CamFlow: Managed data-sharing for cloud services

Thomas F. J.-M. Pasquier, *Member, IEEE,* Jatinder Singh, *Member, IEEE,* David Eyers, *Member, IEEE* and Jean Bacon *Fellow, IEEE,*

Abstract—A model of cloud services is emerging whereby a few trusted providers manage the underlying hardware and communications whereas many companies build on this infrastructure to offer higher level, cloud-hosted PaaS services and/or SaaS applications. From the start, strong isolation between cloud tenants was seen to be of paramount importance, provided first by virtual machines (VM) and later by containers, which share the operating system (OS) kernel. Increasingly it is the case that *applications* also require facilities to effect *isolation and protection* of data managed by those applications. They also require *flexible data sharing* with other applications, often across the traditional cloud-isolation boundaries; for example, when government, consisting of different departments, provides services to its citizens through a common platform.

These concerns relate to the management of data. Traditional access control is application and principal/role specific, applied at policy enforcement points, after which there is no subsequent control over where data flows; a crucial issue once data has left its owner's control by cloud-hosted applications and within cloud-services. Information Flow Control (IFC), in addition, offers system-wide, end-to-end, flow control based on the properties of the data. We discuss the potential of cloud-deployed IFC for *enforcing* owners' data flow policy with regard to protection and sharing, as well as safeguarding against malicious or buggy software. In addition, the audit log associated with IFC provides transparency and offers system-wide visibility over data flows. This helps those responsible to meet their data management obligations, providing evidence of compliance, and aids in the identification of policy errors and misconfigurations. We present our IFC model and describe and evaluate our IFC architecture and implementation (CamFlow). This comprises an OS level implementation of IFC with support for application management, together with an IFC-enabled middleware.

Keywords—Compliance, Security, Audit, Cloud Computing, Information Flow Control, Middleware, PaaS

1 Introduction and Motivation

A model of cloud services is emerging whereby a few trusted providers manage the underlying hardware and communications infrastructure—datacenters with worldwide replication to achieve high data integrity and availability at low latency. Many companies build on this infrastructure to offer higher level cloud services, for example Heroku is a PaaS built on Amazon's EC2, above which SaaS offerings can be built (e.g. the LIFX smart lightbulb cloud service on top of the Heroku platform). From the start, protection was a paramount concern for the cloud as infrastructure is shared between tenants. Strong tenant isolation was provided by means of totally separated virtual machines (VMs) [1], [2] and more recently, isolated containers have been provided that share a common OS kernel [3].

Increasingly, cloud-hosted applications may need not only protection (and isolation) from other applications but also have requirements for *flexible data sharing*, often across VM and container boundaries. An example is the UK GCloud[1] initiative, a government platform designed to encourage small companies to provide cloud-hosted applications. These applications need to be composed and made to interoperate to support citizens' needs for online services. Similarly, the Massachusetts Open Cloud [4] is a marketplace (Open Cloud Exchange (OCX)) to encourage small business development. Solutions are open and one may build on the services of another. The aim is to create a catalyst for the economic development of business clusters.

End-users of cloud services still need to be assured that their data is protected from leakage to other parties by their cloud hosts, due to software bugs or misconfigurations, also safeguarded to the extent possible against insider attacks and external threats. But increasingly, they also need to be able to access their own data across applications and to share their data with others, according to the policies they specify. Containment mechanisms, such as VMs and containers, provide strong isolation between applications, but do not support these sharing requirements. The incorporation of cloud services within 'Internet of Things' (IoT) architectures [5] is another driver of the requirement for both protection and cross-application data sharing, given these IoT architectures' strong emphasis on (safe) interaction. For example, a

- *Thomas F. J.-M. Pasquier, Jatinder Singh and Jean Bacon are with the Computer Laboratory, University of Cambridge, UK. E-mail: firstname.lastname@cl.am.ac.uk*
- *David Eyers is with the Department of Computer Science, University of Otago, New Zealand. E-mail: dme@cs.otago.ac.nz*

Manuscript received 31 Mar. 2015; revised 25 Aug. 2015; accepted 11 Sept. 2015; current version 6 Oct. 2015.

1. https://www.gov.uk/digital-marketplace

patient being monitored at home may store sensor-gathered medical data in the cloud and share it with selected carers, medical practitioners, and medical research (big-data) repositories, via cloud-hosted and mediated services. Once data has left end-users' homes for cloud services, they need to be assured that it is only accessed as they specify.

Traditional access control tends to be principal/role specific, and apply only within the context of a particular application/service. Controls are applied at policy enforcement points, after which there is no subsequent control over where data flows. Once data has left the direct control of its owner, for example, after being shared with others, it is difficult using traditional access controls to ensure and demonstrate that it is not leaked. If a leak is suspected, it often cannot be established whether this is a breach of confidentiality by a person or due to buggy or misconfigured cloud service software.

Encryption offers protection by restricting access to *intelligible* data, even beyond the boundary of one's technical control. However, encryption hinders flexible, nuanced data sharing, in that key management (distribution, revocation) is difficult. Further, traceability is limited, as being mathematically based there is generally no feedback as to when/where decryption occurs; and a compromised key or broken encryption scheme at any time in the future places data at risk. As such, it is important that data flows are managed and audited, even if data items are encrypted.

Although contracts exist between cloud providers and tenants, and cloud services are increasingly subject to regulation [6], there is at present no way to establish that providers remain in compliance with these agreements and requirements. Also, there are often requirements that data should pass through certain processes, e.g., encryption or anonymisation. There is currently no clear mechanism to express such requirements and demonstrate they have been consistently enforced.

An approach to maintaining the association of data with policy is to use "sticky policies" [7]. Here, owner-specified management constraints are attached to encrypted data. Decryption is only allowed by parties accepting the management constraints and able to enforce them. This forms the basis for establishing contractual relationships between data owners and service providers or other applications. However, this approach requires trust in a (relatively large amount of) software. Further, the enforcement is either at too coarse a granularity or prohibitively expensive. This is further explored in §2.4.

As an alternative, Information Flow Control (IFC) augments traditional access control by offering continuous, system-wide, end-to-end flow control based on properties of the data—for example, "medical data may only be used for research purposes after going through consent checking and anonymisation". IFC allows *security contexts* to be defined system-wide and guarantees non-interference between them. This is achieved by tags applied to entities (e.g., processes, files, database entries), inseparable from the entities they are associated with. Every exchange of data between entities is verified against security-context-domain relationships created by the tags, thus allowing tight control over any subsequent transfers of the data.

In this paper we present CamFlow (**Cam**bridge **Flow** Control Architecture). We outline CamFlow's IFC model and implementation which comprises a new operating system (OS) level implementation of IFC as a Linux Security Module (LSM), with support for application management, together with an IFC-enabled middleware. IFC tags are checked on OS system calls and on message passing by the middleware, to determine whether data flows are permissible. Log records can be made efficiently of all attempted flows, whether permitted or rejected, and this log provides a possible basis for audit, data provenance and compliance checking. By this means it can be checked whether application level policy has been enforced and whether cloud service provision has complied with contractual obligations.

We argue that incorporating IFC into the underlying PaaS-provided OSs, as a small, trusted computing base would greatly enhance the trustworthiness of cloud services, whether public or private, and hence all their hosted services/applications. Our evaluation shows that IFC would incur acceptable overhead and our IFC model is designed to ensure that application developers need not be aware of IFC, although some application providers may wish to take explicit advantage of IFC. We demonstrate the feasibility of our approach via an IFC-enabled framework for web services, see §7.

