# 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 provides many related services for its citizens on a common platform. Similar considerations apply to the end-users of applications. But in particular, the incorporation of cloud services within 'Internet of Things' architectures is driving the requirements for both protection and cross-application data sharing.
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' dataflow 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, giving configurable 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. Our contribution is to demonstrate the feasibility of incorporating IFC into cloud services: we show how the incorporation of IFC into underlying IaaS or PaaS provided OSs would address application sharing and protection requirements, and more generally, greatly enhance the trustworthiness of cloud services at all levels, at little overhead, and transparently to tenants.

**Keywords**—Security, Audit, Cloud, Information Flow Control, Middleware, Provenance, Linux Security Module, PaaS, Data Management, Compliance

## 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 AWS, 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 businesses. 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.

Users of cloud services still need to be assured that their data is protected from other cloud users and from leakage 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

- *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 March 31st 2015.*

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

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 and 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 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 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 generally 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 continuously that providers are 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.

Information Flow Control (IFC) augments traditional access control by offering 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 such data to be tagged after these processes have been carried out as, e.g., *consenting, anonymised* (Fig. 2). The tags are metadata, inseparable from the data for its lifetime, system-wide. Another example is that Bob's sensor-gathered medical data may be tagged *medical, bob-private*, and these tags stick to the data after sharing, thus allowing tight control over any subsequent transfers of the data. We have experience of collaborating in the healthcare domain [7] and use this for our examples.

In this paper we present CamFlow (Cambridge 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 through the middleware, to determine whether data flows are permissible. Log records can be made of all flows efficiently, whether permitted or rejected, and this log is a 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 IaaS or 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 data flow transparency, aiding data management, the identification of policy errors/ misconfigurations, and providing evidence for compliance with contractual/regulatory requirements.

§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).

### 2.1 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 *pod* (containers).[2] Whole resources can be shared, but no control over data usage between application 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) [8], 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 [9], by running them in VMs constrained by IFC, at the granularity of whole VMs. The aim is to provide purpose-based disclosure via IFC [10] between VMs, thus guaranteeing that shared data can only be used for a well-defined and agreed-upon purpose.

We propose a solution that offers flexible, scalable isolation; from that of individual parametrisable roles within a system composed of several applications (see §7) to system-wide policy (e.g. legal location requirements [11]). Note that 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. https://cloud.google.com/container-engine/docs/pods/

### 2.2 IFC Models

In 1976, Denning [12] 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 [13] 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 [14]. 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 [15] 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.

### 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 [16]. 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 [17]. 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 [16]. Data leaving the system (i.e. the phone) is analysed to ensure it does not

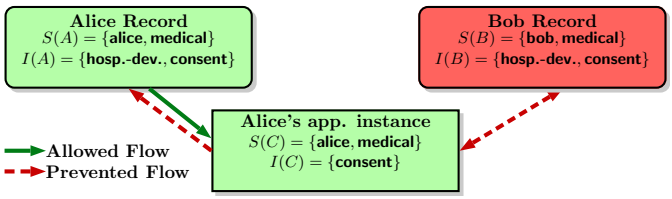


Fig. 1: Allowed safe flow example.

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 [18] 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 [16] 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.

## 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 cloud web applications [19], a MapReduce worker instance [20], a file, a database entry [21], etc. 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.

**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 from his devices is tagged with medical, bob-private 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.

### 3.2 Decentralised Privileges and Security Contexts

In decentralised IFC (DIFC) any active entity can create *new* tags. When an active entity creates a new tag either for secrecy or integrity, this process is given the corresponding privilege to add and remove this 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).

