Information Flow Audit for PaaS clouds

Thomas F. J.-M. Pasquier, Jatinder Singh, Jean Bacon
Computer Laboratory, University of Cambridge
Cambridge, United Kingdom
Email: firstname.lastname@cl.cam.ac.uk

David Eyers
Department of Computer Science, University of Otago
Dunedin, New Zealand
Email: dme@cs.otago.ac.nz

***Abstract*—With the rapid increase in uptake of cloud services, issues of data management are becoming increasingly prominent. There is a clear, outstanding need for the ability for specified policy to control and track data as it flows throughout cloud infrastructure, to ensure that those responsible for data are meeting their obligations.**

This paper introduces Information Flow Audit, an approach for tracking information flows within cloud infrastructure. This builds upon CamFlow (Cambridge Flow Control Architecture), a prototype implementation of our model for data-centric security in PaaS clouds. CamFlow enforces Information Flow Control policy both intra-machine at the kernel-level, and inter-machine, on message exchange. Here we demonstrate how CamFlow can be extended to provide data-centric audit logs akin to provenance metadata in a format in which analyses can easily be automated through the use of standard graph processing tools. This allows detailed understanding of the overall system. Combining a continuously enforced data-centric security mechanism with meaningful audit empowers tenants and providers to both meet and demonstrate compliance with their data management obligations.

I. Introduction

There is increasing awareness of privacy and security concerns in cloud computing. Such concerns mean that certain regulated sectors such as health, finance or government are reluctant to use public cloud services [1], [2]. The wide use of social media has highlighted occasions when data willingly provided by users has been used outside of the data owner's expected context [3], [4]. Those providing cloud services, or using cloud infrastructure to provide services, are often subject to legal obligations, be they through contracts (SLAs) or imposed by regulation, such as data protection law [5].

However, despite the significant body of law and regulation that applies to cloud computing [6], as yet there is little technical basis for enforcing and demonstrating compliance. While much effort has been put into understanding and demonstrating quality of service, solutions that provide transparency and demonstrate the proper handling of data are less mature. Strong, reliable mechanisms to control data dissemination and demonstrate compliance are clearly needed [5].[1]

We have argued [7], as have others [8], that *Information Flow Control* (IFC) addresses these requirements. IFC enforces the proper (i.e. according to a specified policy) use of data, by controlling its exchange between components of a system over the dimensions of *secrecy* and *integrity* [9]. We have shown that IFC can provide Security as a Service for the cloud [10], complementing existing security mechanisms through end-to-end, data-bound security policy.

[1]We are concerned with situations of corporate/organisational compliance; surreptitious actions by malicious parties or government agencies—e.g. for reasons of national security, through the US Patriot Act, or the French 2013 Military Program Law—are beyond the scope of this discussion.

We extend IFC to collect audit records that can be used to demonstrate compliance with data handling requirements, through what we term *Information Flow Audit* (IFA). Compliance concerns can be internally or externally imposed on an organisation, company, industry or product, perhaps emerging from contractual obligations (including SLAs), legal regulation, internal policy or industry standards. In cloud services, managing obligations and demonstrating compliance requires the means for monitoring and understanding the circumstances in which data moves between the components comprising the cloud infrastructure.

Provenance systems [11], [12] concern audit; they assist in understanding the lifecycle of data (further details are given in §III-A): *how was it created? when? by whom? how was it manipulated?* As both provenance and IFC concern the flow of information between entities, IFC enforcement is a natural source of provenance-like data. The advantage of IFC with IFA compared with general provenance metadata collection, is that in IFC, audit data is a by-product of enforcement, whereby IFC audits only selected (labelled) entities. Further, as the audit data of IFA is intrinsically linked to the control mechanism (IFC), it readily assists policy management including the identification of policy errors.

Contributions: In this paper, we discuss our novel approach, combining IFC and provenance techniques to provide Information Flow Audit. We give an in-depth description of the technical implementation of the capture mechanism as a *Linux Security Module* and the simple user-space framework to handle the captured data. The data generated and collected during IFC enforcement can help in understanding system-wide behaviour. Further, we evaluate our mechanism in conjunction with open source and off-the-shelf tools, to allow audit to be performed on the collected data. Finally, we discuss related work and open challenges, giving some of the relevant literature on the subject. The combination of IFC and provenance techniques has the potential to demonstrate compliance with contracts, laws and regulations. Our current focus is within the context of PaaS, based on our implementation described in [7], [10]. In a wider context, such mechanisms will become increasingly relevant, given the emerging Internet of Things (IoT), which cloud services will be integral in realising [13].

In §II, we briefly describe our IFC model. In §III, we describe how audit data is generated during IFC and give examples of its usage. In §IV, we describe our prototype implementation and discuss our design choices. In §V, we evaluate our solution, discussing practical integration with graph visualisation and analysis frameworks. In §VI, we discuss related work, focusing on provenance and data flow management systems. In §VII we discuss open challenges, and conclude in §VIII.

II. Enforcing Information Flow Control

IFC systems provide guarantees over the secrecy and integrity dimensions of data, by controlling and restricting where in a system data is allowed to flow. An IFC security model was first introduced by Denning [14] and later refined for decentralised environments by Myers [15]. When implemented from the OS kernel level upwards, applications running under IFC enforcement do not need to be trusted for the data management policy to be properly enforced [16]. If the enforcement mechanism in the kernel and the mechanism interconnecting machines can be trusted, trust can be established across the whole distributed system [17]. If a Trusted Platform Module (TPM) is present, it is possible to remotely verify the presence of our IFC enforcement implementation using remote attestation [18], [19], further strengthening the trust relationship (this is discussed further in §IV-A).

We do not see IFC as a replacement for existing security schemes such as access control or encryption of sensitive data, but rather as a complement to provide guarantees when sharing sensitive data between parties or services. IFC simplifies the trust assumptions that need to be made. Instead of a need to trust all parties involved, only trust in the cloud provider implementing the IFC security mechanism is required. We now outline our IFC model, focussing on that relevant to the discussion in this paper. See [20] for a more formal definition.

A. Data Flow Constraints

In our model, entities, such as processes and data, are labelled with *secrecy* (S) and *integrity* (I) labels. A label comprises a set of tags, each of which represents some security concern such as secrecy:medical or integrity:verified. The *security context* of an entity is the state of its S and I labels and *domain* is used for entities that inhabit the same security context in the secrecy or integrity dimension.

A flow of information $A \rightarrow B$ is safe if and only if:

$$A \rightarrow B, \text{ iff } \{S(A) \subseteq S(B) \wedge I(B) \subseteq I(A)\}$$

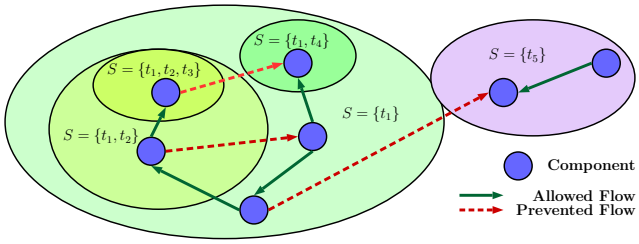

Fig. 1: End to end secrecy enforcement.

