impractical at scale and inadequate when whole-system provenance capture is considered. It also serves as a cautionary tale: while it is easy to implement a variety of applications using `CamQuery`, not all such applications will exhibit acceptable performance. Creating provenance integrity schemes that are practical at scale is an important open problem beyond the scope of this paper.

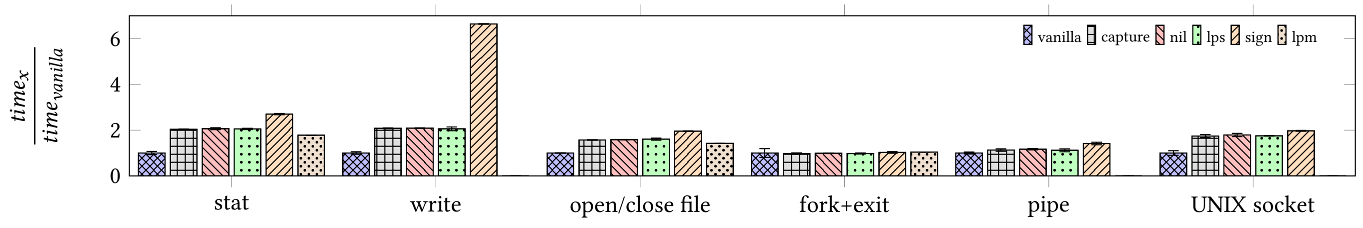

Figure 9: Normalised overhead of queries (LPM capture overhead as reported in [18] is given when available).

Test Type	**vanilla**	**capture**	**nil**	**lps**	**sign**
Process tests, times in μs, smaller is better					
stat	1.20	2.44	2.48	2.46	3.24
read	0.22	0.35	0.35	0.36	1.05
write	0.15	0.31	0.32	0.31	1.01
open/close file	2.04	3.21	3.24	3.28	4.00
fork+exit	87.6	85.5	86.6	85.7	89.8
fork+shell	862	860	866	855	861
Latencies in μs, smaller is better					
pipe	3.47	3.92	4.05	3.88	4.91
UNIX socket	3.70	6.44	6.61	6.47	7.28

Table 2: LMbench measurements.

6 EXPERIMENTAL EVALUATION

We evaluate both the in-kernel and local user-space implementations to determine how much overhead `CamQuery` introduces and how that overhead is split between provenance capture and query support.

We use workloads derived from those found in the whole-system provenance literature to provide meaningful points of comparison. We run the benchmarks on a bare metal Fedora 27 machine, with Linux kernel 4.14.15 and `CamFlow` 0.3.10 with an Intel i7-7700 2.8 GHz CPU and 32 GiB of RAM. Due to space constraints, we present only a subset of our results. Instructions on obtaining our code and reproducing all our results are available online (http://camflow.org) following recommendations by Collberg et al. [26]. Throughout the evaluation, we refer to the following setup:
vanilla: unmodified Linux 4.14.15 kernel;
capture: whole-system provenance capture;
nil: nil in-kernel query (`in_edge` and `out_edge` simply return zero);
lps: the loss prevention scheme in-kernel query described in § 5 Example #1;
sign: the provenance signature in-kernel query described in § 5 Example #4;
ids: the user-space query building feature vectors for the IDS described in § 5 Example #2.

6.1 In-kernel Queries

Micro-benchmark: We used LMbench [67] to illustrate the impact of the provenance capture and query on system call performance. Table 2 and Fig. 9 present a subset of LMbench results. Our provenance capture overhead is comparable to that reported for other systems [18, 82]. This is as expected and provides a sanity check. More interesting is that the addition of online querying introduces relatively little overhead.

Indeed, execution time of a single system call is equal to $V_s + n_s(C + Q) + m_s C$, where V_s is the execution time of the system call s on a vanilla kernel. n_s is the number of edges in the graph corresponding to the system call s (e.g., a socket `send event` contains at least 2 edges, one from the process to the socket, and the other from the socket to the packet, and potentially edges corresponding to kernel object versions). m_s is the number of vertices in the graph corresponding to the system call s. C is the cost of capture and Q is the cost of the query. The relative overhead is higher when V_s is small, as C and Q are independent of the underlying system call execution time. The overhead of LPM [18] (and of other previous provenance capture systems e.g., [72, 82]) is $V_s + C_s$ where C_s is the cost of capturing the event corresponding to s, as LPM records system events rather than directly producing the graph structure (see § 4.1).

One of the advantages `CamQuery` provides over prior work is a drastic reduction in the time between an attack and its detection. Bates et al. [18] reported that it took their system 21ms to evaluate the same policy and further noted that "these results are highly dependent on the size of the graph. [Their] test graph, while large [6.5 million vertices, and 6.8 million edges], would inevitably be dwarfed by the size of the provenance on long-lived systems" [18]. The authors suggested that the performance could be further improved by using deduplication [93] and pre-pruning techniques [14, 79]. However, they did not evaluate the performance impact of such improvements. They did, however, report that graph size can be reduced by up to 89% through pre-pruning techniques [14, 17]. Even if we assume that the reduction produces a proportional improvement in query time, the resulting 2.31ms per query is several orders of magnitude larger than the overhead imposed by `CamQuery` for a similar application (`lps` in Table 2).