Contributions: Our main contribution is to demonstrate the feasibility of providing IFC as part of cloud software infrastructure and showing how IFC can be made to work end-to-end, system-wide. In addition to discussing the 'big picture', in this paper we also present a new kernel implementation of IFC and a new audit function. Our approach enables: (1) protection of applications from each other (non-interference); (2) flexible, managed data sharing across isolation boundaries; (3) prevention of data leakage due to bugs/misconfigurations; (4) extension of access control beyond application boundaries; (5) increased transparency, through detailed logs of information flow decisions.

§2 gives background in protection and IFC, then §3 presents the essentials of the CamFlow IFC model, with examples. §4 and §5 describe our new OS-level implementation of IFC as a LSM and its integration via trusted processes with an IFC-enabled middleware, storage services, etc. §6 emphasises that audit in IFC systems produces logs capable of being processed by 'big-data' analytics tools. Audit is central to establishing provenance and for providers to demonstrate compliance with contract and regulation. §7 shows how standard web services are supported transparently by the CamFlow architecture: only a privileged application management framework need be aware of IFC and unprivileged application instances can run unchanged. In all cases,

evaluation is included within the section. §8 summarises, concludes and suggests future work.

2 Background

We first define the scope of current isolation mechanisms, highlighting the need for flexible data sharing *at application-level granularity*, i.e. where applications manage their own security concerns, as well as strong isolation between tenants and/or applications. As an introduction to IFC we outline the evolution of IFC models. Related work on IFC implementation at the OS level and within distributed systems is given with the relevant sections. We end with a brief comparison of IFC with taint tracking (TT) and sticky policies.

2.1 IFC Models

In 1976, Denning [8] proposed a Mandatory Access Control (MAC) model to track and enforce rules on information flow in computer systems. In this model, entities are associated with security classes. The flow of information from an entity a to an entity b is allowed only if the security class of b (denoted $\underline{b}$) is equal to or higher than $\underline{a}$. This allows the *no-read up, no-write down* principle of Bell and LaPadula [9] to be implemented to enforce secrecy. By this means a traditional military classification *public, secret, top secret* can be implemented. A second security class can be associated with each entity to track and enforce integrity (quality of data); *no read down, no write up*, as proposed by Biba [10]. A current example might allow input of information from a government website in the $.gov.uk$ domain but forbid that from "Joe's Blog". Using this model we are able to control and monitor information flow to ensure data secrecy and integrity.

In 1997 Myers [11] introduced a Decentralised IFC model (DIFC) that has inspired most later work. This model was designed to meet the changing needs of systems from global, static, hierarchical security levels to a more flexible system, able to capture the needs of different applications. In this model each entity is associated with two labels: a *secrecy* label and an *integrity* label, to capture respectively the privacy/confidentiality of the data and the reliability of a source of data. Each label comprises a set of tags, each of which represents some security concern. Data is allowed to flow if the security label of the sender is a subset of the label of the receiver, and conversely for integrity.

Implementations of a decentralised model akin to Myers' include a sensitive embedded system for BMW cars [12] and XBook [13] in a social media context. Our own model is described in §3. When implemented from the OS kernel level, applications running under IFC enforcement do not need to be trusted for the data management policy to be properly enforced [14].

2.2 Protection via VMs and Containers

Isolation of tenants in cloud platforms is through hypervisor-supported virtual machines [1], [2] or OS-provided containers [3]. However, flexible sharing mechanisms are also required to manage data exchange between applications contributing to more complex systems, or to achieve end-user goals. For example, government applications might access citizens' records for various purposes; a user's data from different applications might together contribute to evidence related to health or wellbeing.

At present, the sharing of information between applications tends to involve a binary decision (i.e. to share or not), as for example in Google *pods* (containers).[2] Whole resources can be shared, but no control over data usage between applications is provided. Furthermore, there are no means for preventing leakage outside of the mechanisms implemented by the individual applications/services.

Solutions have been proposed to provide intra-application sandboxes (down to individual end-users) [15], but such schemes are difficult to scale, require changes in application logic, and still do not provide control beyond isolation boundaries (i.e. again, loss of control once the data is shared).

IFC has been proposed to guarantee the proper usage of data by social network applications [13]. The aim is to provide purpose-based disclosure via IFC [16] between isolated components, thus guaranteeing that shared data can only be used for a well-defined and agreed-upon purpose.

IFC is by no means proposed as a replacement for access control, VMs or containers, but rather as a complement to those techniques to provide flexible, managed data-sharing. IFC would allow tenants and end-users to maintain control (within an IFC-enforcing world) and define policy applying to their data consistently and beyond isolation and application borders.

2.3 Taint Tracking (TT) Systems

Runtime, dynamic TT is similar to IFC but with less functionality. TT systems use one tag type "taint" instead of secrecy and integrity tags. Tags propagate with data and data flows may be logged. An entity that inputs tagged data acquires the data's tag(s). Data flow constraints are only enforced at specified sink points, for example, when data attempts to leave a mobile phone [17]. Policy is applied at sink points such as preventing private, unanonymised or unencrypted data from flowing, or strictly controlling to where data may flow.

An example of TT used for integrity purposes is to taint data from untrusted sources, e.g., user input from a TCP stream in a web application environment, and enforce that it is sanitised before being processed [18]. This simple mechanism prevents injection attacks that plague badly designed web applications. An example of TT used for confidentiality purposes is to taint sensitive information, e.g., a list of contacts in a mobile phone, and track it through this closed system [17]. Data leaving the

2. https://cloud.google.com/container-engine/docs/pods/

system (i.e. the phone) is analysed to ensure it does not contain sensitive information. Data containing sensitive information should only leave to a number of closely controlled destinations, such as the cloud backup contact list. This approach aids the detection of malicious applications attempting to steal user-sensitive information and send it to third parties. Equally, this type of concern can be captured through the use of IFC policies.

One concern with TT systems is that there is a gap in time between the occurrence of the issue (e.g. a leak, an attack) and when it is detected [19] i.e. problems become evident only when the tainted data reaches a sink (enforcement point). Depending on the degree of isolation between the different parts of the system, and the number of system components involved, this tainted data may have 'contaminated' much of the system. While this can be managed in smaller, closed environments, it is less appropriate for cloud services in general. IFC policies present the clear advantage to *prevent* problems as they occur and to stop their effects propagating to a potentially large part of the system.

Some argue that TT is simpler to use than IFC, and incurs lower overhead, but when the enforcement is systemic and the granularity identical the overheads are similar (compare [17] and the evaluation in §4 and §5). Indeed, the complexity of verifying IFC policy (see §3) is comparable to the cost of propagating taint. For both techniques, most of the overhead comes from the mechanism for intercepting data exchange.

2.4 Sticky Policies

IFC can be seen as a mechanism for enforcing policy; the labels associated with entities represent application policies. IFC is a simple, low-level mechanism. Sticky policy approaches also consider the enforcement of data-bound policy, but at a higher-level.

Casassa-Mont et al. [20] first introduced *sticky policies*, which involves encrypting data along with a list of policies to be enforced on that data. To obtain the decryption key from a Trusted Authority (TA), a party must agree to enforce the policies associated with the data. This agreement may be considered as part of forming a contractual link between the data owner and the service provider. Work has continued in the area [21]–[23]. Sticky policies, typically enforced at the application-level, are generally more complex and heavyweight than the simple secrecy and integrity constraints of IFC. As such, sticky policies tend only to be enforced at particular points, e.g. at administrative boundaries. IFC on the other hand, as we show in §4.1.2, can be enforced continuously at a reasonable cost. In §3, we discuss how complex policies might be built from IFC labels.

Further, the sticky policy approach builds upon the trust established between the data owner, the TAs and services that use the data. A non-compliant service could be black-listed, but only if and when a breach of agreement is detected and the TA updated. Our IFC approach builds only upon the trust between the data owner and the cloud provider. Services and applications running on top of the cloud provider platform need not be trusted. We believe this to be a great improvement to the overall trustworthiness of the system.