Fig. 1 describes the end-to-end behaviour of data flow, in the secrecy dimension. Data produced in a certain secrecy domain can only flow within the same domain or into a more restricted subdomain. This means that data produced by a component cannot be used for a purpose other than the one originally defined (e.g. medical data is only used within the medical security context domain).

Example—Secrecy: A person may have sensor devices for health and lifestyle monitoring. As the data streams from these sensors are highly sensitive, they can only flow into remote data storage labelled to receive data from that person.

Fig. 2 shows the end-to-end behaviour of data flow in the integrity dimension. Data can only flow within an integrity domain or towards a parent domain with a lower integrity guarantee. In practice, this means that the final data consumer can be assured that all the chain from the original data producer can be trusted.

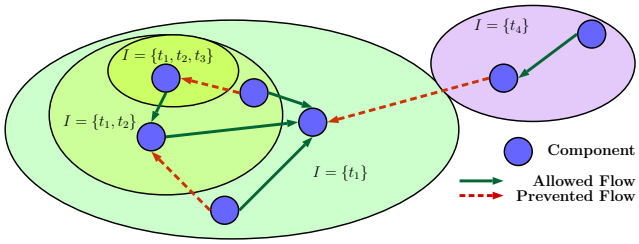

Fig. 2: End to end integrity enforcement.

Example—Integrity: Suppose that in health and lifestyle monitoring, analysis of the collected data indicates that the sampling rate should be increased. Generally, actuators establish a trust relationship with the entity issuing the actuation command. We argue that trust should encompass not only the immediate command issuer, but the data source and all the services (e.g. analytics) that were involved in influencing the actuation command. Integrity tags can capture such a notion, assuring the actuator that the source data and any transformation made on it are part of the trusted chain.

B. Endorsement and Declassification

In Figs. 1 and 2, we have shown how data flows are restricted to equal or increasing secrecy constraints and equal or decreasing integrity constraints. However, data may undergo transformations and/or checks that change its security properties. For example, moving data through an anonymisation engine renders the data less sensitive, so less strict secrecy constraints can apply to the anonymised output. In the integrity dimension, data may go through a validation process on input, thus becoming more trustworthy. Declassifiers and endorsers are the entities in the system that are trusted to change the security context of data. Declassifiers change the secrecy properties and endorsers change the integrity properties.

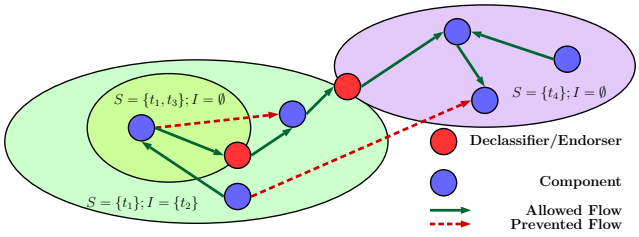

Fig. 3: Declassification and endorsement.

As shown in Fig. 3, declassifiers and endorsers are trusted entities that perform some operation on the data (e.g. analysis, transformation, etc.) and change its security context when the operation has executed successfully, transferring information across security context domains. These entities are allocated *privileges* that allow them to change their security context in order for data to be transferred from one domain to another. As mentioned, IFC allows untrusted applications to run on top of the enforcement mechanism, and declassifiers and endorsers may be small, tightly-scoped elements of a trusted computing

base that generally perform a one-way, well-defined task (e.g. encrypt, anonymise, etc.).

Endorsers and declassifiers can therefore be seen as trusted gateways between security context domains where the general IFC constraints would prohibit a direct flow. Such gateways, when accompanied by audit, can help ensure that regulation is enforced, e.g., medical data might only flow to a research domain if it has gone through a declassifier that applies a well defined anonymisation algorithm [5]. IFA can demonstrate that no other path exists (see §III). Similarly, regulations may indicate that medical data must be encrypted before being stored in databases. IFC labelling and endorsement can ensure compliance and IFA can demonstrate this compliance. See §III for more detail.

Some approaches to data management couple a rich policy specification with the enforcement regime (see §VI). IFC aims more generally to provide secrecy and integrity security primitives to bound data flows. Here, complexity and expressiveness emerge from the interaction of these primitives with the building blocks provided by the trusted declassifiers/endorsers. This avoids every entity in the system being burdened with the overheads associated with complex policy enforcement. That is, policy can be encoded in small endorser/declassifier services, separate from the application, that could be made available by the cloud provider or tenants. IFC constraints guarantee that these transformations are applied before data is allowed to flow between components of a system. The combination of IFC policy and declassifiers/endorsers allows the enforcement of policy such as *medical data stored in database X must have received proper consent and be anonymised* [5] or *European personal data sent to the US must first be anonymised* [21]. We are actively working on a higher-level mechanism to facilitate the translation of such concerns into IFC tags and the composition of cloud-provided security primitives.

C. Privileges and Entity Creation

We have so far considered two types of data flow mechanism: data flow constraints and security context change through endorsement/declassification. Declassifiers and endorsers are allocated privileges associated with specific tags in order to perform security context change operations. Such privileges are owned by the creator of a particular tag and can be passed to other entities in the system. Note that a change in security context does not itself result in a flow of information outside the entity, but still represents a recordable flow, as data has effectively been transferred (logically) from one security context to another through the entity's security context change.

A newly created entity inherits the security context of its parent. Though other implementations [16], [22] allow more flexibility, they modify system call semantics, and therefore require applications to be rewritten [10]. Creation of an entity represents a flow of information between the parent and the child entity. In summary, there are four types of flow within our model: **data flow**, **creation flow**, **security context change** and **privilege passing**.

III. Information Flow Audit (IFA)

Traditional cloud logging systems are mostly based on and composed of legacy and/or service-specific logging systems (OS, web-server, database etc.). These are difficult to interpret system-wide, as they tend to log only those events relevant to the particular system component. As such, it is argued that cloud logging systems should be redesigned to be information-centric (rather than system-centric), thus accounting for the movement of information [23]. Pohly et al. argue that forensic investigation requires the collection of data that captures the actions of processes, IPC mechanisms and the kernel [24].

IFC complements existing security mechanisms by providing guarantees about proper (i.e. policy-compliant) data usage. Our aim is to augment IFC with audit that makes visible how the data flows through the system and is used. This allows tenants to effectively demonstrate that proper mechanisms are in place and that all data goes through those mechanisms. If information has been shared when it should not have been, or this is claimed by some party, we aim to provide forensic data to understand how/whether this sharing happened.

A. Provenance Systems

Provenance systems concern audit, associating with each data object metadata describing the transformation involved in generating this data. They typically concern some aspects of: data quality, replication recipes, ownership attribution, context understanding and audit [25]. Provenance systems generally present the relationship between data objects and transformations (processes) as a directed graph leading to and from the data objects being audited. Such graphs capture when, why, by whom and how this data object was created and/or used and their processing allows such behaviour to be understood. In this work on IFA we leverage the graphs and processing tools that have been developed for establishing data provenance.

