Investigating the Potential of Information Flow Control as a Cloud Security Technology

Jean Bacon, *Fellow, IEEE,* Thomas Pasquier, *Member, IEEE,* Jatinder Singh, *Member, IEEE,* Olivier Hermant and David Eyers, *Member, IEEE*

Abstract—The CloudSafetyNet project is investigating the potential of Information Flow Control (IFC) as a technology for cloud computing. Security concerns are widely seen as an obstacle to the adoption of cloud computing solutions and although a wealth of law and regulation has emerged, the technical basis for enforcing and demonstrating compliance lags behind. IFC complements traditional security technologies, such as principal/role-based access control, applied at policy enforcement points. For authorised principals, IFC adds continuous monitoring of information flow based on the properties of data and the context. This paper presents our work on OS (system-call) level IFC enforcement. We describe an enhanced label model to better capture tenant/end-user information flow constraints, the integration of IFC with a messaging-middleware to enable end-to-end control, and audit capabilities for demonstrable compliance.

Keywords—Information Flow Control, Audit, Messaging Middleware

1 Introduction

The CloudSafetyNet project investigates the potential of Information Flow Control (IFC) for distributed and cloud computing; [1] gives an overview of IFC models and implementations although none have discussed cloud deployment. Concern about data leakage is holding back more widespread adoption of cloud computing by companies and public institutions alike. There is an increasing volume of legislation [16], but ensuring and demonstrating compliance with the legislation by cloud service providers and third parties is problematic. We believe that the deployment of IFC to augment traditional security technologies will make a substantial contribution to the security of distributed and cloud systems, both through enforcement mechanisms and demonstration of compliance through audit. Note that we will use the term IFC generally, to subsume Decentralised (D)IFC, see §3.

IFC can be provided independently of cloud-provider software, either as a language feature [17], [31] or through a library [20], [23], [35]. A great deal of work has been carried out at this level, including storage/database integration [29]. But the trust assumptions of this level of deployment are huge. First, correctness depends entirely on the application implementing the policy correctly, and this is where most errors arise. Also, even if the cloud provider is assumed to be trustworthy, data leakage could still occur within the cloud implementation due to bugs, over-permissive data access policies or attacks of various kinds. We therefore decided to investigate the provision of IFC within the cloud software itself.

We considered IFC provision at a number of levels in the cloud software stack: within Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) clouds. In order to enforce IFC for IaaS we would need to implement IFC in the hypervisor. However, VM isolation techniques are strong at this level and there would most likely be reluctance to rely on IFC instead. IFC mechanisms akin to those discussed in this paper could be put in place by tenants in their own VMs. It is becoming relatively easy to use a hardware-verified hypervisor substrate [13] and we would hope to build on such a layer.

We believe that the logical place to provide IFC for PaaS and SaaS is within the OS and middleware layer, while aiming to make it transparent at the cloud application level. This removes the reliance on applications being implemented correctly and supports intra-tenant as well as inter-tenant data sharing and data protection.

We first present **FlowK** (Information **Flow** Control **K**ernel Module), a kernel module for enforcing IFC within a standard Linux OS, as used by most cloud service providers. FlowK intercepts all system calls that create information flows and enforces the FlowK IFC constraints on those flows. IFC is achieved by associating labels with data and the entities that process them. Labels comprise a number of tags that describe the nature and/or source of the data such as healthcare, personal, government-information etc. Flows are permitted only if the labels match, as defined in §5.

Our FlowK label model follows that used in IFC systems since 1997 [17], following best practice, not requiring OS change and minimising changes needed in cloud applications. The design of FlowK follows the principle of "policy-mechanism separation", in that the FlowK kernel module is concerned only with enforcing rules relating to entities' labels. Labels are assigned to entities after successful authorisation. Any application that does not use IFC is unaffected (subject to a small performance hit on system calls of around 10%, see §10).

Traditionally, (principal/role-specific) access control is applied at policy enforcement points within applications, after which no further control is exercised on where that data flows in the system. It is therefore subject to leakage and misuse through misconfiguration, bugs or malicious intent. IFC augments access control so that data flows are monitored continuously, in context, to enforce more general policy, capturing properties of the data as well as the authorised principal. The challenge in integrating access control policy with IFC is to create labels that express policy accurately, concisely, naturally and efficiently, and in a way that makes policy changes easy to implement.