Macro-benchmark: We contextualise the significance of the overhead measured in the micro-benchmarks using the Phoronix test suite [58]. We select benchmarks commonly used in the system provenance literature. Consistent with the micro-benchmark results, the macro-benchmark results (Table 3) show that provenance capture introduces negligible overhead for the kernel build benchmark and up to 15% overhead for Postmark. For reference, we also include reported overheads for prior systems (PASS and LPM). As the Linux kernel versions (2.6.x for the two mentioned systems vs 4.14.15 for `CamQuery`) and the underlying hardware vary greatly across these evaluations, the results simply provide context and suggest that `CamQuery` exhibits capture overhead comparable to

Test Type	**vanilla**	**capture**	**nil**	**lps**	**sign**	**PASS**	**LPM**
Execution time in seconds, smaller is better							
unpack	14.98	15.48 (3%)	15.63 (4%)	15.76 (5%)	16.68 (11%)	NA	NA
build	402	411 (2%)	416 (3%)	417 (3%)	448 (11%)	15.6%	2.7%
4kB to 1MB file, 10 subdirectories, 4k5 simultaneous transactions, 1M5 transactions							
postmark	127	145 (14%)	144 (13%)	146 (15%)	226 (78%)	11.5%	7.5%

Table 3: Macro-benchmark results. PASS [71] and LPM [18] overhead as reported by the authors.

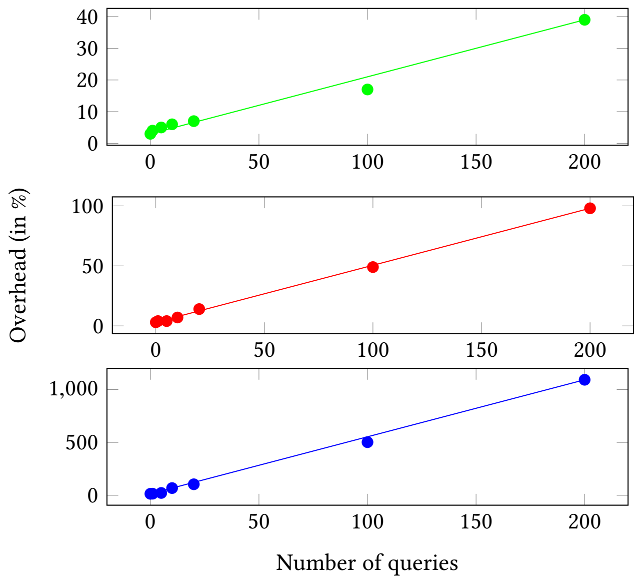

Figure 10: Benchmark results (`unpack` in green/top, `build` in red/middle and `postmark` in blue/bottom) as a function of the number of active queries (we run 0 to 200 concurrent `lps` queries). Note the difference in the y axes for the different benchmarks.

prior systems. The overhead is higher for benchmarks where the number of system calls per unit of time is larger, as the overhead is only incurred on interactions between a process and the system call interface.

Query stacking: The prior results show that a single query introduces acceptable overhead; next we assess the impact of an increasing number of queries executing concurrently. We run the macro-benchmarks from Table 3 with a varying number of active queries and show the results in Fig. 10. On the positive side, overhead increases linearly with the number of queries. On the negative side, the Postmark overhead is particularly high, because it is a system-intensive workload, and system calls trigger query evaluation. While `build` and `unpack` spend approximately 10% and 18% of their time, respectively, in the kernel, Postmark spends 85% of its time in the kernel, making 253,000 system calls per second (over twice the rate of the other benchmarks). It should be noted that production systems running hundreds of queries is unrealistic. Further, we plan to explore the possibility of merging a set of queries into a