B. From Provenance to Information Flow Audit

IFC constrains the flow of information in a system, being enforced as system components interact. As such, information-centric logging is naturally provided by recording information flow decisions, metadata on the entities involved in the flow and any metadata associated with the decisions. This includes details of data exchanges (e.g. reading from a pipe or file, sending a message), process management operations (such as creating a new process and setting up its security context), and security operations such as declassification or endorsement. This covers the four types of flow described in §II.

The information produced by IFC enforcement can allow the generation of a provenance-like directed graph, answering the questions: *how, when, where and by whom* a piece of information was manipulated. This allows understanding of how a particular piece of information moved through the system infrastructure, across various components and services. Importantly, the tight coupling between the enforcement and audit mechanism facilitates understanding and verifying system behaviour and control policy. From this audit graph, it is possible to demonstrate compliance with data management policy and/or provide forensic data to determine the cause of any unintended data disclosure.

As described in Table I, audit entries can be divided into two main categories: *flows* (i.e. **edges** of the graph) and information about *entities* (i.e. describing the **nodes** of the graph). A node corresponds to [entity, security context], with a change in security context represented by a *security context change* flow towards a new node. Fig. 4, gives an example of how flow of information in the system can be represented. An edge entry is relatively simple: it describes the sender and receiver of a data flow, the type of flow, whether or not it was allowed, and an event identifier for allowing dependencies to be determined. Node entries contain metadata describing the

Nodes		Edges	
Attribute	Description	Attribute	Description
Entity ID	Unique local identifier of the entity.	Event ID	The ID of the event (e.g. a number, timestamp, etc.)
Machine ID	Unique identifier for the machine on which the node was recorded.	Machine ID	Unique identifier for the machine on which the edge (flow) was recorded.
Type	Type of node: e.g. process, FIFO, socket, file etc.	Type	Type of flow: data flow, privilege, creation, security context change.
Name	A name for the node (e.g. filename, executable name etc.).	Sender ID	The ID of the entity from which the data is flowing.
User ID	The user ID of the principal owning the entity.	Receiver ID	The ID of the entity to which the data is flowing.
IFC Labels	Secrecy and integrity label of the entity (and privileges for processes).	Allowed	If the flow was allowed or not.
Additional metadata	Node type specific or user space application-specified attributes.	Additional metadata	Edge type specific metadata (e.g. system call name).

TABLE I: The attributes of audit nodes (entities) and edges (flows).

node: its type (e.g. file, socket, process etc.), its ID etc. and again, an event ID so that dependencies can be determined.

C. Example: Discovering Data Disclosure Paths

As discussed, IFA can be represented in a directed graph. The graph can be analysed to 1) trace information flows within, across and between system components; and 2) to examine which components are attempting to violate IFC constraints.

For example, the IFA graph can be used to identify the origin of a data leak. Suppose that an information leak is suspected between different security context domains $[S, I]$ and $[S', I']$. Determining whether such a leak can occur is equivalent to discovering whether there is a path in the graph between the two contexts. If the leak occurred, there must be a path between some entity E such that $S(E) = S \wedge I(E) = I$ and another entity F such that $S(F) = S' \wedge I(F) = I'$.

The existence of such a path demonstrates that a leak is possible. To investigate whether a leak occurred it is essential to consider the event identifier associated with the edges comprising the path. We denote by e_l, the last incoming edge to the entity under investigation with labels $[S', I']$; only edges such that $e < e_l$ should be considered. When applied to all nodes along a path, this rule ensures strictly monotonically increasing event identifiers from the first node to the last.

Fig. 4, shows in pale blue a possible data disclosure path between the $[S', I']$ and $[S'', \emptyset]$ security context domains. We can see from the order of the event identifiers e_0 and e_1 that the data disclosure could not have occurred through file F_1 and process P_3, but occurred through P_1's security context change. P_1 wrote into the public security context domain (represented by a single node as flows are not tracked within this domain) and P_2 read from it at e_7. We present in §V-A how this analysis can be done in practice with our prototype implementation.

D. IFA compared with Provenance Logs

One of the problems of provenance systems is the extremely large amount of data being collected, rendering the approach impractical. Data is typically collected at the granularity of system calls [24], [26] (as in kernel-level IFC systems) but IFA records provenance-like metadata only on IFC labelled entities. “Public" entities are not audited as, from an IFC perspective, the information is not sensitive and therefore can flow freely. IFC essentially aims at labelling sensitive data, which is the data we need to keep track of in practice. IFC

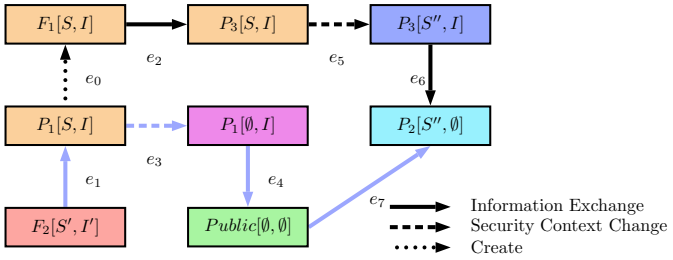

Fig. 4: Simplified audit graph from IFC OS execution (we omit metadata for readability). Blue/pale arrows show the path to disclosure.

audit can be seen as attaching policy metadata to sensitive entities, thus applying a policy filter to select the entities to be audited. In other words, our approach involves a tight coupling between the enforcement and provenance mechanism, which allows a large reduction in the amount of data collected. We argue that much provenance data is excess “background noise" generated by the system, and so is uninteresting and unrelated to the sensitive data that we aim to protect.

The granularity at which provenance is tracked via IFC audit depends on the IFC enforcement mechanisms employed; CamFlow entails OS object-level and message-level enforcement. Enforcing IFC in a database, for example, would require a specific database implementation, such as IFDB [27], where IFC would be enforced at a finer granularity than at the kernel-object level. Different levels of IFC enforcement can be made to interact gracefully, as in [22] or through means described in §IV-C. As IFC mechanisms are made to interoperate, an API should be provided for (internal) IFA to complement system-wide audit data. Similarly, the metadata collected will vary, according to the IFC enforcement mechanism(s), the applications involved, and higher-level provenance requirements [25].

IV. Implementation

We aimed to implement IFA in a modular, flexible fashion, giving freedom to system developers to implement or add different tools to fit their needs. The CamFlow framework provides the mechanism to enforce and audit *information flow* within the local machine, plus a fully fledged messaging middleware to deal with *information flow* across machines. In addition, APIs allow a cloud provider to build additional

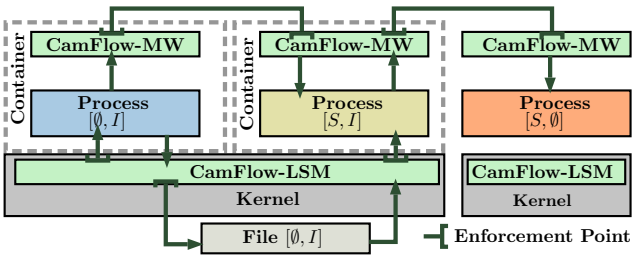

Fig. 5: High-level view of CamFlow use for PaaS.

trusted components (see §IV-C), for example, to link applications' processes to an IFC-enforcing key-value store.