In general, application instances will be set up 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:

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

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 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 Safe Messages

IFC prevents data leakage by controlling the exchange of information. We follow the classic pattern for IFC-

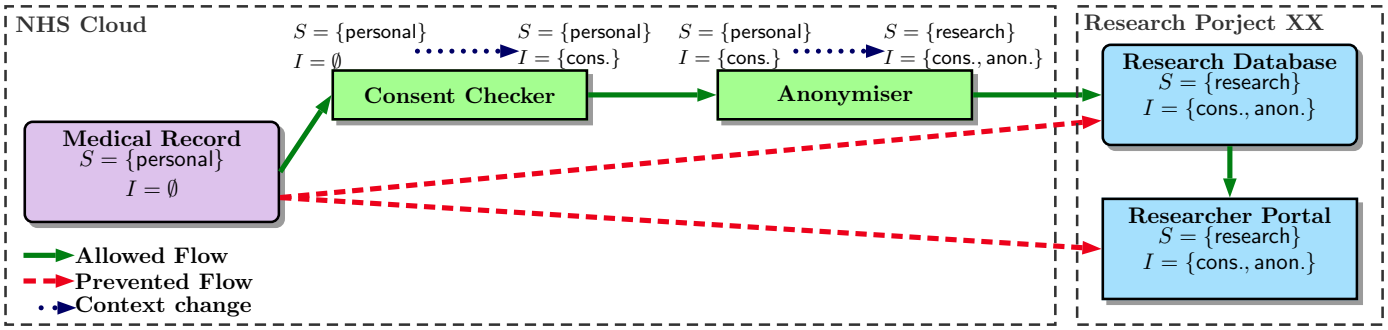


Fig. 2: Medical data declassified and endorsed for research purposes, as in [22], [23]

guaranteed secrecy (*no read up, no write down* [13]) and integrity (*no read down, no write up* [14]).

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

$$A \to B, \textit{ iff } \begin{cases} S(A) \subseteq S(B) \\ I(B) \subseteq I(A) \end{cases}$$

**Example – secrecy enforcement:** Consider our example of patient monitoring after discharge from hospital, where the patient's devices are tagged with medical, bob-private 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-private 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 Safe Label Changes

In CamFlow as in Flume [24] or HiStar [25], only the process itself is able to change its secrecy and integrity labels and must request this explicitly. It has been shown that implicit label changes can lead to covert channels [12], [25].

**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 $t$:*

$$X(A) := X(A) \cup \{t\} \textit{ if } t \in P_X^+(A)$$
$$OR$$
$$X(A) := X(A) \setminus \{t\} \textit{ if } t \in P_X^-(A)$$

**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 their integrity labels. 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.

The portal for medical researchers is authorised to read data from the research database because it has the secrecy tag research. It can only input data that has integrity tags consent, anon.

In IFC systems, label changes are explicit actions. However, previous work [24], [26] allows implicit *declassification* and *endorsement*. That is, if an active entity has the privileges 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 explicitly. We believe that this could in practice lead to unintentional data disclosure between security contexts. Therefore, our model has stronger constraints that require endorsement and declassification operations to be *explicit*.

#### 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 $t \in P_X^{\pm}(A)$.*

This rule 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. This is an additional constraint not present in other IFC systems.

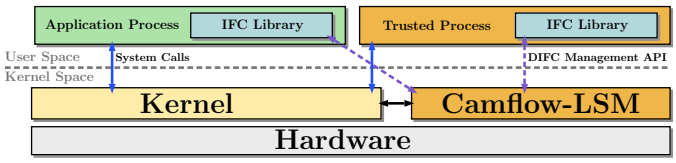


Fig. 3: The CamFlow platform, depicting the interactions of the IFC Security Module (LSM) and a Trusted Process.

**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.

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.

## 4 The CamFlow Platform

We now introduce the CamFlow platform that enforces the IFC constraints as described in §3. Core to the platform is a minimal kernel module dedicated solely to OS-level IFC enforcement. The module is trusted to enforce, transparently, IFC 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, accessible through a high level API. Higher level considerations and policies can be managed through specifically defined Trusted Processes (see §4.2). The architecture is presented in Fig. 5.

### 4.1 CamFlow-LSM: OS enforcement

Our kernel module, *CamFlow-LSM*, is implemented as a Linux Security Module (LSM) [27]. 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 [24], [26] 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. For example, Laminar [26], mainly designed to support an IFC-enabled Java VM, modifies over 500 lines of code across the kernel in order to leverage its LSM. This large kernel modification renders Laminar hard to maintain, and represents non-trivial engineering to port to a new kernel version. By comparison, updating our LSM from kernel version 3.17.8 to 3.18.2 required only five lines of code to be changed, related to an unavoidable need to conform to a kernel API modification.