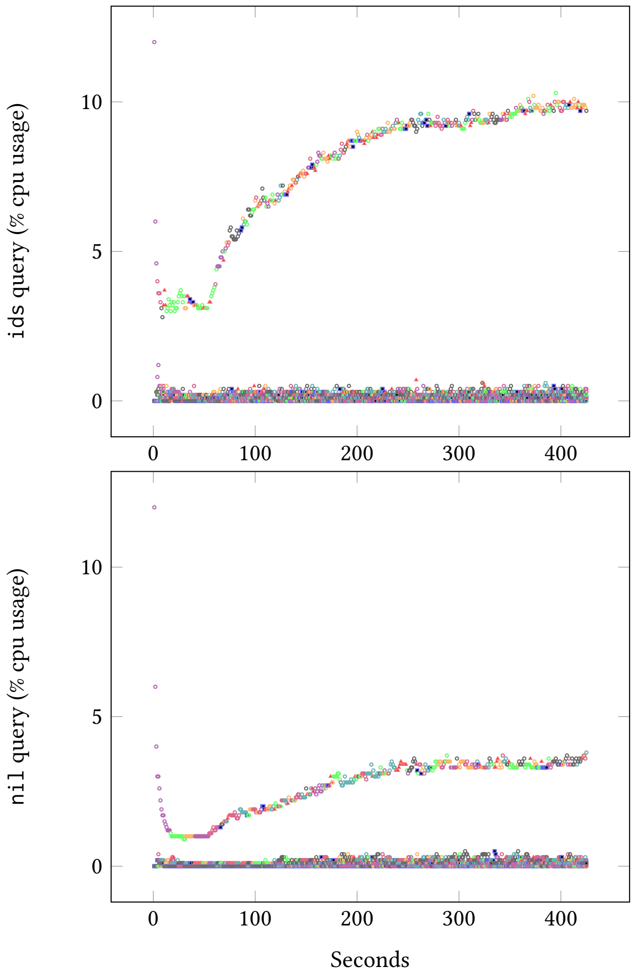

Figure 11: Percentage of CPU usage per core (each colour represents one of the eight cores) used by the `ids` (top) and `nil` (bottom) queries during a kernel compilation.

single module, with the goal of reducing the number of redundant operations. This is a non-trivial task left for future work.

6.2 User space queries

Next, we want to evaluate the performance impact of running queries in user space. We compare the overhead of the `vanilla` and `lps` in-kernel configurations from the previous section to that

Test Type	vanilla	**in-kernel**	overhead (over capture)	**userspace**	overhead (over capture)
Execution time in seconds, smaller is better					
unpack	14.98	15.76	**5%**	15.91	**6%**
build	402	417	**4%**	427	**6%**
4kB to 1MB file, 10 subdirectories, 4k5 simultaneous transactions, 1M5 transactions					
postmark	127	146	**15%**	147	**15%**

Table 4: Overhead of the `lps` query when compiling the Linux kernel.

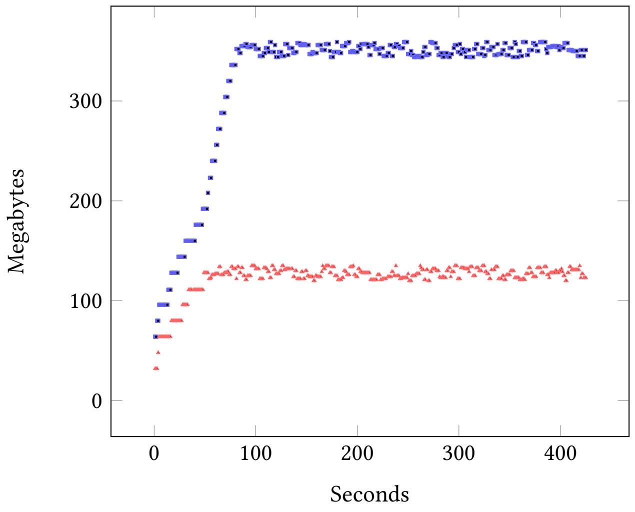

Figure 12: Memory used by the `ids` (blue/square) and `nil` (red/triangle) queries during a kernel compilation.

of the `lps` user space configuration, where the query is run as a `systemd` managed service running on the same machine as the workload, reading provenance from `relayfs`. Table 4 shows the results for the Linux kernel `unpack` and `build` benchmark and Postmark. Note that the user space overhead is only minimally larger than the in-kernel overhead.

We next investigate how user space queries impact system workload by running the `nil` query and the complex `ids` query. The `ids` query generates feature vectors used by a machine learning algorithm to perform intrusion detection. We run the kernel `build` benchmark as our system workload, as it generates a relatively large and complex graph (just over 25 million edges were processed by each query) when compared with the other two benchmarks. At regular intervals, we record the memory and per-core CPU consumption of the two queries.

In contrast to kernel queries, the user space query runs at regular intervals, processing all the newly arrived edges. `Relayfs` creates a ring buffer mapped to a pseudofile per CPU core to transmit data to the query service. The service runs one reader thread per core, reading the data from its `relayfs` file and populating the edge list and the vertex map. Another thread, the processing thread, sorts the edge list and performs the query at regular time intervals. As shown in Table 4, the core running the processing thread reaches about 9% utilisation for the `ids` query and 4% for the `nil` query, while the other cores, which are running reader threads, have a CPU utilisation between 0% and 2%. The user space query competes with other workloads on the system for CPU time, which may degrade application performance. The multi-coloured nature of the lines in Table 4 shows that the processing thread moves among the cores.

Fig. 12 illustrates the memory consumption for the same queries. The memory used by the user space query corresponds to the list of edges and vertices (which includes the propagated values). The memory usage stabilises, as vertices are garbage collected, to around 305 MB for the `ids` query and 125 MB for the `nil` query.