As described in [7] and shown in Fig. 5, the CamFlow framework can be used to underlie current PaaS architectures. Our current prototype cloud derives from Dokku[2] and hosts cloud services within Docker [28] containers. A local machine's IFC&A are provided as part of the underlying kernel (see §IV-B), and other cloud-provided trusted components (such as the messaging middleware or other custom components) can be managed through the CamFlow API.

A. Trust Assumptions

The following assumptions were made when building our IFC enforcement and audit mechanism:

Hardware Integrity: We assume that the cloud providers have taken sufficient technical and non-technical measures to ensure that the hardware has not been tampered with.

Physical Security: We assume that best practices are in place on physical access to hardware, when managed by the cloud provider or by a third party managing the underlying infrastructure [29].

Low-level software stack: We assume that the integrity of the low-level software stack is recorded and monitored, which includes BIOS/UEFI, boot loader code and configuration, Options ROM, host platform configuration, virtualisation hypervisor etc. We assume that such integrity measurements are kept safe through a hardware mechanism and cannot be tampered with.

Trusted Platform Module (TPM): We assume that TPM or vTPM [30] features are leveraged to guarantee the integrity of the platform on top of which cloud hosted applications and service are running. We further assume, that such configurations could eventually be monitored in real-time [31] using remote attestation [18] to ensure that our security mechanism is in place at all times and is correctly configured. Hardware-assured software is relatively new for cloud services, and further work is needed, e.g., to consider issues such as continuous assessment. Without this, attack analysis may suffer from the disparity between time-of-attack and time-of-detection.

Cryptographic Security: We assume cryptographic functions to be secure and data exchange across machines to be encrypted. We assume that message integrity on exchange between machines can be verified. Furthermore, data may be encrypted on disc to provide further guarantees.

In this work we assume that Cloud Service Providers that manage the underlying infrastructure can and should be trusted as they are 1) bound by contract and regulations [6], 2) it is in their best economic interest to ensure proper security measures and 3) they are in practice already trusted by large numbers of cloud tenants.

[2]https://github.com/progrium/dokku

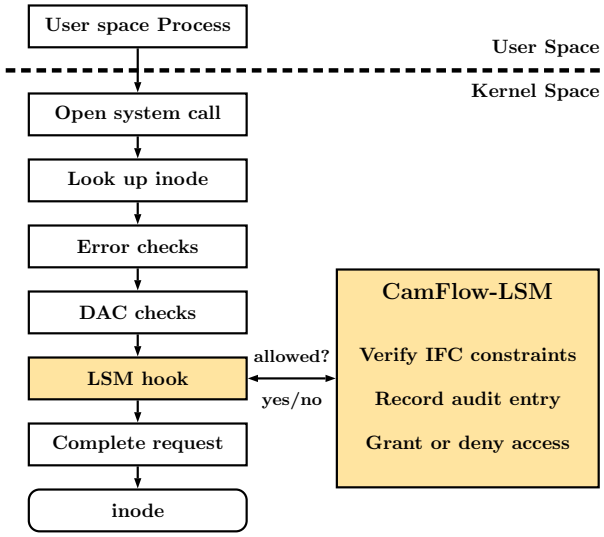

Fig. 6: Linux Security Services Module hooks example on open system call.

B. Local Information Flow Control and Audit

The local enforcement builds upon a Linux Security Module (LSM) [32], a general framework that allows a variety of different access control models to be implemented into the Linux OS. SELinux [33] and AppArmor [34] are examples of two well known mandatory access control implementations as LSMs. The LSM framework has also been used beyond access control e.g. to include Provenance within Linux [24].

The LSM framework calls security hooks when access to a kernel object is attempted, as shown in Fig. 6. Security metadata can be associated with kernel objects and is used by the LSM module to make access decisions. In order to implement IFC, we associate IFC labels and privileges with kernel objects [26].

We assume that the rest of the kernel can be trusted and does not interfere with the IFC/IFA enforcement mechanism. LSM system hooks have been statically and dynamically verified [35], [36], [37], and our implementation inherits from LSM the formal assurance of IFC's correct placement on the path to any controlled kernel object. This is sufficient to guarantee that we control flow and record audit on any operation on a controlled kernel object.

CamFlow-LSM also provides an API for processes (the active entities) to manipulate their security context dynamically at run-time (assuming the process holds the requisite privileges). Additional features are not part of our LSM but rather, are implemented as user space helpers (*ushers*).[3] This allows a greater modularity of the system and customisation to meet particular needs.

Audit-usher: The kernel records data flows between kernel objects and the metadata on those objects. These audit entries are then read by an audit-usher. The usher's role is to translate the raw and binary data provided by the kernel into human/-machine readable log data.

A system developer wanting to implement a custom audit-usher needs to implement the callbacks illustrated in Listing 1. The underlying concerns (access to log data, threading etc.)

[3]An usher is an official in a court of law that ensures secure transactions on documents and escorts participants to the courtroom.

```c
/* callback to handle edge */
void log_edge(edge_t* e);
/* callback to handle label node metadata */
void log_label(meta_label_t* l);
/* callback to handle string metadata */
void log_str(meta_str_t* s);
/* callback to handle node */
void log_node(node_t* n);
/* callback for filer function */
bool filter(byte_t* raw);

audit_op_t op = {
  .log_edge = log_edge,
  .log_label = log_label,
  .log_str = log_str,
  .log_node = log_node,
  .filter = filter // set to NULL if no filter
};

int main(void){
  register_audit(&op, 4); // register callbacks and
    number of worker threads
  /* do whatever */
  stop_audit(); // stop audit recording
  return 0;
}
```

Listing 1: CamFlow Audit-usher API.

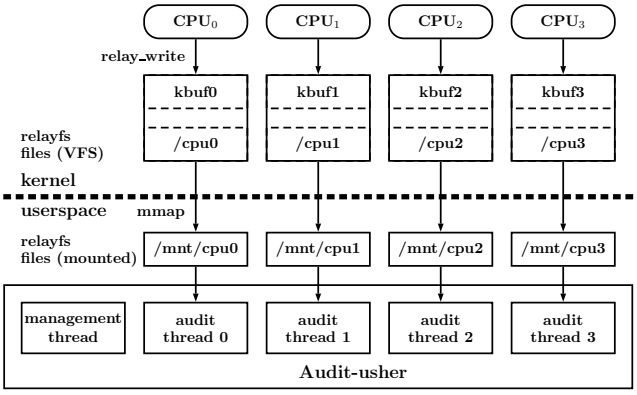

Fig. 7: Audit-usher leveraging relayfs.

are handled transparently by the CamFlow library. The current implementation relies on relayfs [38].

Relayfs provides a per-CPU kernel buffer that can be efficiently written to by kernel code and read from user space. These buffers are represented as files that the audit-usher can `mmap` and read efficiently from user space. Relayfs has been designed to provide the simplest possible mechanism to read and log large amount of data by relaying them from kernel to user space. The architecture is illustrated in Fig. 7.