Since applications running on SELinux [28] or AppArmor [29] 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. This implementation choice is important; cloud providers can incorporate IFC without requiring changes in the software deployed by tenants. Alternatively, policy may be declared by applications through a pseudo-filesystem (as is typical for LSMs) abstracted by a user space library and enforced transparently by the IFC mechanism.

Tags and privileges are represented by 64 bit integers (standard in such systems since Flume [24]). They are stored in an opaque field of the appropriate Linux kernel object (corresponding to processes, inodes, files, shared memory objects, etc.). Only active entities (processes) have mutable labels and privileges, all other (passive) entities have immutable labels and no privileges.

As shown in §3 every label change must be explicit and child objects inherit their parents' labels. File labels are persisted across executions and are stored as extended attributes (as for other LSMs such as SELinux [28]).

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. A process can add or remove a tag from its label if it owns the appropriate privilege, 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 Co-existing with Other Security Modules

Support for simultaneous, multiple Linux Security Modules is being proposed via various kernel patches [30], [31]. The CamFlow-LSM implementation is strictly self-contained, and designed to compose gracefully with other LSMs. Previous work [26] that modifies system calls, would need to rewrite their LSM, among other things, for use with other LSMs. We believe our approach brings practical benefits, as again, we do not see IFC as replacing existing security mechanisms (e.g. SELinux [28] or AppArmor [29]), but rather as a means to provide additional security functionality. Further, minimising the deployment overheads helps to facilitate and encourage wider uptake and adoption.

#### 4.1.2 Checkpointing and Restoration

Checkpointing a process involves halting its execution, allowing it to be restarted at a later stage, and enabling migration, see [32], [33]. LSM state is normally saved and restored by the checkpointing system (e.g. [34]) and our module further exports an API to more efficiently serialise and restore security context. However, as described

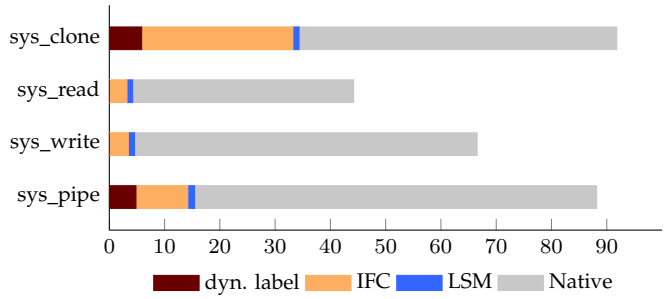


Fig. 4: Overhead introduced into the OS by the CamFlow LSM (x-axis time in $\mu$s).

earlier and unlike Niu et al. [35], we do not modify system calls and therefore it is not necessary to modify existing software in order to run on our platform.

Furthermore, self-checkpointing and restoring the previous state of a process, has been demonstrated [35] 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 after initialisation. This improves performance and prevents data leaks between security contexts.

#### 4.1.3 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 Intel 2.2Ghz i7 CPU and 6GiB RAM machine.

Measurements are done using the Linux tool ftrace [36] 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.

In some previous work [19], [24], IFC was introduced into OS kernels by interposition techniques for which overheads were multipliers. 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 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). 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.

3. It is not feasible to provide a comparison with the Laminar implementation [26], 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).

### 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.

Therefore, 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 §4.4), for checkpointing and restoring processes (see §4.1.2) and for managing inter-process and external communication (see §5).

Figure 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 communication. Such communication may be within or between containers, operating systems 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, e.g., as in IoT, tags may need to be protected by cryptographic means (see §5.3).