7 CHALLENGES & DISCUSSION

`CamQuery` has limitations and raises interesting questions that go beyond the particular framework presented here.

Query Language: `CamQuery` uses a programmable graph processing framework to express policies, rather than the seemingly more user-friendly DSL approach. A DSL would undoubtedly need to be designed with a particular application in mind (e.g., compliance enforcement, access control, etc.) and it would be challenging to make it amenable to queries such as the intrusion detection feature vector computation. We believe that such languages are important but are part of individual applications rather than a general framework. We plan to explore the design and development of a DSL for the provenance-based access-control scenario. We note also that, concurrent to this study, Gao et al. introduced SAQL [32] and AIQL [33] which both introduce a domain-specific query language to aid in forensic investigation. They are implemented over data streams and persistent storages respectively. We plan to explore how a similar language could be expressed through our vertex-centric query API.

Distributed Systems: A challenge for `CamQuery` is the ability to reason about computations that occur in a distributed system. The user space implementation can be extended to support these systems with relative ease, but doing so eliminates the possibility of performing policy enforcement (see § 4.3). Supporting enforcement in a distributed system requires that the query be partitioned into per-machine segments combined into a kernel enforcement mechanism. This partitioning necessitates the ability for the system to validate that the other machines in the system will accurately enforce the policy, i.e., they are high-integrity and have the necessary enforcement mechanisms and provenance policies loaded. Once a machine validates the integrity and suitability of a system, it must generate a “proof” that the policy has been enforced.

The ability to perform policy enforcement would open up new opportunities for `CamQuery` in distributed settings, but also new challenges. For example, it enables `CamQuery` to function as the building block for a secure distributed taint propagation system with the potential to allow sophisticated logic using complex labels. To implement such a system, however, two important considerations must be taken into account, among others. First, we must ensure that `CamQuery` is: minimally invasive, fully integrated into the existing network stack; and is compatible with non-provenance-aware hosts, especially if we hope to insert arbitrarily complex taint information in network packets. Second, transmission must be authenticated and tamperproof to e.g., man-in-the-middle attack. The latter might be addressed by existing secure network protocols such as IPSec, but technical challenges remain.

Trust: The ability for a system to prove statements about its integrity and processing state is best suited to trusted computing, e.g., trusted hardware and remote attestation. In the above distributed system setting, there is a need for systems to generate "proofs" of their current state. These proofs need to account for several system characteristics, including 1) the current integrity state of the system (hardware, firmware, software, etc.); 2) the currently loaded policies and; 3) the current state of the data being processed. To prove the current integrity state of the system and the currently loaded policies, we can turn to techniques such as the Linux Integrity Measurement Architecture (IMA) [87]. IMA measures the load-time integrity of user space applications and files read by root. These measurements are stored in the Trusted Platform Module (TPM) to support remote attestation, i.e., generating an unforgeable proof of the measurements stored in the TPM. The TPM is an inexpensive trusted hardware component that provides a small amount of protected storage for measurements and cryptographic keys. These measurements can be signed by a key loaded into the TPM to support remote attestation, proving the current integrity state of the loaded system. IMA will measure the policies being loaded as an LKM as long as the policy loading is done by root since the default policy measures all files read by root. The remote attestation allows a remote verifier to determine the current state of the kernel and user-space applications. What is still needed are mechanisms that enable a remote verifier to validate that the currently loaded policies are correctly enforced.
Storage: The issue of storing provenance is orthogonal to the topic of this paper. However, we believe that the work presented here represents a paradigm shift in provenance systems. Whole-system provenance implementations have been faced with the issue of building a back-end that can ingest high throughput [70], provide integrity and non-repudiability [12], and handle large volumes of data while providing low latency queries. Decoupling query performance from storage overhead introduces myriad new architectures for such systems.
False positives from flow tracking: A well-understood limitation of the proposed approach is the potential for false positives when information flows are inferred. For example, if a `task` reads from a file and writes to another, whole-system provenance capture systems conservatively assume that information was transferred, even though it is not necessarily always the case. Conservatively inferring information flow via shared memory is another major source of false positives. Similar issues also arise in most system-level information flow control or taint tracking systems. A potential solution to reduce the number of false inferences is to capture information flow within applications, using techniques such as bitcode transformation [90], binary rewriting [25, 59], or static analysis [74]. Such techniques are related to provenance layering [71], the capture of internal application provenance alongside system level provenance to improve the accuracy of provenance records. While the capture of such provenance is a well-understood problem, its analysis and scalability remain relatively unexplored.