Customisation of the audit collection allows the cloud provider to tailor the mechanism to its needs. For example, one may want to format the data in accordance with the *Open Provenance Model* [39], feed the data to a graph database, use a graph processing framework to perform real-time event detection, etc. Our proposed implementation does not constrain developers into a particular usage pattern, and while dealing transparently with the underlying mechanisms, allows them to focus on the aspects relevant to them. We describe an example implementation to graphically display an IFA graph through a web interface or to feed the information to a graph database in §V-A.

System Objects: Processes are the only active entities within the Linux OS. Each process is associated with IFC labels and privileges at creation and assigned a unique ID within the current boot (boot and machine are also allocated unique IDs). A process and its memory are treated as a black box. `Fork` generates a create flow from the current process to the forked process. `Exec` creates a data flow from the file being executed to the calling process.

Files, pipes, sockets etc. fall under the *inode* category within the kernel. They are passive entities and their security context is immutable. Creation, read and write from those entities are protected by IFC policies and flows are recorded.

Files need to be identified as they persist across boots. A file inode ID is unique within its file system and a file system is generally associated with a unique identifier at creation. We generate unique identifiers for kernel-internal pseudo-filesystems. The combination of inode ID and filesystem ID allows files to be identified uniquely within our audit logs. Sockets and pipes can be identified in the same fashion as, from the kernel perspective, they are inodes that belong to pseudo-filesystems.

Messages in message queues are handled individually and each message represents a unique node in the audit graph. As they have no kernel source of identifier, their IDs are generated by CamFlow-LSM.

Finally, a file mapped to an address space or shared memory does not provide fine grained read/write semantics from the audit perspective. We can only enforce and record flows when mapping is established. However, to prevent such a mechanism leaking data across security contexts, once memory has been mapped (in read/write/both mode), the security context of the associated process is frozen. We conservatively assume that any data accessed by a process mapped to this shared memory flows to other mapped processes. Again, the underlying mechanism relies on filesystems or pseudo-filesystems, which can be used to uniquely identify nodes within the audit graph.

C. Bridging with Other Layers of Enforcement

IFC can be implemented across different layers of the software stack, for example, within applications [22] or within database systems [27]. The CamFlow framework provides a mechanism to bridge from an IFC-constrained process to a trusted process enforcing IFC at a different layer of abstraction (our messaging middleware is such a process and is described in more detail in §IV-D). A system developer can implement such a mechanism to build more complex systems.

CamFlow associates a bridge-usher process with a constrained process and allows communication through a standard socket interface. This usher process can perform operations outside the IFC constraints applied to the constrained process. Data sent through the kernel socket is forwarded by CamFlow-LSM from the constrained process to its associated bridge-usher (and *vice versa*). Such messages are recorded and associated with a unique identifier by the CamFlow-LSM module, i.e. logging the flow of information between the bridge-usher and its attached process. As there is "layered" IFC, it is possible to provide layered audit data [40]. An API allows a bridge-usher to generate an audit subgraph of its internal behaviour and allows incoming or outgoing messages' nodes to be connected to this subgraph. This complements the system's *observed* provenance, by *disclosed* provenance from applications [41], which provides richer semantic knowledge and allows a better understanding of the system. For example, in the case of a

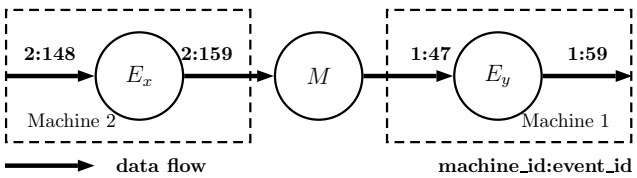

Fig. 8: Communication through an inter-machine message. Partial order along the path: $2:159 \precsim 1:47$.

database, this could be providing details of information flow in relation to database objects.

Integration with a system natively supporting IFC is trivial, since IFA is a simple by-product of IFC enforcement. One possible approach to integrate IFC&A with a legacy solution is to use aspects for instrumentation. We used aspect-oriented programming to incorporate IFC into web applications [42], and in [43], aspects are used to track data flows in MapReduce/HDFS. Our solutions provide the API to insert the data collected at that level into the whole system graph, but integration is beyond the scope of current work.

D. Inter-machine Enforcement

Only processes P such that $S(P) = \emptyset$ (i.e. not subject to security constraints) are allowed to directly connect or receive messages from outside connections (e.g. through a socket). In order to connect directly to the outside world, a process must either: 1) be able to declassify to $S = \emptyset$; or 2) communicate through a bridge-usher.

A bridge-usher is used to integrate our messaging substrate, CamFlow-MW (see [10] and [44]), that handles cross-machine communication. Each communicating process has an associated CamFlow-MW process to manage its messaging, through the kernel-mediated socket discussed earlier. The substrate then enforces IFC in its dealings with the substrate processes of other applications (local or remote), ensuring that the tags on each side accord. This enforcement occurs on the establishment of communication (messaging) channels between components, where a channel is only established if the IFC policy allows. This is monitored throughout the connections' lifetime, where a change in security context triggers re-evaluation: a channel is terminated if it is no longer authorised. All of these operation are recorded to generate an audit-graph.

In §III, we discussed how the ordering of event identifiers is used to understand the succession of events within the audit graph. Once the system is distributed, one may be tempted to introduce a complex synchronisation scheme to maintain this ordering. We believe this is not necessary and should be avoided for simplicity and runtime performance. Indeed, the flow of sensitive data is allowed across machines only through limited IFC-aware communication channels with well-understood semantics. This creates a partial order of events across machines, which is sufficient to order events along *any* given path. For example, as shown in Fig. 8, in the case of cross-machine message passing, all writes at the message sending entity/node happened before any read on the destination machine.

V. Evaluation

A. Using the Framework for Information Flow Audit

In order to evaluate the usefulness and usability of CamFlow *Information Flow Audit* we now show that the collected

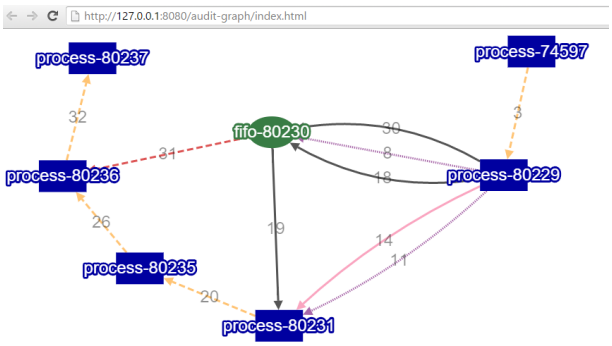

Fig. 9: Example screen-shot of a small audit sub-graph. Edges' key: orange/dashed—security context change; purple/dotted—creation flow; pink/light—privilege passing; black/plain—allowed data-flow; red/dashed—disallowed data-flow.

data can provide useful insights and can easily be integrated with existing tools. We demonstrate the feasibility of our approach through two simple *audit-usher* prototypes that connect to open-source graph visualisation tools and graph databases respectively. We selected the open-source Cytoscape tool [45] for visualisation and Neo4J[4] for the graph database as they have previously been used in a provenance context [46], [47], [48], [49] and can therefore be considered as realistic solutions. The code base is small and relies on off-the-shelf libraries and tools. We have shown that it is possible to generate an audit graph from IFC enforcement and now discuss the use of this data by means of a graph analysis framework.