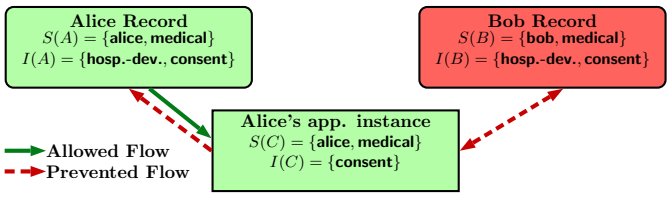

Fig. 1: An allowed safe flow and prevented flows.

3 CamFlow-Model: IFC for the Cloud

IFC operates to ensure that only permitted flows of information can occur, by enforcing data flow policy dynamically, end-to-end, within and across applications/services. *Entities* to which IFC constraints are applied can include a MapReduce worker instance [24], a file, a process, a database entry [25], etc. In CamFlow, IFC is applied continuously, typically on every system call for an IFC-enabled OS, and on communication mechanisms for enforcement across applications/runtime environments. IFC policy should therefore be as simple as possible, to allow verification, human understanding and to minimise runtime overhead. Indeed, there is no need for IFC to encapsulate every possible policy; rather, it augments other control mechanisms, and can help enforce their policies.

3.1 Tags and Labels

We define tags that are tokens, each representing some security concern over secrecy or integrity. The tag bob-private could for example represent Bob's personal data. We associate every entity in the system with two *labels* (sets of tags): an entity A has a secrecy label $S(A)$ and an integrity label $I(A)$. The state of these labels is the *security context* of the entity. The power of IFC is that it guarantees non-interference between security contexts [26], [27].

Example – secrecy: Suppose a patient, Bob is discharged from hospital to be medically monitored at home. The data streams from his sensors are transferred to a cloud service and are to be shared with his medical team at the hospital. The data items from his devices are tagged with medical, bob in their secrecy labels.

Example – integrity: The cloud-based home monitoring support service needs to be assured that the data it receives is from a hospital-issued device. Each sensing device is checked and issued with the tag hospital-device in its integrity label.

Fig. 1 illustrates information flow constraints being applied over both secrecy and integrity dimensions.

3.2 Decentralised Privileges and Security Contexts

In decentralised IFC (DIFC) any active entity can create *new* tags. Tag creation is typically carried out by application managers when setting up application instances.

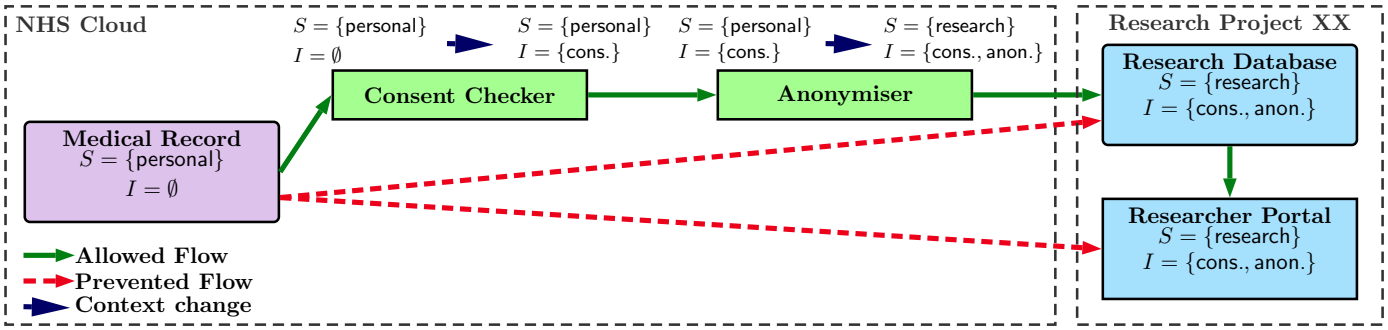

Fig. 2: Medical data declassified and endorsed for research purposes.

When an active entity creates a new tag either for secrecy or integrity, this process is given the corresponding privilege to add and remove the tag to its secrecy or integrity label respectively. If an active entity A has a privilege to add t to its secrecy label, we denote this $t \in P_S^+(A)$, and to remove t from its secrecy label: $t \in P_S^-(A)$ (and similarly $P_I^+(A)$ and $P_I^-(A)$ are the privileges for integrity). An active entity may therefore have four privilege sets in addition to its security context. Application managers will normally set up application instances in security contexts, without the privileges to change them. An example is given in §7.

3.3 Creating a New Entity

We define $A \Rightarrow B$ as the operation of the entity A creating the entity B. An example is creating a process in a Unix-style OS by clone. We have the following rules for creation:

$$\text{if } A \Rightarrow B, \text{ then } \begin{cases} S(B) := S(A) \\ I(B) := I(A) \end{cases}$$

That is, the created entity inherits the security context of its creator. These rules force the creating entity to explicitly change its security context to that required for the entity to be created. We motivate this below in §3.4.2. Note that only labels pass to the created entity; privileges have to be passed explicitly.

3.4 Security

The purpose of IFC models is to regulate flows between entities, and effect label changes and privilege delegation.

Definition 1. *A system is secure in the CamFlow IFC model if and only if all allowed messages are safe (Definition 2), all allowed label changes are safe (Definition 3) and all privilege delegation is safe (Definitions 4 and 5).*

3.4.1 Information Exchange

IFC prevents data leakage by controlling the exchange of information. We follow the classic pattern for IFC-guaranteed secrecy (*no read up, no write down* [9]) and integrity (*no read down, no write up* [10]).

Definition 2. *A flow of information $A \to B$ is safe if and only if:*

$$A \to B, \textit{ iff } \{S(A) \subseteq S(B) \wedge I(B) \subseteq I(A)\}$$

Example – secrecy enforcement: Consider our example of patient monitoring after discharge from hospital, where the patient's devices are tagged with medical, bob in their secrecy labels. In order for the cloud service to be able to receive this data it must also include the tags medical, bob in its secrecy label. Therefore an application instance accessing Bob's medical data must be labelled as such. In §7 we describe how applications can be designed to meet such requirements.

Example – integrity enforcement: The cloud-based home monitoring support service needs to be assured that the data it receives is from a hospital-issued device. To achieve this, the service has an integrity tag hospital-issued in its integrity label and will only accept data from devices with tags hospital-issued.

3.4.2 Label Change

Under the above constraints, information 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. In CamFlow only the process itself is able to change its secrecy and integrity labels, which requires the appropriate privileges and must be explicitly requested.

Definition 3. *A label change noted $A \rightsquigarrow A'$ is safe if and only if for a label X (either S or I) and a tag t:*

$$X(A') := X(A) \cup \{t\} \textit{ if } t \in P_X^+(A)$$
$$\textit{OR}$$
$$X(A') := X(A) \setminus \{t\} \textit{ if } t \in P_X^-(A)$$

Declassifiers and endorsers are the entities with the privileges to perform security context transformations. Declassifiers change the secrecy properties and endorsers change the integrity properties.

Example – declassification: A medical record system is held in a private cloud. Research datasets may be created from these records, but only from records where the patients have given consent. Also, only anonymised data may leave the private protected environment. We assume a health service approved anonymisation procedure. Fig. 2 shows the anonymiser inputting data tagged as personal and declassifying the data by outputting data with secrecy tag research.

Example – endorsement: In the same example, the Research Database is on a public cloud and may only receive research data tagged with consent, anon in its integrity label. In the private cloud we see a process that selects appropriate records for specific research purposes, checks for patient consent and adds the tag consent to the integrity label of its output. The anonymiser process can only input data with this tag; it anonymises the data and outputs data with the tag anon in its integrity label.