We have considered how labels can be used to enforce certain laws and regulations, such as "data originating in the EU must not leave its boundaries, except to certain Safe Havens" [16]. A simple tag *EU* can be used for this purpose. We found access control policies to have more richness and complexity, and difficulties arise in representing some policies via IFC labels. Essentially, problems arise when some software can access all data items of a certain kind, for example to generate statistics or anonymise, and other software is restricted to access only one such item. In work on Role-Based Access Control (RBAC), by ourselves [2] and others, parametrised roles were used in these circumstances to avoid proliferation of the number of required roles.

We decided that some modifications were needed to improve the expressiveness of tags and to express correctly the trust that should be placed in entities according to the Principle of Least Privilege (PoLP). The FlowK2 label model comprises two-component tags to represent the *concern* of data and a *specifier* for an item of that kind, for example $\langle medical, bob \rangle$ for Bob's medical record. A subtyping relation allows us to use $\langle medical, * \rangle$ for all principals' medical data and $\langle *, bob \rangle$ for data of any kind tagged as Bob's. Such labels more closely reflect the policy maker's intent than previous models, and current single-component tags can easily be expressed in this way, such as $\langle location, EU \rangle$ or $\langle regulation, EU111 \rangle$. We will consider incorporating this tag model as part of future work on a tag naming scheme for distributed IFC.

The point at which IFC is enforced in FlowK is the natural place to create an audit of allowed and forbidden flows. We have designed and deployed a proof of concept audit tool, see §7. The aims are: (1) to support post-hoc digital forensics; (2) to demonstrate compliance with tenant/provider contracts; (3) to detect bugs in programs that try to perform illegitimate operations; and (4) to evaluate policies and detect unaccounted flows.

To demonstrate application integration with FlowK we have adapted a framework for cloud-deployed web services to be IFC-compliant. An implication of this work is that untrusted/unverified applications can be run on a trusted cloud platform without compromising their end-users' data, without using strong isolation between users of an application.

To extend IFC to system-wide operation we have extended our SBUS middleware [32], [33] to include IFC enforcement. SBUS is a messaging middleware and IFC labelling is incorporated within structured messages at the attribute (field) level. The FlowK architecture comprises a kernel module with User-Space Helper processes (Ushers); each process has such a helper to support external communication.

The contributions of CloudSafetyNet are (1) FlowK: an implementation of a current-practice IFC label model, modified to minimise the reengineering required by its users and to support SoD [22]. (2) Creation of an audit log and processing tools together with the ability to specify access control and IFC policy for the log. (3) FlowK2: an enhanced but compatible label model that captures trust and application policy more precisely. (4) Integration with the application level via an IFC-adapted webworkers architecture [22], and (5) integration of IFC-enabled middleware as a basis for cloud-wide, inter-cloud and general, distributed communication.

We first state our trust assumptions in §2, then introduce related work in §3, outlining IFC models, mechanisms and previous deployments. We also summarise authorisation policy. In §4 we motivate two-component tags in FlowK2. In §5 we present the basic FlowK IFC model, integrating the enhancements needed for matching two-component tags in FlowK2. In §6 we give details of the implementation of FlowK and FlowK2. In §7 we describe how we have supported audit in the FlowK kernel module. In §8 we demonstrate how web applications can use FlowK. §9 discusses the integration of our SBUS-IFC middleware with FlowK. §10 provides and discusses performance measurements. §11 discusses work in progress, summarises and concludes.

2 Trust Assumptions

We assume the *cloud provider* to be non-malicious and bound through legal requirements to do its best to protect its tenants' (end-users') data. Even so, the cloud provider infrastructure could be misconfigured and leaks could happen through shared infrastructure.

We assume that *tenants* running applications in the cloud do not actively try to leak their users' data. Even so, an application provided by a tenant may contain bugs that can be abused to steal users' data or intentionally leak data. In current infrastructure, a tenant's data store containing all its users' data, may not be securely isolated; bugs might cause inter-tenant or intra-tenant leaks.