Visualising Data Flow: In order to evaluate the feasibility of our approach we built a small tool that reads raw data from the kernel and formats it to generate an audit file. The audit usher application is very simple, comprising fewer than 100 lines of C code. These log files are then parsed by a Ruby script that builds a graph description in JSON that can be visualised through the Cytoscape tool.

Fig. 9 presents a sub-graph generated from IFA logs. Nodes are labelled with the tuple {type-id}, where the id uniquely identify the pair {object, security-context}. The following events are represented in the graph: edge 8 shows a parent process process-80229 creating a pipe fifo-80230 and, down edge 11, a child process process-80231. The parent passes privileges (edge 14) to the child and writes to the pipe (edge 18) then the child reads from it (19). The child changes its security context (20, 26) and its process ID. Finally, the parent writes to the pipe (30) but the child process-80236 can no longer read from it (31) due to incompatible security contexts.

Analysing the Audit Graph: Our second experiment with the Information Flow Audit aspect of our framework consisted of pushing data into a graph database in order to perform query and analyses of system behaviour. The implementation of the audit-usher has a small code footprint and easily allows the well-established Neo4J graph database to be leveraged.

Listing 2 presents a single-machine query using the Cypher query language.[5] This solves in practice the example presented in §III. The query searches for all paths between a node in the medical domain to a node in the public domain on machine 1234. The results are a collection of nodes and edges representing the paths between the nodes. These paths can

[4] http://neo4j.com/

[5] http://neo4j.com/developer/cypher-query-language/

```
// Find all paths on machine "1234" from medical to public
MATCH p =(:Entity {machineid:1234, secrecy: "medical" })-[:FLOW*]->(:Entity {machineid:1234,
    secrecy: "public" })
// Restrict to path with monotonically increasing flow event ids
WITH p, range(0,length(p)-2) AS idx, relationships(p) AS rs
WHERE ALL (i IN idx
           WHERE (rs[i]).eventid <(rs[i+1]).eventid)
RETURN p;
```

Listing 2: Query (simplified) to find all paths from medical to public.

be used to generate subgraphs that represent the transfer of information between the two security contexts' domains. The query presented here, for simplicity, does not deal with node-specific semantics (e.g. shared memory) and is restricted to a single machine (see §IV). Such considerations can be either 1) encoded within a more complex query by extending the where clause to deal with entity-specific semantic, or 2) managed through Neo4J's traversal API.[6]

Compliance with obligations can be demonstrated through queries over the graph. For example, the plain English policy example given in §II-B:
– *"Medical data stored in database X must have received consent and be anonymised"* [5] can be expressed as a query verifying that there is no path between *medical* labelled data and the database without a consent and anonymiser process;
– *"European personal data sent to the US must be anonymised"* [21] is equivalent to writing a query that verifies that there is no path between EU and US labelled data without an encryption process.

In practice, further human input may be required to investigate data leakage or compliance. This is reasonable given that flow policy will be specified by users, another advantage of coupling enforcement and provenance. The subgraphs generated by a query for a disclosure path may be visualised as described above. In addition, other types of query can be performed over the audit graph such as determining how a particular piece of data has been generated, determining ownership in case of dispute, understanding the cause of a confidentiality breach etc. Exploitation of the type of data we collect creates many opportunities for forensics and demonstration of compliance.

B. Performance

We tested the CamFlow-LSM module on Linux Kernel version 4.1.5 (08/2015) from the Fedora distribution. The tests were run on an Intel 2.6Ghz i7 CPU and 8GiB RAM machine.

Measurements are done using the Linux tool ftrace [50] to provide a microbenchmark. Two processes read from and write to a pipe respectively. Each has 20 tags in its security label, substantially more than we have seen a need for in current use cases. We measure the overhead induced by: creating a new process (sys_clone), creating a new pipe (sys_pipe), writing to the pipe (sys_write) and reading from the pipe (sys_read). The results are given in Fig. 10.

We can distinguish two types of induced overhead on core CamFlow IFC enforcement: verifying an IFC constraint (sys_read, sys_write) and allocating labels (sys_clone, sys_pipe). The sys_clone overhead is roughly twice that of sys_pipe as memory is allocated dynamically for the active

[6]http://neo4j.com/docs/stable/tutorial-traversal-java-api.html

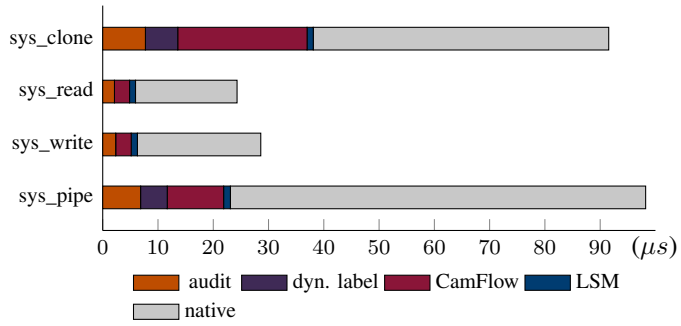

Fig. 10: Overhead introduced into the OS by the CamFlow LSM.

entity's labels and privileges. Note that passive entities have no privileges (§IV-B). Audit on creation of a new entity is more costly as, in addition to the flow being logged, the new entity and its associated metadata are logged. Overhead measurements for other system-calls/data-structures are essentially identical, as they rely on the same underlying enforcement mechanism, and are not presented.

The CamFlow-LSM overhead is a few percent, see Fig. 10. We provide a build option that further improves performance by declaring labels and privileges with a fixed size (by default, label size can increase dynamically to meet application requirements). This reduces the overhead of the system calls that create new entities.

The overhead on system calls is in line with IFC [22] or provenance [24], [26] systems that operate at OS level. For most applications, the overhead imposed is minimal and hard to measure; the variation between two executions due to system noise is often greater than the overhead. On kernel compilation, which evaluates a typical combination of process execution and file manipulation, we incur an overhead of 3.6% compared[7] with 2.5% [24], 2.7% [26] (but these systems only deal with provenance).

VI. Related Work

PASS [51] collects data within the Linux OS, mostly concerning the file system, recording relationships between processes and files, but does not capture whole-system data flows. Hi-Fi [24] collects whole-system provenance data at the kernel level leveraging the LSM framework. Macko et al. [52], also proposed the collection of provenance data in the Xen [53] hypervisor, capturing provenance information from the host VM without requiring its modification. As discussed in §III-B, the amount of data collected by a provenance system is hard to manage and collection should be limited, based on the

[7]Here the values are as reported in their respective publications. Note that the kernel versions are different from ours, namely 3.2.15 (Arch Linux) and 2.6.32 (RedHat) respectively.

policy in place. Bates et al. [54] used SELinux policy to reduce the amount of data collected by Hi-Fi. Expanding on this work, Bates et al. [26], propose a Linux Provenance Module providing hooks akin to LSM, but for the specific purpose of provenance data collection and enforcement of Provenance-Based Data Loss Prevention—i.e. preventing sensitive data from leaving a corporate domain—policy that can easily be expressed in IFC. Provenance checking is easily incorporated as part of IFC by enabling IFA. Coupling the enforcement and audit mechanisms enables more concise logs (i.e. corresponding to labelled flows). This facilitates the management of audit logs, and at the same time, more readily indicates issues in policy specification and/or enforcement.