Trusted processes are either set up through static

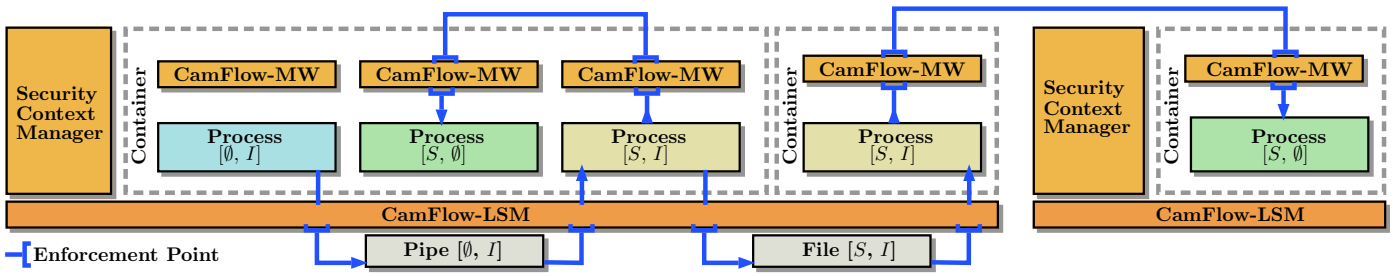


Fig. 5: CamFlow Architecture: Labelled OS Objects and Trusted Processes

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 §4.5).

### 4.3 Outside Connections

In order to guarantee flow constraints, 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 to the outside world, a process must either: 1) be able to declassify to $S = \emptyset$; 2) communicate through an intermediate trusted process.

To build a PaaS platform, we developed a message-passing middleware (§5) which, as a trusted process, allows data exchange between applications and services while guaranteeing IFC properties. In §7, we give another example of such a process through a gateway that authenticates clients and routes requests to the application instances running within the proper security context.

### 4.4 Integrating with Persistent Storage

A first technique to provide IFC with data stores comes directly from work, including our own, on library-provided IFC [19], [37]. In such work, the tags are stored within the persistent store alongside the data, and a trusted software component ensures that when information is read from the store, the corresponding labels are applied. In Flume [24], 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 [21]. Our work [38], [39] integrates a messaging middleware with database queries which, coupled with IFC-aware databases, could provide a clean implementation minimising the TCB. Moreover, IFC can be enforced by middleware and storage without application intervention. The CamFlow-middleware §5 interfaces between storage systems and the rest of the platform. The middleware behaves as a proxy, translating system-wide labels into their data-store specific representation (similar to how a string or cryptographic tag representation of tags is translated into an OS identifier).

Note that there is a need to extend the trust of coarse-grained labels at OS kernel level to this fine-grained implementation of IFC in the database, over which the kernel itself has no control. A first scenario is that a whole database might be labelled at coarse granularity as, say, medical, research. A second scenario is to use CamFlow-MW to mediate between a trusted/auditable IFC-enforcing database (such as IFDB [21]) and IFC constrained applications running on an CamFlow-enabled OS. We are continuing to work in this area.

### 4.5 Leveraging hardware to reinforce trust in the platform

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 have made it possible to attest that the software layers on which our platform runs are those that have gone through an audit process.

The Trusted Platform Module (TPM) [40], as used for remote attestation, [41] 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 [42] are reaching maturity, with IBM rolling out an open source, scalable trusted platform based on virtual TPM [43]. Indeed, Berger et al. [43] describe a mechanism allowing 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 CamFlow-Middleware

CamFlow contains a fully-featured general messaging middleware that 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 of the middleware is its ability to support 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.

For want of space, we only consider the middleware concepts relevant to IFC enforcement. Full technical details on the general middleware (as it was prior to IFC/CamFlow integration) can be found in [38], [44].

In CamFlow, the role of the middleware is to move towards end-to-end data flow management, such that IFC can be enforced *across* applications/machines (kernels). There is some work on IFC enforcement across machines; however, these impose specific requirements, such as design-time considerations [45], a particular language/runtime [46], or constraints on system architecture/implementation [47], [48]. In contrast, we integrate IFC functionality into a general, fully featured distributed systems middleware, to enable 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 CamFlow: Message-level enforcement

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, primitive-type* and *value*. The support for IFC within messages is fine-grained, in that individual attributes within messages are labelled.