Among the most common attacks on web servers are URL interpretation attacks, malicious file execution, injection attacks and buffer overflow. IFC compartmentalises risk by confining the effects of such attacks. This is achieved by restricting the flow of data an application instance is allowed, according to its security context. This security context can be defined on a per-user basis or may correspond to roles, as most appropriate for the application, thus reducing the potential attack surface.

IFC can also be used to enforce the integrity of data used by the application, preventing for example script injection or SQL injection, see §5.

In the web server architecture in §8, we do not address insider attacks. However, we believe this risk could be addressed with a combination of encryption techniques and IFC, but this is beyond the scope of the work presented here. We also do not protect individuals or end-users' machines. These are the target of social engineering, cross site scripting, cross site request forgery and exploitation of browser vulnerability. Existing techniques can help prevent such attacks contributing to overall security. The integration of IFC-enabled middleware with cloud software provides the possibility of end-to-end IFC, including end-user systems.

Our aim is that applications with a requirement for data confidentiality and integrity, such as those handling medical records could be safely hosted on a trustworthy public cloud. In previous work [21] we used IFC in building a web portal for brain cancer patients with the background database hosted on a secure server farm, exclusive to the UK's NHS.

3 Background and Related Work

The CloudSafetyNet project is investigating the potential of IFC enforcement end-to-end, system-wide: within and between containers and Virtual machines (VMs); intra-cloud and inter-cloud; and between end-users and clouds. In this section we give a brief evolution of IFC, describe work on IFC enforcement at the OS level then outline approaches to distributing IFC. To our knowledge, cloud deployment of IFC has not been addressed.

3.1 IFC Models

In 1976, Denning [9] 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 [3] 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) during any *reading down* and *writing up*, as proposed by Biba [5]. 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 [18] introduced a Decentralised IFC model (DIFC) that has inspired most later work, including FlowK. 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 are allowed to flow if the security label of the sender is a subset of the label of the receiver, and conversely for integrity. We describe in §5 the model we use in FlowK that derives from this general idea.

3.2 IFC in Operating Systems

Other research projects have implemented IFC constraints at the OS level, most notably in Flume [14]. Here, a model similar to Myers' original one [18] is used, and the entities considered are files, pipes, sockets and processes. In Flume, monitored (labelled) processes have access to a restricted set of system calls, and some (such as fork and pipe) are completely replaced by IFC-specific ones. This means that Flume applications running under IFC constraints need to be rewritten, even when they do not need to manipulate IFC labels during their life-cycles.

The Asbestos OS [11] implements the *send* and *receive* label paradigm, while also proposing a solution to improve the performance of IFC, which relies heavily on the fork operation to build applications. Asbestos is a rewritten OS rather than an imported module, requiring substantial changes in software that uses it. HiStar [37] extended Asbestos with security enhancements and a user-space library to emulate a Unix-like OS interface.

In Aeolus [6] and Laminar [27], an IFC aware operating system is used to enforce inter-process IFC constraints and a modified Java Virtual Machine ensures intra-process isolation via programming language objects. Again, it is necessary for application developers to be IFC-aware and to rewrite their Java programs.

3.3 IFC in Distributed and Cloud Systems

There has been some work considering the communication aspects of IFC. Component Information Flow [30] is a design framework for component-based system architectures where security constraints (labels) can be specified on communication interfaces. It provides tools for model validation and code generation.

To achieve communication more dynamically, DIFCA-J [36] modifies Java bytecode to enforce IFC throughout the JVM, including remote method calls. External objects (files, databases) can be labelled, in order to regulate flows to and from the JVM. Aeolus [6] uses abstractions to control data flows in a distributed system, where IFC is enforced against interactions between Aeolus nodes (isolated applications), boxes (shared objects) and the custom (label-aware) filesystem. Interactions with entities (files, applications, etc.) outside these abstractions are untrusted, thus unlabelled. In DStar [38], each machine has a dedicated exporter component, through which all inter-machine communication occurs. Leveraging an IFC-compliant OS, e.g. Flume [37], the exporter translates between local machine and global labels, to regulate flows across machines.