8 RELATED WORK

We place this work both in the context of prior work on whole-system provenance capture and more general information flow tracking approaches, as techniques such as Information Flow Control and Taint Tracking share many characteristics with provenance collection systems.
Provenance Systems. There have been several provenance capture mechanisms implemented in the Linux kernel [18, 65, 72, 82]. LPM [18] uses provenance DAGs to enforce information flow constraints by querying graphs at sink points (e.g., at the network interface). The authors verify that paths from a source A to a sink B respect some well-defined properties expressed in the query. However, their approach requires performing database queries where query latency is a function of the graph size, which increases linearly over time. Therefore, it suffers from a lack of scalability, slowing down over time as provenance accumulates. `CamQuery` addresses this issue by executing queries at runtime over the provenance stream, introducing bounded overhead independent from the graph size as shown in § 6.
Provenance Reduction. Recently, the issue of provenance storage and query performance has received considerable attention in the literature. LogGC performs garbage collection on redundant events that have no forensic value [60], while BEEP [59] and MPI [64] improve post-mortem analysis by solving the problem of dependency explosion. PrioTracker [61] accelerates forensic queries by prioritising the traversal of rare events in large provenance graphs. These systems primarily exist at either the storage and query layer of the provenance stack; in tackling the issue of log reduction through taint tracking, ProTracer employs a similar approach to `CamQuery` by merging the capture and storage layers [65]. While this work has led to dramatic improvement in the efficiency of provenance, `CamQuery` achieves the orthogonal but interrelated goal of improving provenance performance through deep integration of analysis routines with the underlying capture framework. An interesting avenue for future research would be considering how the above reduction techniques could be incorporated into the flattened provenance stack that `CamQuery` envisions.
Provenance Applications. Provenance has been leveraged in the service of a variety of security applications. Because provenance can be used to generate a model of known good executions of a system, recent work has considered using provenance data to perform anomaly detection [43, 46]. Han et al. [43] use machine learning (ML) algorithms to detect outlier graph structures. Hassan et al. [46] use a graph grammar to build deterministic finite state automata and verify that the graph can be parsed. Unlike the example shown in § 5, where we generate data as the graph is being produced, they accumulate provenance to generate "windows" that are then analysed. We have shown in § 5 that it was possible to generate feature vectors for the ML-based approach. It should also be possible to implement the graph parsing stage (i.e., detection stage) of Bates et al.'s work using the `CamQuery` framework.

Park et al. [77] formalise the notion of provenance-based access control (PBAC) systems along three dimensions: 1) the type of data used to make decisions (observed vs. disclosed provenance [21]); 2) object dependencies (information flow between objects) vs. user dependencies (information flow between users); and 3) whether policies are available to the system or learnt through the traversal of provenance graphs; `CamQuery`, like most PBAC systems in the literature ([16, 89]), uses observed provenance, although it could be augmented by disclosed provenance. Layering of provenance systems [71] could enable such a capability, although we are not aware of any layered PBAC enforcement model. We plan to explore this approach in future work, with both application level [71] and network level provenance [102].

Information Flow Control Systems. Previous work on information flow control enforcement at the OS level, such as HiStar [98], Flume [55], and Weir [75], uses labels to define security and integrity contexts that constrain information flows between kernel objects. Labels map to kernel objects, and a process requires decentralised management capabilities to modify its labels. Point-to-point access control decisions are made to evaluate the validity of an information flow. Through transitivity, it is possible to express constraints on a workflow (e.g., collected user information can only be shared with third parties as an aggregate). SELinux [88] provides a similar information flow control mechanism but without decentralised management. A typical way of representing and thinking about information flow in a system is through a directed graph. However, current object labelling abstractions do not take advantage of this representation, and it is difficult to reason about when defining policies. `CamQuery` differs from these systems in that it allows the implementation of such mechanisms directly on the graph abstraction.

Taint Tracking Systems. Techniques such as "colouring" [48] or tainting [73] of data and resources have been proposed as a means to detect data misuse. TaintDroid [29] implements such an approach in the Android OS to detect applications disclosing personal information to an unexpected third party (e.g., disclosing the owner's contact list to advertisers). `CamQuery` can be used to achieve similar results as taint tracking systems but provides more control through its expressive query mechanism on how taints are propagated within the system. Furthermore, the provenance records, kept as forensic evidence, provide a rich resource that can be mined to identify, understand, and explain the source of a disclosure.

Security Monitoring. In today's enterprise environments, security incidents occur when a primary indicator of compromise is triggered from security monitoring software such as an anti-virus detection alert or a blacklisted URL in the organisation's network logs [62]. In current security products, such indicators report only limited context as to the circumstances under which the alert occurred, e.g., process ID or packet header information, but do not report the historical chain of events that led to the suspicious activity. Past work has attempted to compensate for this lack of lineage through the fusion [13, 41] or correlation [86, 91, 95, 97] of multiple indicators of the compromise. However, it does not address the fundamental limitation that security monitoring tools lack the ability to reason over the entire context of a system execution. Thus, attack reconstruction has typically been relegated to (offline) forensic analysis [52, 53, 60, 63–65, 83, 100, 101]. In contrast, `CamQuery` provides a mechanism to build runtime security monitoring based on the entire history of system execution, thus representing a significant step forward compared to the state-of-the-art.