Hierarchical message structures are supported, meaning an attribute might contain a number of sub-attributes (*children*), e.g. similar to XML. Thus, the information flow of a child $c$ can only be more restricted than its parent $p$; $S(p) \subseteq S(c)$ and $I(c) \subseteq I(p)$. All child attributes are subject to the same labelling constraints, with the top-level attribute referring to the label of the message type.

Labels can be defined within message type schema, which sets the attribute's IFC label for all message instances of the type, i.e. this cannot be changed by entities dealing in such messages, and the entities must hold the requisite labels to interact using messages of that type. Otherwise, the *entity* (process/task using the middleware for communication) producing/publishing a message can set the security labels for the attributes (for those not predefined), if the entity holds the associated privileges.

If an entity does not assign a label to an attribute, the middleware sets this label to the entity's current label subject to any definitions in the message type. In this way, applications can be subject to IFC enforcement completely transparently (i.e. without their direct involvement); while also providing the interface for the application to be actively involved if required.

**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:** A sending entity cannot produce an attribute whose labels do not agree with its own labels. This is enforced when an entity attempts to send a message, ensuring values for any attributes violating this policy are removed, before message propagation.

### 5.2 Policy-driven, event-based reconfiguration

In some circumstances, it may be necessary to modify the allowed flow of information. For example, the general policy for a tenant may be to restrict the flow of personal data to within the European Union, but in case of failure in a data centre the service may be temporarily hosted in a US data centre to guarantee availability. Another example in a medical context is the detection of a life-threatening event regarding a patient, where data restrictions are relaxed and patient data automatically disclosed to the emergency services [7].

These examples illustrate a necessary change in system behaviour (e.g. restricting data to EU territory, not disclosing patient data) in order to maintain quality of service (e.g. maintaining availability) or responding to other functional requirements (e.g. protecting patient life). Middleware can encapsulate the policy to automatically effect these event-based reconfigurations, at runtime, when the circumstances arise. Such implicit change occurs through well defined policies and should be considered with extreme care. Indeed, they break the rule of *no implicit context change* (§3.4.2) and may disclose data to parties outside of the normal system behaviour.

It again follows that cloud applications and services need not be IFC aware (though they are not precluded from taking an active role). The middleware sets the labels to the current runtime levels of the entity, to appropriately manage message exchange without application intervention. As such, system components are subject to IFC enforcement policy, without requiring any direct involvement with IFC specifics. 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.

### 5.3 System-wide, Secure Label Representation

For IFC to be enforced across machines, tags must be managed. Towards this, we have proposed that the widely used and available X509 certificates could repre-

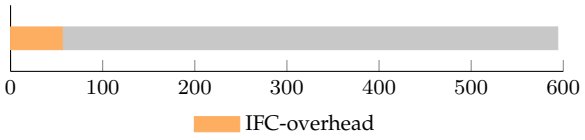


Fig. 6: Average IFC overhead for the CamFlow-MW for a workload transmission of 5000 messages (x-axis in ms)

sent tags, when in transit outside trusted closed systems, see [49]. The approach relies on public key certificates and attribute certificates [50], and uses cryptographic means to support IFC-controlled exchange of data across a range of federated applications and clouds. Data exchanges are only allowed from sources belonging to the federation, which is verified through the PKI.

### 5.4 Evaluation

We have previously evaluated the overhead that IFC brings to our messaging middleware—this is detailed in [39]. In summary, the results indicate that IFC enforcement introduces an average overhead of ∼9% in performance time compared to the standard, non IFC-enabled middleware. 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 may be less onerous.

## 6 CamFlow-Audit: Data-Centric Logs

IFC, in addition to providing strong assurances that policy is being enforced, can also provide a data-centric log [51] detailing the information flows within and between system components. Cloud logging systems are generally based on legacy logging systems (OS, webserver, database etc.) that either fail to capture the needed information, or are extremely complicated to interpret in a useful manner [52]. More importantly, such logs tend to be relevant only to the particular service or component, which makes 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. Moreover, such audit brings a level of transparency allowing the potential of IFC (§1) to be demonstrated: (1) mutual protection of applications; (2) data sharing according to policy; (3) investigation of data leakage; (4) system-wide access control policy enforcement; (5) compliance with regulations and contracts. In addition, attempted security attacks and possible security breaches (whose effects may be confined by IFC) can be investigated. The existence of audit enhances trust in cloud services.