Some previous work [14], [28] allows implicit *declassification* and *endorsement*. That is, if an active entity has the privilege to declassify/endorse and the privilege to return to its original state (i.e. for declassification/endorsement over t the entity has privilege t^- and t^+), the declassification/endorsement may occur implicitly without the need for the entity to make the label changes. We believe that this could in practice lead to *unintentional* data disclosure. Suppose an entity has the privilege to declassify top-secret information. The requirement for explicit label change makes it unlikely that the entity will send such data accidentally to an unintended recipient. Our model has stronger constraints that require endorsement and declassification operations to be programmed *explicitly*.

3.4.3 Privilege delegation

An entity is only able to delegate a privilege it owns.

Definition 4. *A privilege delegation is safe if and only if* $t \in P_X^{\pm}(A)$.

3.5 Conflict of Interest

In CamFlow alone among IFC systems, privilege delegation is further restricted by Conflict of Interest (CoI) (or Separation of Duty (SoD)) enforcement. The receiving entity A, must not be put in a situation where it would break a CoI constraint. By this means, an application manager is prevented from creating an application instance with access to conflicting data.

Definition 5. *An entity A does not violate a CoI C if and only if:*

$$\left|\left(S(A) \cup I(A) \cup P_S^+(A) \cup P_I^+(A) \cup P_S^-(A) \cup P_I^-(A)\right) \cap C\right| \leq 1$$

Example – conflict of interest: A CoI might arise when data relating to competing companies is available in a system. In a hospital context, this might involve the results of analysis of the usage and effects of drugs from competing pharmaceutical companies. The companies might agree to this analysis only if their data is guaranteed to be isolated, i.e. not leaked to other companies.

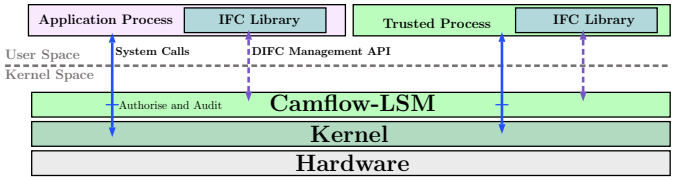

Fig. 3: The interactions of the IFC Security Module (LSM) and a Trusted Process within an OS.

The hospital may be participating in drug trials and want to ensure that information does not leak between trials: suppose a conflict is $C = \{\mathsf{Pfizer}, \mathsf{GSK}, \mathsf{Roche}, ...\}$ and some data (e.g. files) are labelled $\mathsf{PfizerData}[S = \{\mathsf{Pfizer}\}, I = \emptyset]$ and $\mathsf{RocheData}[S = \{\mathsf{Roche}\}, I = \emptyset]$. The CoI described ensures that it is not possible for a single entity (e.g. an application instance) to have access to both RocheData and PfizerData either simultaneously or sequentially, i.e. enforcing that Roche-owned data and Pfizer-owned data are processed in isolation.

The next sections describe the CamFlow platform that enforces the IFC constraints described.

4 OS Enforcement

At the heart of the architecture is a minimal kernel module dedicated solely to OS-level IFC enforcement. The module is trusted to enforce IFC, transparently, across all flows between entities within the OS. User space processes can directly interact with the kernel module, e.g. to delegate privileges (§3.4) through a pseudo-file system, abstracted through a high level API. Higher level considerations and policies can be managed through specifically defined Trusted Processes (see §4.2). The local machine architecture is represented in Fig. 3.

Note that IFC operates alongside and complements other security technologies. It is not a cloud security panacea; challenges regarding covert and side channels, and direct access to hardware by an attacker remain, as they do for systems in general. There are approaches that can help address these security threats, but many are highly disruptive (e.g. synchronisation approaches to reducing timing channels) and are infrequently used. Other threats may be easier to mitigate and solutions may be used when appropriate (e.g. on-disk encryption).

4.1 CamFlow-LSM

Our kernel module, *CamFlow-LSM*, is implemented as a Linux Security Module (LSM) [29]. Although our work is Linux-specific, a similar approach could be used on any system providing LSM-like security hooks. Unlike other DIFC OS implementations [14], [28] our kernel patch is self-contained, strictly limited to the security module, does not modify any existing system calls and follows

LSM implementation best practice. This allows, among other things, LSM stacking [30], [31] and coexistence with other security modules such as e.g. SELinux [32] or AppArmor [33] and complements their MAC enforcement with decentralised information flow policies.

We assume that the rest of the kernel can be trusted and does not interfere with the IFC enforcement mechanism. LSM system hooks have been statically and dynamically verified [34]–[36], 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.

Since applications running on SELinux [32] or AppArmor [33] need not be aware of the MAC policy being enforced, we see no reason to force applications running on an IFC system to be aware of IFC; only those performing declassification or endorsement operations are necessarily aware. This implementation choice is important; cloud providers can incorporate IFC without requiring changes in the software deployed by tenants. Alternatively, applications that wish to manage their own IFC constraints can declare policy through a pseudo-filesystem (as is typical for LSMs) abstracted by a user space library and enforced transparently by the IFC mechanism.

The LSM framework calls security hooks when access to a kernel object is attempted. Security metadata can be associated with kernel objects and is used by the LSM module to make access decisions. Tags and privileges are represented by 64-bit opaque nonces associated with kernel objects such as processes, inodes, files, shared memory objects, messages etc. On interaction between kernel objects, CamFlow-LSM security hooks are called to enforce data-flow policy (§3.4.1) or propagate tags on entity creation (§3.3) as appropriate.

Only active entities (processes) have mutable labels and privileges, all other (passive) entities have immutable labels and no privileges.

Privileges are allocated by the kernel and owned by the creating process (any process can create tags and the associated privileges in a decentralised fashion). Privileges can be passed to other processes, users or groups, CamFlow-LSM verifying that constraints on privilege delegation (§3.4.3) and conflict of interest (§3.5) are not violated. A process can add or remove a tag from its label if it owns the appropriate privilege (following IFC constraints described in §3.4.2), if the current user owns the privilege or if the current group owns the privilege. How tags are shared and managed must be considered with care when designing an application and the system must be administered accordingly.

4.1.1 Checkpointing and Restoration

Checkpointing a process involves halting its execution, allowing it to be restarted at a later stage, and enabling migration, e.g. [37]. LSM state is normally saved and restored by the checkpointing system, e.g. [38], and our module further exports an API to more efficiently serialise and restore security context.

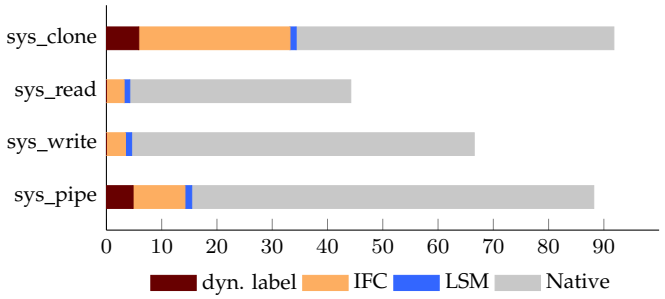

Fig. 4: Overhead introduced into the OS by CamFlow LSM (x-axis time in μs).

Furthermore, self-checkpointing and restoring the previous state of a process, has been demonstrated [39] to be a beneficial feature for IFC systems. This is particularly useful for processes serving requests. In such a scenario the state of the process is saved after initialisation. When a request is received, the serving process sets itself up in the security context appropriate to serve the request. After the request is served (or a series of requests if the system is session-based as described in §7), the process restores its memory state and security context to what they were immediately after initialisation. This improves performance and prevents data leaks between security contexts.