Our focus is unique in that it aims to integrate IFC functionality into a general, distributed communications middleware, SBUS [32], [33]. Further, we see that work on IFC in networked environments tends to impose constraints on system design, architecture, implementation and/or the operating environment. A deliberate design decision was not to impose a structure on system design; but rather to integrate IFC functionality into the kind of communications infrastructure already common to enterprise systems. In addition, we also aim to provide more flexible, finer-grained control, by enforcing IFC at the level of message-attributes (data fields), cf. other work that concerns the entire channel or message.

4 Policy Requirements for IFC

In this section we consider how IFC labels can embody application policy. We assume a cloud computing environment and envisage that our model will ultimately be used across a whole cloud platform. In such a scenario there is likely to be a large amount of user data with persistent labels stored in files, databases, key-value stores etc. In a company context, data records may relate to individual employees; in a public health context, data may be the medical records of patients; in an educational context, data may relate to students, staff etc. Specifying and enforcing access to only one, some specified subgroup or all data records of a given type is a common requirement of authorisation policy.

Our experience is that enforcing access to a specified subgoup of entities' data is best carried out as part of authorisation [2]. For example, doctors may be able to access only the records of the patients they are currently treating; teachers, those students they are currently teaching, etc. Temporally separated instantiations of the application are likely, i.e. to one patient's record at a time. Each time, current authorisation policy is enforced and is translated into the labels required by an instance of application software to access one patient's data. Indeed, separation between the patients' data is desirable; there is no need for subgroup enumeration to be represented in the runtime labels. Also, note that since a doctor's group of patients under treatment changes, and student groups change, a lookup of current patients/students will ensure that a label is created only for access to a current patient or student record (assuming that is the policy), selected at runtime from the entire database. The FlowK model can be used, together with application-level policy, to implement such subsets.

The remaining requirements from access control are specifying and enforcing access through labels, at runtime, as follows:

1) To one principal's data, as for a parametrised role, e.g. medical(bob) at authorisation policy level. The basic IFC model achieves this by associating the separate tags medical and bob with the data.
2) To records of a given type for all principals, as for a parametrised role, e.g., medical(*). The basic IFC model can only achieve this by enumeration of all the principal's tags that have the medical tag.
3) To be able to express e.g. "all Bob's personal data of any kind". The basic model can only achieve this by enumerating all the tags that are associated with the tag bob in any label being matched.
4) To access all data in some naming context.

Examples of requirement 2) arise from the need to support software to perform computations across large numbers of data records whereas other software is authorised only to access records on behalf of a single individual. Taking medical data as an example, data will be tagged with both medical and bob for Bob's record. An entity carrying out statistical analysis of medical data would need to have not only the medical tag, but also a tag corresponding to every patient's data. This is computationally and administratively infeasible as well as raising consistency issues regarding the enumerated tags compared with the stored data. In practice such software would most likely run with IFC disabled. Similar problems would arise in "big data" processing, when setting up a MapReduce job. Mappers require access to all data of a certain type and set up tags on their output data for Reducers. Reducers might create output with a tags such as medical, statistics in the S label, depending on the application's naming scheme for tags.

Requirement 4) arises when a system log as described in §7 contains records from all running applications relating to large numbers of principals. The requirement here is that access can be granted to all or selected parts of the log as appropriate. For example, digital forensics after system penetration is suspected may be required to check all the log for attacks or malpractice, in role notation *(*); in practice, such software would most likely run with IFC disabled. Applications may audit their own records in the log for all their users app(*); individuals may be authorised to access only their own records across all applications they have used *(bob).

The examples above (in parametrised role notation) show the benefit of having two-dimensional tags, able to represent $\langle tag\_type, specifier\rangle$ to at least mirror parametrised roles in RBAC. Examples are $\langle medical, bob\rangle$, $\langle employee, bob\rangle$, $\langle student, bob\rangle$, etc. More general examples are $\langle *, bob\rangle$, $\langle *, *\rangle$, $\langle medical, statistics\rangle$, $\langle medical, anonymised\rangle$ etc.