9 CONCLUSION

More than a decade ago, PASS [72] represented a paradigm shift in how we think about provenance capture, moving from application-specific capture, to a system-wide holistic mechanism. In this paper, `CamQuery` rethinks how we envision provenance applications, severing the always-present, implicit link to database back-ends. We make the distinction between runtime detection applications which should be built above live streams of provenance data to identify policy violations or anomalies, and forensic applications that run post-mortem, leveraging database support to provide explanations. By drastically rethinking the conventional provenance architecture, we are able to reduce the time between an event (such as an attack, data leakage, non-compliance with regulation, etc.) and its detection, by several orders of magnitude, while simultaneously storing the data for post-mortem forensic investigation. We continue to actively develop `CamFlow` and `CamQuery` as we investigate provenance applications. The work is entirely open-source and we invite others to build upon it.

AVAILABILITY

The work presented in this paper is open-source and available for download at http://camflow.org/ under a GPL v2 license.

ACKNOWLEDGMENTS

This work was supported by the US National Science Foundation under grants SSI-1450277 End-to-End Provenance, CNS-1750024 CAREER and CNS-1657534 Transparent Capture and Aggregation of Secure Data Provenance for Smart Devices. Early versions of CamFlow open source software were supported by UK Engineering and Physical Sciences Research Council grant EP/K011510 CloudSafetyNet.

REFERENCES

[1] Accessed August 25, 2018. Apache Flume. https://flume.apache.org/.
[2] Accessed August 25, 2018. Apache Kafka. https://kafka.apache.org/.
[3] Accessed August 25, 2018. CamFlow. http://camflow.org/.
[4] Accessed August 25, 2018. CamFlow automated reports. https://github.com/CamFlow/camflow-dev/tree/master/docs.
[5] Accessed August 25, 2018. CamFlow information flow patch. https://github.com/CamFlow/information-flow-patch.
[6] Accessed August 25, 2018. CamFlow relations list. https://github.com/CamFlow/camflow-dev/blob/master/docs/RELATIONS.md.
[7] Accessed August 25, 2018. CamFlow vertices list. https://github.com/CamFlow/camflow-dev/blob/master/docs/VERTICES.md.
[8] Rocio Aldeco-Perez and Luc Moreau. 2009. Information Accountability supported by a Provenance-based Compliance Framework. (December 2009). http://eprints.soton.ac.uk/268305/ Event Dates: Monday 7th – Wednesday 9th December 2009.
[9] Rocío Aldeco-Pérez and Luc Moreau. 2010. Securing provenance-based audits. In *International Provenance and Annotation Workshop*. Springer, 148–164.
[10] J. P. Anderson. 1972. *Computer Security Technology Planning Study*. Technical Report ESD-TR-73-51. ESD/AFSC, Hanscom AFB, Bedford, MA.
[11] James P Anderson. 1972. *Computer Security Technology Planning Study. Volume 2.* Technical Report. Anderson (James P) and Co Fort Washington PA.
[12] Nikilesh Balakrishnan, Lucian Carata, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. 2017. Non-repudiable disk I/O in untrusted kernels. In *Asia-Pacific Workshop on Systems*. 24:1–24:6.
[13] Tim Bass. 2000. Intrusion Detection Systems and Multisensor Data Fusion. *Commun. ACM* 43, 4 (2000), 99–105.
[14] Adam Bates, KR 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 (TaPP'15)*. USENIX, 7–7.
[15] Adam Bates, Ben Mood, Masoud Valafar, and Kevin Butler. 2013. Towards Secure Provenance-based Access Control in Cloud Environments. In *Proceedings*

of the Third ACM Conference on Data and Application Security and Privacy (CODASPY '13). ACM, New York, NY, USA, 277–284. https://doi.org/10.1145/2435349.2435389

[16] Adam Bates, Ben Mood, Masoud Valafar, and Kevin Butler. 2013. Towards secure provenance-based access control in cloud environments. In *Conference on Data and Application Security and Privacy*. ACM, 277–284.

[17] Adam Bates, Dave Jing Tian, Grant Hernandez, Thomas Moyer, Kevin RB Butler, and Trent Jaeger. 2017. Taming the Costs of Trustworthy Provenance through Policy Reduction. *Transactions on Internet Technology* 17, 4 (2017), 34.

[18] Adam M Bates, Dave Tian, Kevin RB Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *USENIX Security*. 319–334.

[19] 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/.

[20] Donald J Berndt and James Clifford. 1994. Using dynamic time warping to find patterns in time series. In *KDD workshop*, Vol. 10. Seattle, WA, 359–370.

[21] Uri Braun, Simson Garfinkel, David A Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I Seltzer. 2006. Issues in automatic provenance collection. In *Provenance and annotation of data*. Springer, 171–183.

[22] David FC Brewer and Michael J Nash. 1989. The Chinese Wall security policy. In *Symposium on Security and Privacy*. IEEE, 206–214.

[23] 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'17)*. USENIX.

[24] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 15.

[25] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*. IEEE, 749–754.

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

[27] 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 (CCS'02)*. ACM, 225–234.