4.1.2 OS Evaluation

We tested the CamFlow-LSM module on Linux Kernel version 3.17.8 (01/2015) from the Fedora distribution.[3] The tests are run on an Intel 2.2Ghz i7 CPU and 6GiB RAM machine.

Measurements are done using the Linux tool ftrace [40] 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. 4.

We can distinguish two types of induced overhead: 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 entity's labels and privileges. Recall that passive entities have no privileges. Overhead measurements for other system calls/data structures are essentially identical as they rely on the same underlying enforcement mechanism, and are not included.

3. It is not feasible to provide a comparison with the Laminar implementation [28], that is closest in technical terms to our work, as the implementation available https://github.com/ut-osa/laminar is for an obsolete kernel version 2.6.22 (07/2007).

The CamFlow-LSM overhead is a few percent, see Fig. 4. We provide a build option that further improves performance by declaring labels and privileges with a fixed maximum 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 dynamic label component in Fig. 4). This is an accessible trade-off as in practical scenarios, labels rarely exceed more than five tags. However, for most applications, the overhead is imperceptible and lost in system noise; it is hard to measure without using kernel tools, as the variation between two executions may be greater than the overhead.

4.2 Trusted Processes

The CamFlow-LSM is trusted to enforce IFC at the kernel level. Its functionality is minimal; strictly confined to the enforcement of IFC policies as described in §3. This guarantees easier maintainability and a system that is agnostic to higher level application requirements, thus minimising the constraints imposed on user-space application design.

We introduce the concept of a *trusted process*, that allows application/platform-specific concerns to be managed in user space by bypassing some LSM-enforced IFC constraints. For example, a trusted process might serve as a proxy for external connections, as in the Trusted IFC Gateway in the example in §7, setting up and managing application components' labels. Trusted processes are used to interact with persistent storage (see §5.3), for checkpointing and restoring processes (see §4.1.1) and for managing inter-process and external communication (see §5).

Fig. 5 shows OS instances running the CamFlow-LSM hosting a number of application processes, that may be grouped in containers. Each OS instance has a single trusted process (Security Context Manager) to manage its hosted processes' IFC labels and privileges. In addition, each process has an associated trusted middleware process to handle inter-process and inter-machine communication. Such communication may be within or between containers, OSs or clouds.

In this example, S represents a particular set of secrecy tags, and I a particular set of integrity tags, both of which remain the same throughout. The application processes and other OS objects, such as pipes and files, are labelled $[S, I]$. The process labelled $[\emptyset, I]$ writes 'public' data to a pipe, which is read by a process labelled $[S, I]$, assuming all the I tags match correctly. Similarly, two processes are shown writing to and reading from a file.

The Security Context Manager maps between the kernel-level representation of tags (as 64-bit integers) and the representation of tags in user space. Within a cloud or other trusted environment, tags may be simple strings. When tags need to cross domain boundaries, e.g., when cloud services form part of a wider architecture, as in IoT, tags may need to be protected by cryptographic means (see §5.1).

Trusted processes are either set up through static configuration, read at boot time by the CamFlow-LSM module, or created at runtime by another trusted process. Trusted processes must either be managed by a trusted party (in our current approach the underlying infrastructure provider) and/or the code must be auditable and a means to verify the current version running on the platform must be provided (see §4.3).

4.3 Leveraging Hardware Roots of Trust

Incorporating IFC into cloud-provider OSs would enhance the trustworthiness of the platform. However, IFC only guarantees protection above the technical layer in which it is enforced. Recent hardware and software developments make it possible to attest that the software layers on which our platform runs have been audited.

The Trusted Platform Module (TPM) [41], as used for remote attestation [42], is one such hardware mechanism. TPM is used to generate a nearly unforgeable hash representing the state of the hardware and software of a given platform, that can be remotely verified. Therefore, a company could audit the implementation of our IFC enforcement mechanism and ensure that our kernel security module, messaging middleware and the configuration they provide are indeed running on the platform. Any difference between the expected state of the software stack and the platform could be considered a breach of trust; such considerations can easily be embedded in the contractual obligations of the cloud provider.

TPM and remote attestation for cloud computing [43] are reaching maturity, with IBM rolling out an open source, scalable trusted platform based on virtual TPMs [44]. Indeed, Berger et al. [44] describe a mechanism allowing the TPM and remote attestation to be provided for virtual machine offerings and container-based solutions, covering the whole range of contemporary cloud offerings. Furthermore, the approach not only allows the state of the software stack to be verified at boot time, but also during execution, and can thus prevent run-time modification of the system configuration.

5 Cross-Machine Enforcement

CamFlow-LSM operates to protect flows within the OS. However, it is also important that flows are protected across OS instances.

Generally, in order to guarantee flow constraints, only processes P such that $S(P) = \emptyset$ and $I(P) = \emptyset$, i.e. not subject to IFC constraints, are allowed to directly connect to or receive messages from connections on remote OS (e.g. through a socket). In order to connect to another machine, a process must either: 1) be able to declassify to change its security context to $S(P) = \emptyset$ and $I(P) = \emptyset$; 2) communicate through an intermediate trusted process.

As such, CamFlow contains an IFC-enabled, fully-featured messaging middleware (**CamFlow-MW**) to both facilitate communication and guarantee enforcement across machines. For want of space, we only consider the middleware concepts relevant to IFC; details on the

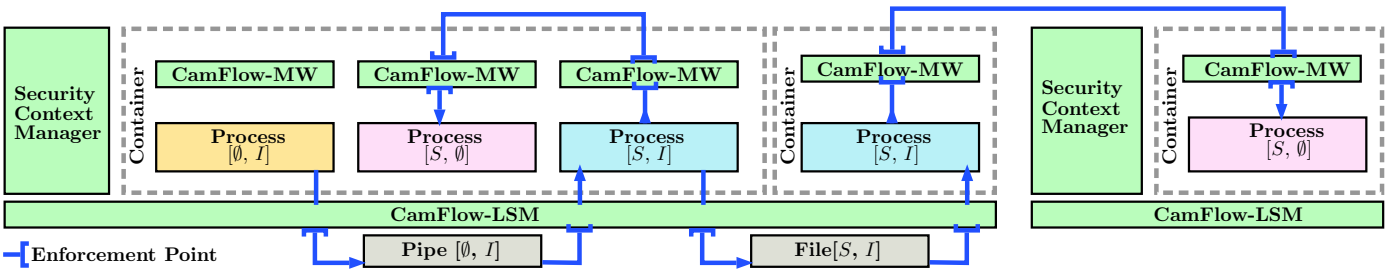

Fig. 5: CamFlow Architecture: Labelled OS objects, trusted processes and communication middleware.

general middleware (as it was prior to IFC/CamFlow integration) can be found in [45]. In short, the middleware supports strongly-typed messages; a range of interaction paradigms, including request-reply, broadcast, and streams; flexible resource discovery; and security mechanisms including access controls and encrypted communication. A particular feature is its support for dynamic reconfiguration based on event-driven policy. This simplifies both application development and deployment, as concerns can be abstracted and tailored to the particular environment, rather than embedded within application code.

The role of the middleware is to move towards continuous, end-to-end data flow management, such that IFC can be enforced *across* applications/machines (kernels). There is work on IFC enforcement across machines; however, these impose specific requirements, such as design-time considerations [46], a particular language/runtime [47], or constraints on system architecture/implementation [48]. In contrast, we integrate IFC functionality into the general, fully featured distributed systems middleware mentioned above (see [49]), to provide flexibility and be more generally applicable. We deliberately avoid imposing a structure on system design, instead integrating IFC functionality into the sort of communications infrastructure common to current enterprise and cloud systems.