In §5 we describe the basic FlowK model where the flow rules carry out matching of tags, each representing a single security concern. For each rule we show the extension required to match two-component tags in place of atomic tags. In practice, having this extended tag model allows policy to be expressed more concisely as well as more accurately.

This idea is extensible to a general n-component scheme for tags, thus carrying through to runtime the full expressiveness of parametrised RBAC, see [2], such as

$medical(hospital\_ID, doctor\_ID, patient\_ID, allow\_research)$. However, such a scheme adds complexity to the label matching algorithm and could well impact performance.

5 FlowK and FlowK2 IFC Models

IFC extends (principal/role-specific) authentication and authorisation by enforcing dynamically, that only permitted flows of information can take place, adding data characteristics and context as aspects of flow control.

Flows are enforced by means of labels. We assume that labels are associated with entities after successful authorisation and here define how labels control information flow. We introduce a notation for the different operations related to information flow, which we use to describe succinctly the behaviour of an IFC system.

A tag within a label's set of tags represents a particular security concern for a category of data. In our IFC model two labels are associated with every entity A: a *secrecy label* $S(A)$ and an *integrity label* $I(A)$. The current state of these labels (sets of tags) is an entity's *security context*.

In this section we define a secure system.

Definition 1. *The IFC system is secure if and only if all allowed information flows are safe (Definition 2) all allowed label changes are safe (Definition 3) and all privilege delegation is safe (Definitions 4 and 5).*

In all previous models, tags have been atomic, representing a property of the data or an allowed context, e.g. *medical*, *personal-bob*, *encrypted*, *anonymised*, *user-input*, and flow controls have checked subset relations in source and destination labels' tags. In FlowK2 we have extended our tag structure to capture application policy requirements more precisely (to capture trust in accordance with the PoLP), and to avoid tag enumeration, as motivated and defined in §4.

We propose to decompose a tag t into a pair $\langle c, s\rangle$ with c the concern of type $\mathscr{C}$ and s a specifier of type $\mathscr{S}$; i.e. FlowK2 has two-component (2D) tags. For example, the pair $\langle medical, bob\rangle$ represents Bob's medical data. A statistical analysis over a set of patients' medical data is represented as $\langle medical, statistical\_analysis\rangle$ and anonymised medical records as $\langle medical, anonymised\rangle$. A major improvement of this policy is the ability to specify all data of a certain kind without enumerating all tags which meets the requirements of §4. For this we use e.g. $\langle medical, *\rangle$ to represent the medical data of all principals and $\langle *, bob\rangle$ to represent all Bob's data of any kind.

To achieve this, for any concern c and specifier s we establish the following subtyping relation:

$$\begin{tikzcd}
 & \langle *, * \rangle & \\
\langle c, * \rangle \arrow[ur, "\preceq"] & & \langle *, s \rangle \arrow[ul, "\preceq"'] \\
 & \langle c, s \rangle \arrow[ul, "\preceq"] \arrow[ur, "\preceq"'] &
\end{tikzcd}$$

That is, a tag $t = \langle c, s\rangle$ is a subtype of $t' = \langle c, *\rangle$ and $t'' = \langle *, s\rangle$ which are themselves subtypes of $t''' = \langle *, *\rangle$. For instance, $\langle medical, bob\rangle$ (Bob's medical data) is a subtype of both $\langle medical, *\rangle$ (medical data) and $\langle *, bob\rangle$ (Bob's data), which are both subtypes of $\langle *, *\rangle$ (all data in the current naming context).

We now consider Definition 1 of FlowK IFC as a safe system. For each contributing sub-definition we elaborate on any extended tag matching required in FlowK2.

5.1 Enforcing Information Flow via Labels

The purpose of IFC is to prevent data leakage by controlling exchange of information.

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

$$A \to B, \textit{ iff} \begin{cases} S(A) \precsim S(B) \\ I(B) \precsim I(A) \end{cases} \tag{1}$$

where, in our original FlowK model the preorder $\precsim$ denoted mere inclusion $\subseteq$.