Any monitored system call issued by a labelled process can be logged, along with middleware operations concerning connections and message transmission. Operations on labels are also recorded. We have defined

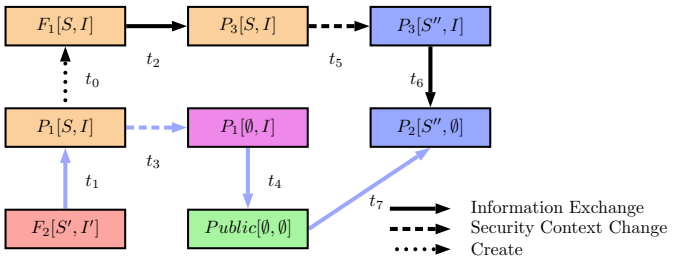


Fig. 7: Simplified audit graph from IFC OS execution (we omit meta-data for readability). In blue/pale path to disclosure.

several different types of flow as part of our IFC model (see §3), namely data flow, creation flow, security context change and privilege delegation. These flow types may be important in security forensics and are recorded as part of the audit log. In Table 1, we show the type of information recorded: unique IDs representing the origin and destination entities, the labels associated with those entities, whether the flow was allowed, the type of flow and the corresponding timestamps and further metadata that may vary depending on the entity type. In a middleware context, the message types involved, and even details of the messages can also be recorded. All of this allows an auditor to 1) trace information flows within, across and between applications and clouds; and 2) to examine which applications are attempting to violate IFC constraints and further investigate why.

### 6.1 Analysing paths to disclosure

To assist in interpreting log information, it is possible to build a directed graph corresponding to the allowed flows during the execution of our system. An illustration is shown in Fig. 7. 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, 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 timestamps associated with the edges comprising the path. We denote by $t_e$, the last incoming edge to the entity under investigation with labels $[S',I']$; only edges such that $t < t_e$ should be considered. When applied to all nodes along a path, this rule ensures

| origin | origin labels | destination | destination labels | permitted | timestamps | type | origin metadata | destination metadata |
|---|---|---|---|---|---|---|---|---|
| $P_1$ | [S, I] | $F_1$ | [S, I] | true | $t_0$ | create | {uid, pid [...]} | {filename, [...]} |
| $F_2$ | [S′, I′] | $P_1$ | [S, I] | true | $t_1$ | exchange | {filename, [...]} | {uid, pid [...]} |
| $P_3$ | [S″, I] | $P_2$ | [S″, ∅] | true | $t_6$ | exchange | {middleware, endpt [...]} | {middleware, endpt [...]} |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| $P_1$ | [S, I] | $F_3$ | [S‴, I‴] | false | $t_8$ | create | {uid, pid [...]} | {filename, [...]} |

TABLE 1: Sample audit log structure

strictly monotonically increasing timestamps from the first node to the last. Fig. 7, shows in pale blue a possible data disclosure path, from file $F_2$, from a very simple audit graph. We know from the timestamps $t_0$ and $t_1$ that the data disclosure did not occur through file $F_1$ and process $P_3$, but through $P_1$'s declassification.

### 6.2 Analysing data provenance

In §3, Fig. 2, we can see that IFC is guaranteeing that for medical records to flow to a research database (and subsequently to the corresponding research portal), patient consent is verified and the data is anonymised. We can guarantee by using IFC that this pattern is followed.

However, an investigator may want to know which anonymisation algorithm has been run, which data has been used to generate the anonymised records etc., questions relating to *data provenance* that arise from our IFC audit log. Provenance can be described as *metadata that represents the history of an object* [53]. Provenance systems are used, for example, in data forensic analysis to determine how a certain file has been generated [54]. Indeed, with the metadata stored in our audit graph, it is possible to reconstitute the history of a record.

It has been suggested that provenance systems could be used to retroactively enforce IFC constraints [20]. Here we demonstrate that conversely, audit logs generated by IFC systems can be used to provide a provenance system.