5.1 Remote Interactions

CamFlow-MW operates by associating a trusted process (see §4.2) with an entity that seeks to communicate via the messaging system. Its fit within the broader architecture is depicted in Fig. 5. The process is responsible for handling the communication of messages, and enforces IFC based on the current runtime labels of the entity on whose behalf it operates.

It follows that for IFC to be enforced across machines, tags require system-wide management, i.e. throughout the cloud service. In [50] we proposed that the widely used and available X.509 certificates could be used. The approach relies on public key certificates and attribute certificates [51], to respectively identify the application associated with the CamFlow-MW instance and the tags associated with this application.

As part of establishing a connection, CamFlow-MW ensures that each entity authorises communication with the other, according to a local access control policy. Decisions are based on component metadata, the relevant authentication aspects secured through PKI (certificates). Similarly, IFC policy must also be verified to ensure data flows are authorised according to IFC policy. Attribute certificates provide cryptographic means to determine and verify the tags associated with the remote entity (see [50]), on which policy can be enforced. If tags do not accord, the connection will not be established.

We also see potential for remote attestation, based on hardware integrity measures (see §4.3), to be integrated into this authorisation phase, to ensure the remote machine operates a reliable IFC enforcement regime.

5.2 Message-Level Enforcement

CamFlow messages are strongly typed, where a *message type* is defined by a schema describing its set of attributes. For an instance of a message, an *attribute* consists of a name, type and value. The support for IFC within messages is fine-grained, in that individual attributes within messages can also be labelled. These attribute labels introduce additional IFC constraints over and above those already applying to the entity, i.e. as recognised by the kernel-LSM, and validated on connection establishment.

Labels can be defined within message type schema, which sets the attributes' IFC labels for all message instances of the type. These labels cannot be changed by entities dealing in such messages, and the entities must hold the requisite labels to interact with the attributes. Otherwise, the entity producing/publishing a message can set the security labels for the attributes (for those not predefined), if the entity holds the associated privileges. Enforcement occurs as follows:

Receiving: If the receiving entity's labels do not agree with those of an attribute value, the attribute value (and any sub-attributes) are removed from (made null in) the message. This is enforced on message receipt, before it is delivered to the entity.

Sending: An entity cannot send values for attributes where its labels do not agree with those of the attribute. This is enforced when an entity attempts to send a message, ensuring values for any attributes violating this

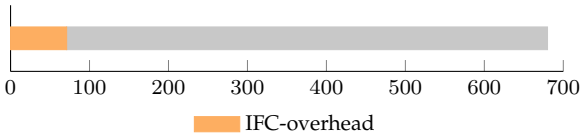

Fig. 6: IFC overhead of CamFlow-MW for a workload transmission of 5000 messages (x-axis in ms)

policy are removed, before message propagation.

Enforcement is automatic, meaning that applications using the messaging system can be subject to IFC enforcement completely transparently (i.e. without their direct involvement); though again, there is the interface for the application to actively manage IFC where required. In addition, the general reconfiguration capabilities of the middleware enable connections between components to be defined and managed at runtime, providing another mechanism for controlling communication [45].

5.3 Integrating With Persistent Storage

One technique to provide IFC with persistent data stores, is to store the tags alongside the data, and a trusted software component ensures that when information is read from the store, the corresponding labels are applied. In Flume [14], a trusted process provides the interface between untrusted applications and persistent storage. More recent work has seen the emergence of databases that natively understand IFC concepts and can enforce IFC policies [25].

We see much promise in having the middleware mediate between persistence systems and the kernel, to ensure consistent IFC application.

5.4 Evaluation

As shown in Fig. 6, the results indicate that IFC enforcement introduces an overhead of ∼13% in performance time compared to the standard, non IFC-enabled middleware (see [49] for details). Note that these results were measured in the context of a particular workload, deliberately designed to highlight the impact of IFC enforcement. It follows that the overheads associated with real-world usage are most likely less onerous.

6 Audit: Data-Centric Logs

IFC, in addition to providing strong assurances that policy is being enforced, can also provide a data-centric log [52] detailing the information flows within and between system components. In addition to enforcing IFC via our LSM module we log the data flows of labelled processes, policy decisions, privileges and IFC security context manipulations. Equivalent inter-machine operations via the middleware are also recorded.

Cloud logging systems are generally based on legacy logging systems (OS, web-server, database etc.) that either fail to capture the needed information, or are extremely complicated to interpret in a useful manner [53]. More importantly, such logs tend to be relevant only to the particular service or component, which makes

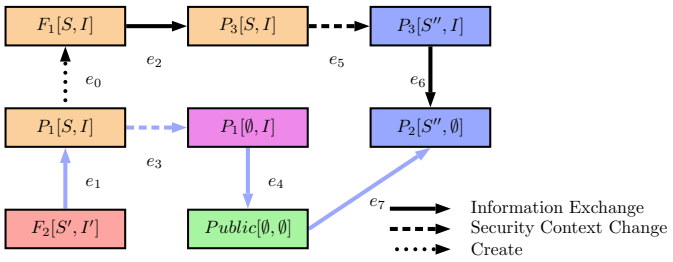

Fig. 7: Simplified audit graph from IFC OS execution (we omit metadata for readability). The path to disclosure is shown in blue/pale.

it difficult, if not impossible, to audit across a range of applications, clouds, etc.

IFC logs, as provided by our platform, allow us to capture information on application-level data flows, both attempted and permitted, allowing the correct expression and implementation of data flow policy to be checked. This provides transparency allows for meaningful audit, in terms of investigating the circumstances in which data leakage occurs, and provides evidence of compliance, e.g. with legal obligations [54].

6.1 Analysing Paths to Disclosure

To assist in interpreting log information, we build a directed graph corresponding to the allowed flows during the execution of our system, as shown in Fig. 7. The flows defined in our IFC model (see §3), namely **data flow**, **creation flow**, **security context** change and **privilege delegation**, correspond to the edges of the directed graph. Entities (such as processes, files, messages etc.) are represented as nodes in the graph.

In addition to information necessary to build the graph (as shown in the figure), additional metadata is collected for forensic purposes, which is context/entity/event-dependent. These audit entries, provided by the LSM and middleware, can be exploited by a dedicated service implemented in user space, connected to the kernel collection mechanism via relayfs [55]. This service feeds, for example, a graph visualisation tool such as Cytoscape [56] or a graph database such as Neo4J.[4]

Such a directed graph helps one identify data leaks. For example, a tenant might discover that some sensitive medical data leaked into a data store where only anonymised research data were supposed to be stored. IFC is enforced in line with the policy encapsulated in labels; thus data may leak if such policy is improperly expressed and/or declassification/endorsement processes are not correctly implemented (e.g. if the anonymisation process in Fig. 2 allows re-identification).

Suppose that an information leak is suspected between different security contexts $L_1[S,I]$ and $L_2[S',I']$. 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,

4. http://neo4j.com/

there must be a path between some entity E_i such that $S(E_i) = S \wedge I(E_i) = I$ and another entity F_i such that $S(F_i) = S' \wedge I(F_i) = I'$.

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 ID associated with the edges comprising the path. We denote by e_i, the last incoming edge to the entity under investigation with labels $[S', I']$; only edges such that $e < e_i$ should be considered. When applied to all nodes along a path, this rule ensures strictly monotonically increasing timestamps from the first node to the last. Fig. 7 shows in blue/pale a possible data disclosure path, from file F_2, from a very simple audit graph. We know from the event IDs e_0 and e_1 that the data disclosure did not occur through file F_1 and process P_3, but through P_1's declassification.