[28] E Allen Emerson and Joseph Y Halpern. 1982. Decision procedures and expressiveness in the temporal logic of branching time. In *Symposium on Theory of Computing*. ACM, 169–180.

[29] 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.

[30] Birhanu Eshete, Rigel Gjomemo, Md Nahid Hossain, Sadegh Momeni, R Sekar, Scott Stoller, VN Venkatakrishnan, and Junao Wang. 2016. Attack Analysis Results for Adversarial Engagement 1 of the DARPA Transparent Computing Program. *arXiv preprint arXiv:1610.06936* (2016).

[31] 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 (CCS'05)*. ACM, 330–339.

[32] Peng Gao, Xusheng Xiao, Din Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Whan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. 2018. SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *Proceedings of the 27th USENIX Security Symposium (Security'18)*. Baltimore, MD, USA.

[33] Peng Gao, Xusheng Xiao, Zhichun Li, Kangkook Jee, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. 2018. AIQL: Enabling Efficient Attack Investigation from System Monitoring Data. *USENIX Annual Technical Conference (ATC'18)* (2018).

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

[35] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel, and Valérie Viet Triem Tong. 2017. Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory. In *International Conference on Software Engineering and Formal Methods*. Springer, 1–16.

[36] 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'17)*. IEEE/ACM, 10–16.

[37] Dawid Golunski. 2016. CVE-2016-6663: MySQL / MariaDB / PerconaDB 5.5.x/5.6.x/5.7.x - 'mysql' System User Privilege Escalation / Race Condition. https://www.exploit-db.com/exploits/40678/.

[38] Dawid Golunski. 2016. CVE-2016-6664: MySQL / MariaDB / PerconaDB 5.5.x/5.6.x/5.7.x - 'root' System User Privilege Escalation. https://www.exploit-db.com/exploits/40679/.

[39] Dawid Golunski. 2016. CVE-2016-9566: Nagios < 4.2.4 - Privilege Escalation. https://www.exploit-db.com/exploits/40921/.

[40] 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 *Conference on Operating Systems Design and Implementation (OSDI'14)*, Vol. 14. 599–613.

[41] Guofei Gu, Alvaro A. Cárdenas, and Wenke Lee. 2008. Principled Reasoning and Practical Applications of Alert Fusion in Intrusion Detection Systems. In *Symposium on Information, Computer and Communications Security (ASIACCS'17)*. ACM, 136–147.

[42] Michael Austin Halcrow. 2005. eCryptfs: An enterprise-class encrypted filesystem for Linux. In *Proceedings of the 2005 Linux Symposium*, Vol. 1. 201–218.

[43] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, 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.

[44] Xueyuan Han, Thomas Pasquier, and Margo Seltzer. 2018. Provenance-based Intrusion Detection: Opportunities and Challenges. In *Workshop on Theory and Practice of Provenance (TaPP'18)*. ACM.

[45] Ragib Hasan, Radu Sion, and Marianne Winslett. 2009. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *Conference on File and Storage Technologies (FAST 09)*. USENIX.

[46] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs. In *Network and Distributed Systems Security Symposium*. Internet Society.

[47] Simon Hawkins, Hongxing He, Graham Williams, and Rohan Baxter. 2002. Outlier detection using replicator neural networks. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 170–180.

[48] Kai Hwang and Deyi Li. 2010. Trusted cloud computing with secure resources and data coloring. *Internet Computing, IEEE* 14, 5 (2010), 14–22.

[49] Dino Ienco, Ruggero G Pensa, and Rosa Meo. 2017. A semisupervised approach to the detection and characterization of outliers in categorical data. *IEEE Transactions on Neural Networks and Learning Systems* 28, 5 (2017), 1017–1029.

[50] 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.

[51] 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.

[52] Xuxian Jiang, A. Walters, Dongyan Xu, E.H. Spafford, F. Buchholz, and Yi-Min Wang. 2006. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. In *International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, 38–38.

[53] Samuel T King and Peter M Chen. 2003. Backtracking intrusions. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 223–236.

[54] Ryan KL Ko, Markus Kirchberg, and Bu Sung Lee. 2011. From system-centric to data-centric logging-accountability, trust & security in cloud computing. In *Defense Science Research Conference and Expo (DSR), 2011*. IEEE, 1–4.

[55] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 321–334.

[56] George Kurtz. 2010. Operation Aurora Hit Google, Others. Available at http://securityinnovator.com/index.php?articleID=42948&sectionID=25.

[57] Aapo Kyrola, Guy E Blelloch, Carlos Guestrin, et al. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Conference on Operating Systems Design and Implementation (OSDI'12)*, Vol. 12. 31–46.

[58] Michael Larabel and Matthew Tippett. [n. d.]. Phoronix test suite. http://www.phoronix-test-suite.com.

[59] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of NDSS '13*.

[60] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: Garbage Collecting Audit Log. In *Conference on Computer and Communications Security (CCS'13)*. ACM, 1005–1016.