### 6.3 The audit log 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 [55].

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. For example, 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 the data 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 be considered as 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 can also be leveraged. For example, an auditor, before being granted access to audit logs, must demonstrate ownership of the corresponding secrecy IFC tags (for example through cryptographic means as in §5.3). 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. Our platform runs classic and unmodified Ruby web applications.

As discussed in §4, a labelled process cannot interact directly with standard externally-facing sockets and must interact with the outside world through a trusted middleware process. Similarly, interaction with cloud services (such as data stores) is also achieved through our messaging middleware as discussed in §5 and §4.4.

Interaction with clients is achieved through a "gateway" between the IFC and non-IFC worlds. 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 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 §4.1.2.

Several application types are running over our cloud-based, web services platform. For example, in a medical

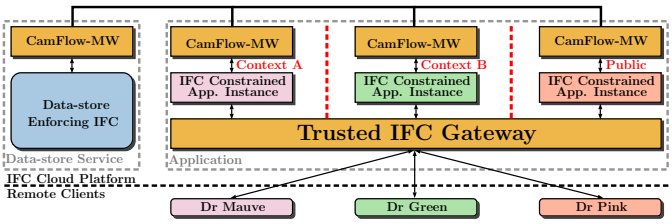


Fig. 8: PaaS Architecture on top of IFC-OS

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 [56]).

Here, we use OAuth identities to correspond to the IFC tags required by applications. The aim is that the same patient identity (and corresponding IFC tag(s)) can be used by a doctor, authorised as treating that patient, to access multiple applications such as medical record editing etc, on behalf of Alice. We assume such applications, in order to be used on Alice's behalf, have secrecy tags medical, alice (as in §3, Fig. 1) and, for simplicity, we assume no integrity tags.

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 $S = [\text{medical}, \text{alice}], I = [\emptyset]$ 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. Emergency access to data is covered by event-based reconfiguration as discussed in §5.2 (for example granting any doctor privileges over Alice's tags).

Note that 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

In §1 we described 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) increased data flow transparency, aiding data management, the identification of policy errors/misconfigurations, and providing evidence for compliance with contractual/regulatory requirements.

We have demonstrated the feasibility of providing IFC at the OS level within IaaS or PaaS cloud services and thus underpinning SaaS. We presented a new kernel implementation of IFC as a LSM, demonstrating low overhead for worst-case scenarios where processes continuously make read/write system calls. Our IFC model and LSM implementation are designed so that applications can run unchanged over IFC, thus making cloud adoption feasible, particularly for smaller companies.

We built a web service support framework to show that if the platform is trusted to deploy IFC correctly at the OS level (perhaps with hardware verification of lower levels) then applications are constrained to run in user and platform-defined security contexts. Such an approach supports sharing between applications running in the same security context and prevents unauthorised data flows between security contexts by malicious or buggy/misconfigured applications.

The example begins to address the extension of IFC beyond a single cloud. We showed a trusted IFC gateway (§7), designed to set up security contexts for external users. Should such users be running in a trusted, compatible IFC environment, the gateway would not be needed. In terms of operating within, across and between systems, we envisage IFC being enforced on the end-user device, together with the IFC-aware messaging layer described in §5 and [39]. In such a scenario, the user's end-device process could communicate with the cloud-side process directly through an IFC-secured channel.

CamFlow was developed with cloud deployment in mind. When multiple cloud services become part of a wider distributed architecture, such as in IoT, a trustworthy deployment of IFC can no longer be assumed outside of the immediate cloud context. When data leaves the environment of a 'thing's' owner, IFC gives the potential of controlling where it may flow subsequently. Towards this, we intend to extend CamFlow to support mobile environments. This is feasible: Android now supports the full enforcement of SELinux,[4] and an Android-LSM implementation has been demonstrated [57].