Taint tracking (TT) systems (e.g. [55], [56]) entail the recording of data flows. TT is used to determine whether an application has accessed *tainted data*: based on secrecy (data sensitivity) or integrity (trust in data sources) concerns. TT systems only *enforce* policy (check for this taint) at specified *sink* points, e.g., at the point where the results of a MapReduce procedure are to be output. In contrast, IFC enforces policy on every data flow within a system. In TT systems, there is likely to be delay between some 'problematic' event (i.e. a violation of data flow policy) and its detection at a sink point—meaning there is scope for applications to operate on tainted data. We argue that TT can be equated to forming a query on the existence of a path between the *source* and the *sink*, therefore, potentially subsumed by our work (e.g. is the data written to a destination derived from sensitive information?).

Akoush et al. [57] demonstrated that it is possible to verify IFC constraints in MapReduce *a posteriori* from provenance records. This approach is conceptually similar to TT and suffers from the same gap in time between a 'problematic' event and its time of detection, requiring re-execution of all or some MapReduce jobs.

VII. Open Challenges

So far we have described and practically demonstrated how IFC can naturally lead to provenance-like capabilities. However, a number of open challenges remain.

Controlling access to audit data: Audit data may be sensitive and controlling access to provenance data is challenging [58]. A number of different languages have been proposed to model provenance access control (PAC) [59], [60], but there is no well-accepted standard nor model.

The need for storage: Managing the size of IFA data is a significant challenge, especially since every flow in the system may potentially be recorded. As audit data is represented in a tree structure, pruning techniques [51] can be used to 'garbage collect', i.e. compress or delete audit data that is unnecessary, such as that irrelevant to an application's context. However, it is important to ensure that relevant data is not deleted. Machine learning could be used to help filter and record only that audit data relevant to the situation.

Data visualisation and analysis: In §V-A we showed how IFA data can be visualised. As our focus is on IFA in PaaS clouds, we aimed to provide information relevant to system architects. More work is needed on representing data relevant to application end-users and auditors.

IFA data can be considered 'big data', meaning that big data analytics techniques such as those based on machine learning can be used to infer meaning from the audit data. Such approaches already exist in intrusion detection systems [61].

Widely distributed audit: Our work has considered the cloud provider acting as the trusted policy enforcer and audit data collector. Future work involves considering provenance across cloud boundaries, e.g. to other clouds, IoT devices, etc. This requires means for trust management—we discuss the use of hardware-based roots of trust in [19]—and also tools for distributed provenance analyses [62].

VIII. Conclusion

In this paper, we outlined how IFC can be enforced in the cloud and discussed how audit data can be collected as an intrinsic part of IFC. By leveraging tools developed for provenance systems, we demonstrated that the collected data can have practical use investigating system behaviour, and in demonstrating compliance with data management obligations.

Acknowledgement

This work was supported by UK Engineering and Physical Sciences Research Council grant EP/K011510 CloudSafetyNet: End-to-End Application Security in the Cloud. We acknowledge the support of Microsoft through the Microsoft Cloud Computing Research Centre. Thanks to S. Akoush and K. R. Jayaram for their comments.

References

[1] M. Bellamy, "Adoption of Cloud Computing services by public sector organisations," in *World Congress on Services*. IEEE, 2013, pp. 201–208.

[2] R. F. El-Gazzar, "A Literature Review on Cloud Computing Adoption Issues in Enterprises," in *Creating Value for All Through IT*. Springer, 2014, pp. 214–242.

[3] C. Zhang, J. Sun, X. Zhu, and Y. Fang, "Privacy and security for online social networks: challenges and opportunities," *Network, IEEE*, vol. 24, no. 4, pp. 13–18, 2010.

[4] Z. Papacharissi and P. Gibson, "Fifteen minutes of privacy: Privacy, sociality, and publicity on social network sites," in *Privacy Online*. Springer, 2011, pp. 75–89.

[5] J. Singh, J. Powles, T. Pasquier, and J. Bacon, "Data Flow Management and Compliance in Cloud Computing," *IEEE Cloud Computing Magazine, SI on Legal Clouds*, 2015.

[6] C. J. Millard, Ed., *Cloud Computing Law*. Oxford University Press, 2013.

[7] T. F. J.-M. Pasquier, J. Singh, and J. Bacon, "Information Flow Control for Strong Protection with Flexible Sharing in PaaS," in *IC2E, International Workshop on Future of PaaS*. IEEE, 2015.

[8] K. Singh, S. Bhola, and W. Lee, "xBook: Redesigning Privacy Control in Social Networking Platforms," in *Security Symposium*. USENIX, 2009, pp. 249–266.

[9] N. Kumar and R. Shyamasundar, "Realizing Purpose-Based Privacy Policies Succinctly via Information-Flow Labels," in *Big Data and Cloud Computing (BDCloud'14)*. IEEE, 2014, pp. 753–760.

[10] T. Pasquier, J. Singh, D. Eyers, and J. Bacon, "CamFlow: Managed Data-Sharing for Cloud Services," *IEEE Transactions on Cloud Computing*, 2015.

[11] A. Chapman, M. D. Allen, and B. T. Blaustein, "It's About the Data: Provenance as a Tool for Assessing Data Fitness." in *Workshop on the Theory and Practice of Provenance*. USENIX, 2012.

[12] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Selter, and A. Hopper, "A primer on provenance," *Communications of the ACM*, vol. 57, no. 5, pp. 52–60, 2014.

[13] J. Singh, T. Pasquier, J. Bacon, H. Ko, and D. Eyers, "Twenty security considerations for cloud-supported Internet of Things," *IEEE Internet of Things Journal*, 2015.

[14] D. E. Denning, "A lattice model of secure information flow," *Communication of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[15] A. C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 1997, pp. 129–142.

[16] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information Flow Control for Standard OS Abstractions," in *Symposium on Operating Systems Principles*. ACM, 2007, pp. 321–334.

[17] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing Distributed Systems with Information Flow Control," in *5th USENIX Symposium on Networked System Design and Implementation*, 2008, pp. 293–308.

[18] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence," in *Dependable Systems & Networks (DSN'09)*. IEEE, 2009, pp. 115–124.

[19] T. F. J.-M. Pasquier, J. Singh, and J. Bacon, "Clouds of Things need Information Flow Control with Hardware Roots of Trust," in *International Conference on Cloud Computing Technology and Science (CloudCom'15)*. IEEE, 2015.

[20] T. F. J.-M. Pasquier, J. Singh, J. Bacon, and O. Hermant, "Managing Big Data with Information Flow Control," in *International Conference on Cloud Computing (CLOUD)*. IEEE, 2015.