6.2 Demonstrating Compliance

Compliance with certain requirements can be demonstrated through queries over the graph. We assume the audit data is stored in a graph database that we can query. For example, the following plain English policy: *"European personal data sent to the US must be anonymised"* [57], is equivalent to writing a query that verifies that there is no path between EU- and US-labelled data without an anonymisation process.

The policy *"Medical data stored in database X must have received proper consent and be anonymised"* [54] can be expressed as a query verifying that there is no path between data labelled as medical and the database, without consent-checking and anonymiser processes. In addition, an investigator may want to know which anonymisation algorithm has been run, which data has been used to generate the anonymised records etc. Our audit graph assists in answering such questions.

Note that IFC only applies guarantees with respect to flows. Demonstrating the overall effectiveness of the management regime, e.g. the quality and suitability of the anonymisation algorithm, is out-of-scope for a flow-based enforcement mechanism.

6.3 Audit as 'Big Data'

We are potentially generating a vast amount of data in our IFC logs. However, unlike standard system logs that are complex to analyse, our logs generate graphs that are ideal for analysis by "big data" tools that have been developed for this purpose [58].

Since the amount of data is potentially huge, the amount of data being logged can be fine-tuned to meet the requirements of the platform/tenant; e.g. by reducing the amount of metadata being stored, by logging only security context changing operations, by logging only information corresponding to some target security context, keeping operations on unlabelled entities outside of the log etc. The decision on what needs to be logged then becomes a tradeoff between utility and the volume (cost) of log generated, which can be decided in order to correspond to legal or contractual requirements (for example, a regulated sector may need to have a fine-grained log to satisfy data forensic requirements). Indeed, as such an approach is new to the cloud, such considerations will be refined by experience, with best practices developing over time.

6.4 Audit Access

Logs can contain sensitive information and access to them should be controlled. This represents an area of our ongoing work. Traditional access controls clearly play a role; however, secrecy tags could also be leveraged. For example, an auditor, before being granted access to audit logs, could be forced to demonstrate ownership of the corresponding secrecy IFC tags (for example through cryptographic means as in §5.1). The auditor may be granted access to a log entry only if $S(origin) \cup S(destination) \subseteq S(auditor)$.

7 Example: Support for Web Services

One of the most common uses of PaaS is to host web applications. In this section we present the implementation of such a solution built on the infrastructure described in §4, in order to evaluate and demonstrate the feasibility of our proposed approach. This is illustrated in Fig. 8. We run standard and unmodified Ruby web applications.

Interaction with end-users is achieved through a "gateway" between the IFC and non-IFC worlds. Similarly, interaction with cloud services (such as data stores) is also achieved through our messaging middleware as discussed in §5. The requirement for this gateway can be removed if a trustworthy IFC implementation can be provided at the client side, consistent with the cloud implementation with respect to tag naming, enforcement, etc. Tag naming in general, system-wide, is an issue beyond the scope of this paper, see further §8. In our proof of concept implementation the gateway is a simple Apache server running a custom-built module.

The role of the gateway is to authenticate the end-user when a session is created, and to associate this session with an application instance running within the security context corresponding to the user. Recall that a security context comprises the S and I labels. Any further requests to the gateway in that session are routed to the corresponding application instance. Once an instance no longer has an associated session it can be recycled using self-checkpointing, as described in §4.1.1.

Several application types are running over our cloud-based, web services platform. For example, in a medical context these might be medical record editing, pharmacy ordering, social services etc. A single, shared, identity service for the end-user is part of the cloud provider offering (in our proof of concept implementation we used OAuth [59]).

The GP authenticates, is authorised as treating doctor for Alice and selects the 'identity' that corresponds to Alice. A new session is created server-side by the gateway, with the requested application instance running in the corresponding security context, with $S =$

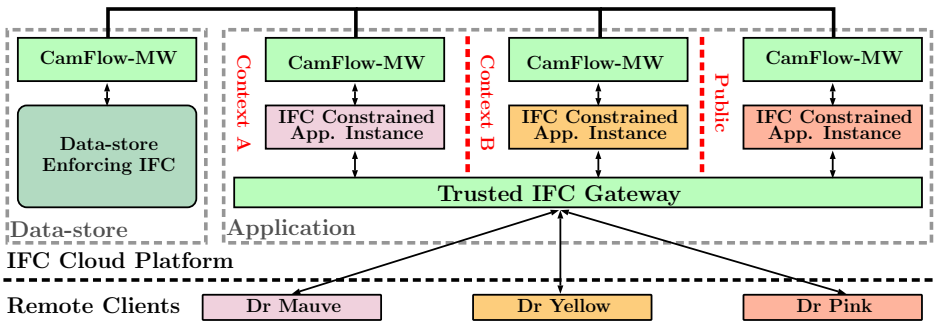

Fig. 8: PaaS Architecture on top of IFC-OS

$[\mathsf{medical}, \mathsf{alice}], I = [\emptyset]$. When the GP wants to access applications on behalf of a new patient, he needs to close Alice's session, authorise as treating doctor for Bob and open a new session for Bob.

The control described above is not achieved by the application, but by the platform itself and can be controlled by the end-user, subject to access control. That is, a medical application used on Alice's behalf runs in a security context in which data cannot flow to that of another patient. Furthermore, applications running on behalf of a given user can share the data of that user without the risk of seeing a buggy application leaking data between end-users; the flow of data is not controlled by the application, but by the platform.

As described in §4, we assume the middleware and the OS enforcement are provided as a service by the underlying platform. A tenant wanting to use the third-party, web-service offering, once his trust in the underlying platform is established, needs only to audit the gateway; again, the underlying infrastructure provider could either provide such a gateway or audit it. The rest of the software stack of the third-party, web-service provider is bound by the IFC enforcement mechanism and therefore need not be trusted.

8 Conclusion & Future Work

IFC allows data flows to be controlled continuously throughout a system, by providing an information-centric MAC scheme that continuously ensures non-interference between security contexts. This paper presented the CamFlow platform, that demonstrates the potential of cloud-deployed IFC as supporting: (1) protection of applications from each other; (2) flexible data sharing across isolation boundaries; (3) prevention of data leakage due to bugs/misconfigurations; (4) extension of access control beyond application boundaries; (5) data flow transparency.

Specifically, we detailed a new kernel implementation of IFC as an LSM, demonstrating low overhead even for worst-case scenarios, where processes continuously make read/write system calls. We also described the integration of a messaging middleware to enforcing IFC across machines. This combination makes it possible to provide whole-system IFC to PaaS cloud services, and therefore also SaaS. Our approach and implementation were designed so that applications can run unchanged over IFC, thus making cloud adoption feasible.

We also indicated how the data-centric logs based on IFC enforcement could provide the means to audit an IFC-enabled system, whereby a log can be processed as a directed graph to investigate leaks and attacks and show compliance with data management requirements. Though this represents our initial work in the area, there appears much promise.

In light of the above, we believe that IFC has great potential as a security mechanism for the cloud whereby trust in a few major cloud providers, deploying IFC, can be built on to provide a demonstrably trustworthy computing environment.

CamFlow was developed with cloud deployment in mind. Our future work will investigate the challenges of a broader distributed context.

It is already feasible to extend CamFlow to support mobile environments. Android supports the full SELinux enforcement,[5] and an Android-LSM integration has been demonstrated [60]. But when dealing with multiple cloud services, particularly as they become part of a wider distributed architecture such as in the IoT, a trustworthy, system-wide deployment of IFC can no longer be assumed. Much work remains on establishing trust in (and the trustworthiness of) the IFC enforcement mechanism within end-users' devices. Outside a cloud context, all parties' trust in a common third party's enforcement of IFC constraints, cannot be assumed (unlike the cloud provider for cloud services). We intend to explore leveraging hardware roots of trust and remote attestation to ensure the integrity and trustworthiness of IFC enforcement mechanisms.