A scheme for managing tag definitions (tag naming) is required for wide-scale IFC enforcement. Though clearly beyond the scope of this paper, initial thoughts on how tags might be used to manage the location of data, according to EU legislation, are given in [11]. We envisage agreements on tag names, associated with legislation,

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

and domain-specific naming, e.g. for Facebook 'friends' or as we have defined for our projects with the UK National Health Service [22], [23]. The cryptographic representation of tags (§5.3) appears useful, particularly in dealing with issues of ownership, e.g. user-owned devices in IoT, and/or multiple domains, where different organisations could maintain both separate and interoperable tagging regimes. More experience on designing and using IFC labels is required; however, it appears that a worldwide tag naming scheme would require support akin to that provided by DNS.

A related issue concerns tag sensitivity, i.e. whether the tags themselves leak information. This no doubt depends on the mechanisms for tag management.

IFC logs provide the means to audit an IFC-enabled system, thus demonstrating trustworthy behaviour. Logs can be processed to show compliance with contracts and regulations, to investigate leaks and attacks and, in general, to show that data has been managed in accordance with policy, even after it has left the control of its owner. More work is needed on regulating and managing access to audit information, though as detailed in §6 we have made progress towards this.

IFC allows data flows to be controlled system-wide, end-to-end, creating more powerful and comprehensive access controls than traditional authentication and authorisation. We believe that IFC has great potential as a security mechanism whereby trust in a few major cloud providers, deploying IFC, can be built on to provide a demonstrably trustworthy computing environment.

## 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," *Cloud Computing*, 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] J. Singh and J. Bacon, "On Middleware for Emerging Health Services," *Journal of Internet Services and Applications*, vol. 5, no. 6, pp. 1–34, 2014.

[8] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov, "$\pi$Box: A Platform for Privacy-Preserving Apps." in *10th USENIX Symposium on Networked System Design and Implementation*, 2013, pp. 501–514.

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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 *Proc. 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, 2010, pp. 1–6.

[17] 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.

[18] 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*. Berkeley, CA: IEEE, 2010.

[19] T. F. J.-M. Pasquier, J. Bacon, and D. Eyers, "FlowK: Information Flow Control for the Cloud," in *6th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2014.

[20] 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.

[21] 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.

[22] P. Hosek, M. Migliavacca, I. Papagiannis, D. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch, "SafeWeb: A Middleware for Securing Ruby-based Web Applications," in *Middleware*, 2011, pp. 491–512.

[23] T. Pasquier, B. Shand, and J. Bacon, "Information Flow Control for a Medical Web Portal," in *e-Society 2013*. IADIS, 2013.

[24] 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.

[25] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proc. 7th USENIX OSDI '06*, 2006, pp. 19–19.

[26] 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.

[27] 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 Computer Society, 2003, pp. 213–213.

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

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

[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, 2004.

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

[32] 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," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[33] "CRIU." [Online]. Available: http://criu.org/

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

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

[36] T. Bird, "Measuring Function Duration with ftrace," in *Japan Linux Symposium*, 2009.

[37] B. Davis and H. Chen, "Dbtaint: cross-application information flow tracking via databases," in *Proc. 2010 USENIX conference on Web application development*, ser. WebApps'10, 2010, pp. 12–12.

[38] J. Singh and J. Bacon, "SBUS: A Generic, Policy-enforcing Middleware for Open Pervasive Systems," *University of Cambridge Computer Laboratory Technical Report TR*, vol. 847, 2014.

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

[40] B. Parno, "Bootstrapping Trust in a" Trusted" Platform," in *HotSec*. USENIX, 2008.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

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

[47] 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 *Proc. USENIX Annual Technical Conference*, Boston, 2012.

[48] 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.

[49] 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.

[50] J. S. Park and R. Sandhu, "Binding Identities and Attributes Using Digitally Signed Certificates," in *16th Annual Conference on Computer Security Applications*. IEEE, 2000, pp. 120–127.

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

[52] 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.

[53] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, "Provenance for the Cloud," in *Conference on File and Storage Technologies*. USENIX, 2010, pp. 15–14.

[54] R. Lu, X. Lin, X. Liang, and X. S. Shen, "Secure Provenance: the Essential of Bread and Butter of Data Forensics in Cloud Computing," in *Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 282–292.

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

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

[57] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android." in *NDSS*, 2013.


**Thomas Pasquier** is a PhD student and a Research Assistant at the University of Cambridge. 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.