[21] T. Pasquier and J. Powles, "Expressing and Enforcing Location Requirements in the Cloud using Information Flow Control," in *IC2E International Workshop on Legal and Technical Issues in Cloud Computing (CLaw'15)*. IEEE, 2015.

[22] D. E. Porter, M. D. Bond, I. Roy, K. S. McKinley, and E. Witchel, "Practical Fine-Grained Information Flow Control Using Laminar," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 1, p. 4, 2014.

[23] R. K. Ko, M. Kirchberg, and B. S. Lee, "From System-centric to Data-centric Logging-accountability, Trust & Security in Cloud Computing," in *Defense Science Research Conference and Expo (DSR), 2011*. IEEE, 2011, pp. 1–4.

[24] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting High-Fidelity whole-system provenance," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 259–268.

[25] Y. L. Simmhan, B. Plale, and D. Gannon, "A Survey of Data Provenance in e-Science," *ACM SIGMOD Record*, vol. 34, no. 3, pp. 31–36, 2005.

[26] A. Bates, D. Tian, K. Butler, and T. Moyer, "Trustworthy Whole-System Provenance for the Linux Kernel," in *Proceedings of 24th USENIX Security Symposium on USENIX Security Symposium*, 2015.

[27] D. Schultz and B. Liskov, "IFDB: Decentralized Information Flow Control for Databases," in *European Conference on Computer Systems (Eurosys'13)*. ACM, 2013, pp. 43–56.

[28] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing Magazine*, no. 3, pp. 81–84, 2014.

[29] Cloud Security Alliance, "Security guidance for critical areas of focus in cloud computing v3.0," 2011, accessed: 13th January 2016. [Online]. Available: https://cloudsecurityalliance.org/research/security-guidance/

[30] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *Security Symposium*. USENIX, 2006, pp. 305–320.

[31] S. Berger, K. Goldman, D. Pendarakis, D. Safford, E. Valdez, and M. Zohar, "Scalable Attestation: A Step Toward Secure and Trusted Clouds," in *International Conference on Cloud Engineering (IC2E)*. IEEE, 2015.

[32] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux Security Modules: General security support for the Linux kernel," in *Foundations of Intrusion Tolerant Systems*. IEEE, 2003, pp. 213–213.

[33] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux Security Module," *NAI Labs Report*, vol. 1, p. 43, 2001.

[34] M. Bauer, "Paranoid Penguin: an Introduction to Novell AppArmor," *Linux Journal*, vol. 2006, no. 148, p. 13, 2006.

[35] A. Edwards, T. Jaeger, and X. Zhang, "Runtime verification of authorization hook placement for the Linux Security Modules framework," in *Conference on Computer and Communications Security*. ACM, 2002, pp. 225–234.

[36] T. Jaeger, A. Edwards, and X. Zhang, "Consistency analysis of authorization hook placement in the Linux security modules framework," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 2, pp. 175–205, 2004.

[37] V. Ganapathy, T. Jaeger, and S. Jha, "Automatic placement of authorization hooks in the Linux security modules framework," in *Conference on Computer and Communications Security*. ACM, 2005, pp. 330–339.

[38] T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore, and M. Dagenais, "relayfs: An efficient unified approach for transmitting data from kernel to user space," in *Linux Symposium*, 2003, p. 494.

[39] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers *et al.*, "The Open Provenance Model Core Specification (v1. 1)," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, 2011.

[40] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. W. Margo, M. I. Seltzer, and R. Smogor, "Layering in provenance systems." in *USENIX Annual technical conference*. USENIX, 2009.

[41] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Issues in automatic provenance collection," in *Provenance and annotation of data*. Springer, 2006, pp. 171–183.

[42] T. F. J.-M. Pasquier, J. Bacon, and B. Shand, "FlowR: Aspect Oriented Programming for Information Flow Control in Ruby," in *International Conference on Modularity*. ACM, 2014.

[43] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, 2015.

[44] J. Singh, T. Pasquier, J. Bacon, and D. Eyers, "Integrating Middleware and Information Flow Control," in *International Conference on Cloud Engineering (IC2E)*. IEEE, 2015, pp. 54–59.

[45] M. E. Smoot, K. Ono, J. Ruscheinski, P.-L. Wang, and T. Ideker, "Cytoscape 2.8: New features for data integration and network visualization," *Bioinformatics*, vol. 27, no. 3, pp. 431–432, 2011.

[46] P. Chen, B. Plale, Y. Cheah, D. Ghoshal, S. Jensen, and Y. Luo, "Visualization of network data provenance," in *International Conference on High Performance Computing (HiPC)*. IEEE, 2012, pp. 1–9.

[47] P. Chen and B. A. Plale, "Big data provenance analysis and visualization," in *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE/ACM, 2015, pp. 797–800.

[48] G. Tylissanakis and Y. Cotronis, "Data provenance and reproducibility in grid based scientific workflows," in *Grid and Pervasive Computing Conference*. IEEE, 2009, pp. 42–49.

[49] S. Woodman, H. Hiden, P. Watson, and P. Missier, "Achieving reproducibility by combining provenance with service and workflow versioning," in *workshop on Workflows in support of large-scale science*. ACM, 2011, pp. 127–136.

[50] T. Bird, "Measuring Function Duration with ftrace," in *Japan Linux Symposium*, 2009, pp. 47–54.

[51] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *USENIX Annual Technical Conference*, 2006, pp. 43–56.

[52] P. Macko, M. Chiarini, and M. Seltzer, "Collecting Provenance via the Xen Hypervisor," in *TaPP*. USENIX, 2011.

[53] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[54] A. Bates, K. R. Butler, and T. Moyer, "Take only what you need: leveraging mandatory access control policy to reduce provenance storage costs," in *Conference on Theory and Practice of Provenance*. USENIX, 2015, pp. 7–7.

[55] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Conference on Operating systems design and implementation (OSDI'10)*. USENIX, 2010, pp. 1–6.

[56] C. Priebe, D. Muthukumaran, D. O'Keeffe, D. Eyers, B. Shand, R. Kapitza, and P. Pietzuch, "CloudSafetyNet: Detecting Data Leakage between Cloud Tenants," in *Cloud Computing Security Workshop*. ACM, 2014.

[57] S. Akoush, L. Carata, R. Sohan, and A. Hopper, "MrLazy: Lazy Runtime Label Propagation for MapReduce," in *6th Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2014.

[58] U. Braun, A. Shinnar, and M. I. Seltzer, "Securing provenance," in *HotSec*. USENIX, 2008.

[59] Q. Ni, S. Xu, E. Bertino, R. Sandhu, and W. Han, "An access control language for a general provenance model," in *Secure Data Management*. Springer, 2009, pp. 68–88.

[60] T. Cadenhead, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham, "A language for provenance access control," in *Proc. 1st ACM Conference on Data and Application Security and Privacy*, 2011, pp. 133–144.

[61] C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin, "Intrusion detection by machine learning: A review," *Expert Systems with Applications*, vol. 36, no. 10, pp. 11 994–12 000, 2009.

[62] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *Proc. 13th ACM/IFIP/Usenix Middleware*. Springer, 2012, pp. 101–120.