Another area of investigation concerns the representation of tags across administrative domains. In a cloud context, a federated approach can be envisaged where a common understanding of tags could be negotiated across multiple domains. For example, in [54] we dis-

5. https://source.android.com/devices/tech/security/selinux

cussed initial thoughts on managing data according to specific obligation regimes. However, as the number of administrative domains increases, both a global tag naming scheme and mechanisms for ad-hoc negotiation become necessary.

Related is the sensitivity of the tags themselves. Knowledge of the meaning of a tag can indicate that the associated entity contains or deals with certain information. If a relationship can be established between an entity and the information owner, this may disclose private information about the information owner. This may lead to work on a need-to-know negotiation mechanism to establish a secure channel between hosts, especially in a wide-scale distributed system. A promising mechanism is *private set intersection* to determine tags' subset relationship (§3.4.1).

Another challenge concerns extending audit to distributed architectures, both in terms of resource management and regulating access to log data.

ACKNOWLEDGMENTS

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.

REFERENCES

[1] 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.

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

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

[4] P. Desnoyers, O. Krieger, B. Holden, and J. Hennessey, "Using OpenStack for an Open Cloud eXchange (OCX)," in *International Conference on Cloud Engineering (IC2E)*. IEEE, 2015.

[5] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A Gap Analysis of Internet-of-Things Platforms," 2015, Arxiv, arXiv:1502.01181. [Online]. Available: http://arxiv.org/abs/1502.01181

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

[7] S. Pearson and M. Casassa-Mont, "Sticky Policies: An Approach for Managing Privacy across Multiple Parties," *Computer*, vol. 44, July 2011.

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

[9] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model," The MITRE Corp., Bedford MA, Tech. Rep. M74-244, 1973.

[10] K. J. Biba, "Integrity Considerations for Secure Computer Systems," MITRE Corp., Tech. Rep. ESD-TR 76-372, 1977.

[11] A. C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control," in *17th Symposium on Operating Systems Principles (SOSP)*. ACM, 1997, pp. 129–142.

[12] A. Bouard, B. Weyl, and C. Eckert, "Practical information-flow aware middleware for in-car communication," in *Workshop on Security, privacy & dependability for cyber vehicles*. ACM, 2013, pp. 3–8.

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

[14] 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.

[15] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov, "πBox: A Platform for Privacy-Preserving Apps." in *Symposium on Networked System Design and Implementation*. USENIX, 2013, pp. 501–514.

[16] 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.

[17] 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.

[18] I. Papagiannis, M. Migliavacca, and P. Pietzuch, "PHP Aspis: Using partial taint tracking to protect against injection attacks," in *2nd USENIX Conference on Web Application Development*, 2011, p. 13.

[19] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Symposium on Security and Privacy*. IEEE, 2010.

[20] M. Casassa-Mont, S. Pearson, and P. Bramhall, "Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services," in *Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on*. IEEE, 2003, pp. 377–382.

[21] S. Bandhakavi, C. C. Zhang, and M. Winslett, "Super-sticky and declassifiable release policies for flexible information dissemination control," in *Proceedings of the 5th ACM workshop on Privacy in electronic society*. ACM, 2006, pp. 51–58.

[22] D. W. Chadwick and S. F. Lievens, "Enforcing sticky security policies throughout a distributed application," in *Proceedings of the 2008 workshop on Middleware security*. ACM, 2008, pp. 1–6.

[23] M. Casassa-Mont, I. Matteucci, M. Petrocchi, and M. L. Sbodio, "Towards Safer Information Sharing in the Cloud," *International Journal of Information Security*, pp. 1–16, 2014.

[24] 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.

[25] 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.

[26] D. Von Oheimb, "Information Flow Control Revisited: Noninfluence = Noninterference + Nonleakage," in *Computer Security–ESORICS 2004*. Springer, 2004, pp. 225–243.

[27] D. Hedin and A. Sabelfeld, "A perspective on Information-Flow Control." NATO, 2012.

[28] 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.

[29] 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.

[30] M. Quaritsch and T. Winkler, "Linux Security Modules Enhancements: Module Stacking Framework and TCP State Transition Hooks for State-Driven NIDS," *Secure Information and Communication*, vol. 7, pp. 7–13, 2004.

[31] C. Schaufler, "LSM: Generalize existing module stacking," *Linux Weekly News*, 2014.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems." Springer, 2013, vol. 65, no. 3, pp. 1302–1326.

[38] O. Laadan and S. E. Hallyn, "Linux-CR: Transparent application checkpoint-restart in Linux," in *Linux Symposium*, 2010, p. 159.

[39] B. Niu and G. Tan, "Efficient User-space Information Flow Control," in *Symposium on Information, Computer and Communications Security (SIGSAC'13)*. ACM, 2013, pp. 131–142.

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

[41] B. Parno, "Bootstrapping Trust in a" Trusted" Platform," in *Conference on Hot Topics in Security (HotSec'08)*. USENIX, 2008.

[42] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards Trusted Cloud Computing," in *Conference on Hot Topics in Cloud Computing*. USENIX, 2009, pp. 3–3.

[43] 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.

[44] 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.

[45] J. Singh, D. Eyers, and J. Bacon, "Policy Enforcement within Emerging Distributed, Event-Based Systems," in *ACM Distributed Event-Based Systems (DEBS'14)*, 2014, pp. 246–255.

[46] L. Sfaxi, T. Abdellatif, R. Robbana, and Y. Lakhnech, "Information Flow Control of Component-based Distributed Systems," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 2, pp. 161–179, 2013.

[47] S. Yoshihama, T. Yoshizawa, Y. Watanabe, M. Kudoh, and K. Oyanagi, "Dynamic Information Flow Control Architecture for Web Applications," in *ESORICS 2007*, J. Biskup and J. Lopez, Eds. Springer, 2007, vol. LNCS 4734, pp. 267–282.

[48] W. Cheng, D. R. K. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov, "Abstractions for Usable Information Flow Control in Aeolus," in *USENIX Annual Technical Conference*, Boston, 2012.

[49] 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.

[50] J. Singh, T. F. J.-M. Pasquier, and J. Bacon, "Securing Tags to Control Information Flows within the Internet of Things," in *International Conference on Recent Advances in Internet of Things (RIoT'15)*. IEEE, 2015.

[51] J. S. Park and R. Sandhu, "Binding Identities and Attributes Using Digitally Signed Certificates," in *Annual Conference on Computer Security Applications*. IEEE, 2000, pp. 120–127.

[52] A. Ganjali and D. Lie, "Auditing cloud management using information flow tracking," in *Workshop on Scalable Trusted Computing*. ACM, 2012, pp. 79–84.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] R. Angles and C. Gutierrez, "Survey of Graph Database Models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.

[59] D. Hardt, "The OAuth 2.0 Authorization Framework," IETF, Tech. Rep. RFC 6749, 2012.

[60] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *Network and Distributed System Security Symposium*. Internet Society, 2013.

Thomas Pasquier is a PhD student and research assistant in the Computer Laboratory at the University of Cambridge. His research interests include identity and data management in distributed systems, particularly the cloud. His MPhil from Cambridge included a project on "Prevention of identity inference in de-identified medical records".

Jatinder Singh is a Senior Research Associate at the Computer Laboratory, University of Cambridge. His research interests concern management control in distributed systems, particularly regarding cloud and the Internet of Things.

David Eyers is a Senior Lecturer at the University of Otago, New Zealand and a Visiting Research Fellow at the Cambridge Computer Laboratory.

Jean Bacon is a Professor of Distributed Systems at the University of Cambridge, and leads the Opera research group, focussing on open, large-scale, secure, widely-distributed systems.