[61] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysisfor Enterprise Security. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS'18)*. San Diego, CA, USA.

[62] John Lyle, Andrew P Martin, et al. 2010. Trusted Computing and Provenance: Better Together. In *Workshop on Theory and Practice of Provenance (TaPP'10)*. USENIX.

[63] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows. In *Annual Computer Security Applications Conference*. ACM, 401–410.

[64] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *USENIX Security Symposium*.

[65] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Network and Distributed System Security Symposium (NDSS'16)*. Internet Society.

[66] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Symposium on Operating Systems Principles (SOSP'15)*. ACM, 378–393.

[67] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference (ATC'96)*. 279–294.

[68] Luc Moreau and Mufajjul Ali. 2014. A provenance-based policy control framework for cloud services. (May 2014). http://eprints.soton.ac.uk/364997/

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

[70] Thomas Moyer and Vijay Gadepally. 2016. High-throughput ingest of data provenance records into Accumulo. In *High Performance Extreme Computing Conference (HPEC'16)*. IEEE, 1–6.

[71] 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 (ATC'09)*.

[72] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. 2006. Provenance-aware storage systems. In *USENIX Annual Technical Conference (ATC'06)*. 43–56.

[73] Divya Muthukumaran, Dan O'Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch. 2015. FlowWatcher: Defending against Data Disclosure Vulnerabilities in Web Applications. In *Conference on Computer and Communications Security (CCS'15)*. ACM, 603–615.

[74] Andrew C Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 228–241.

[75] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC enforcement on Android. In *USENIX Security Symposium*. 1119–1136.

[76] Dang Nguyen, Jaehong Park, and Ravi Sandhu. 2013. A provenance-based access control model for dynamic separation of duties. In *International Conference on Privacy, Security and Trust (PST'13)*. IEEE, 247–256.

[77] Jaehong Park, Dang Nguyen, and Ravi Sandhu. 2012. A provenance-based access control model. In *International Conference on Privacy, Security and Trust (PST'13)*. IEEE, 137–144.

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

[79] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical Whole-System Provenance Capture. In *Symposium on Cloud Computing (SoCC'17)*. ACM, ACM.

[80] Thomas Pasquier, Jatinder Singh, David Eyers, and Jean Bacon. 2015. CamFlow: Managed Data-Sharing for Cloud Services. *IEEE Transactions on Cloud Computing* (2015).

[81] 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. *Springer Personal and Ubiquitous Computing* (2017).

[82] 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.

[83] Phillip A. Porras, Martin W. Fong, and Alfonso Valdes. 2002. A Mission-Impact-Based Approach to INFOSEC Alarm Correlation. In *International Symposium on Recent Advances in Intrusion Detection*. Springer, 95–114.

[84] Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. 2017. struc2vec: Learning Node Representations from Structural Identity. In *International Conference on Knowledge Discovery and Data Mining*. ACM, 385–394.

[85] Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. 2009. Laminar: Practical fine-grained decentralized information flow control. In *Conference on Programming Language Design and Implementation*, Vol. 44. ACM.

[86] Alireza Sadighian, José M. Fernandez, Antoine Lemay, and Saman T Zargar. [n. d.]. ONTIDS: A Highly Flexible Context-Aware and Ontology-Based Alert Correlation Framework. In *International Symposium on Foundations and Practice of Security*. Springer, 161–177.

[87] 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.

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

[89] Wai Kit Sze and R Sekar. 2015. Provenance-based Integrity Protection for Windows. In *Annual Computer Security Applications Conference*. ACM, 211–220.

[90] Dawood Tariq, Maisem Ali, and Ashish Gehani. 2012. Towards Automated Collection of Application-Level Data Provenance.. In *Workshop on Theory and Practice of Provenance (TaPP'12)*.

[91] F. Valeur, G. Vigna, C. Kruegel, and R. A. Kemmerer. 2004. Comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on Dependable and Secure Computing* 1, 3 (2004), 146–169.

[92] Frank Wang, Yuna Joung, and James Mickens. 2017. Cobweb: Practical Remote Attestation Using Contextual Graphs. In *Workshop on System Software for Trusted Execution (SysTEX'17)*. ACM.

[93] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell DE Long. 2013. Evaluation of a hybrid approach for efficient provenance storage. *ACM Transactions on Storage (TOS)* 9, 4 (2013), 14.

[94] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Symposium on Operating Systems Principles (SOSP'09)*. ACM, 117–132.

[95] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-scale Log Analysis for Detecting Suspicious Activity in Enterprise Networks. In *Annual Computer Security Applications Conference*. ACM, 199–208.

[96] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 4.

[97] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX.

[98] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, 263–278.

[99] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. 2002. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th USENIX Security Symposium*.

[100] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX, 603–618.

[101] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. Lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX, Berkeley, CA, USA, 629–644.

[102] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure network provenance. In *Symposium on Operating Systems Principles (SOSP'11)*. ACM, 295–310.