In rule (1), the subrule concerning secrecy labels ensures that an entity only passes information to an entity that is allowed to receive it, thus enforcing the "no read up, no write down" policy of the Bell-LaPadula model [3]. The subrule concerning integrity labels enforces quality of data during reading down and writing up, as proposed by Biba [5]. It is therefore possible to represent traditional security requirements as IFC constraints, although we use labels for more general security contexts.

To extend rule (1) for 2D tags for the flow $A \to B$, we need only redefine the $\precsim$ binary relation between sets of tags X and Y as follows:

$$X \precsim Y \text{ iff } \forall t \in X \; \exists t' \in Y : t \preceq t' \tag{2}$$

Together with rule (1), this entails that a flow $A \to B$ is allowed if and only if for all secrecy tags of A there exists a supertype in the secrecy tags of B

Together with rule (1), this entails that a flow $A \to B$ is allowed if and only if for all secrecy tags of A there exists a supertype in the secrecy tags of B, and for all integrity tags of B there exists a supertype in the integrity tags of A. For example, an entity A labelled with $S(A) = \{\langle medical, bob\rangle, \langle legislation, EU\rangle\}$ is able to send data to an entity B with $S(B) = \{\langle medical, *\rangle, \langle legislation, EU\rangle\}$.

5.2 Creation of an 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 fork. We have the following rules for creation:

$$\text{if } A \Rightarrow B, \text{ then} \begin{cases} S(B) := S(A) \\ I(B) := I(A) \end{cases} \tag{3}$$

That is, the created entity inherits the labels of its creator.

5.3 Privileges for Managing Tags and Labels

Certain active entities (processes) have privileges that allow them to modify their labels. This is needed to support application management and for declassification as described below. An entity has two sets of privileges for removing tags from its secrecy and integrity labels (P_S^- for S and P_I^- for I), and two sets for adding tags to these labels (P_S^+ for S and P_I^+ for I). That is, for an entity A to remove the tag $t_s \in S(A)$, it is necessary that $t_s \in P_S^-(A)$, similarly to add the tag t_i to the label $I(A)$ it is necessary that $t_i \in P_I^+(A)$. That is, for any entity A, label $X(A)$ and tag t:

Definition 3. *A label change denoted $A \rightsquigarrow A'$ is safe if and only if all label updates respect:*

$$\begin{array}{lcl} X(A) & := & X(A) \cup \{t\} \textit{ if } t \in P_X^+(A) \quad \textit{or} \\ X(A) & := & X(A) \setminus \{t\} \textit{ if } t \in P_X^-(A) \end{array} \tag{4}$$

For example, in order to receive information from an entity B, an entity A will need to set its labels (if it has the privilege) such that the flow constraints expressed by the tags in B's labels are respected, that is, such that the flow $B \rightarrow A$ respects the subrules in rule (1).

In FlowK/2 (as in Flume or HiStar) only the process itself is able to change its secrecy and integrity labels and must request this explicitly. Indeed, it has been shown that implicit label changes can lead to covert channels [9], [37]. Our main difference compared with these systems is the separation of privilege over S and I.

In more detail, for 2D tags, rule (4) becomes:

$$\begin{array}{lcl} X(A) & := & X(A) \cup \{t\} \text{ if } \exists t' \in P_X^+(A) : t \preceq t' \quad \textit{or} \\ X(A) & := & X(A) \setminus \{t\} \text{ if } \exists t' \in P_X^-(A) : t \preceq t' \end{array} \tag{5}$$

We also add special privileges noted $\langle c, \Delta \rangle$, $\langle \Delta, s \rangle$ and $\langle \Delta, \Delta \rangle$ that allow removal only of the labels $\langle c, * \rangle$, $\langle *, s \rangle$ and $\langle *, * \rangle$ respectively. See "declassification" below for an important motivation for Δ.

We propose the following notation: for a process and its labels $(A, S, I) \rightsquigarrow (A, S', I')$ is the modification of the process labels following rule 4.

Declassification. An important example of changing security context is *declassification*. For example, (1) plain-text data may be secret and tagged accordingly whereas the same data when encrypted may flow more freely (2) the flow of personal medical data may be restricted whereas anonymised data derived from sets of medical records may be made available to medical researchers. Processes such as anonymisers and decrypters must be trusted to have the privilege to *declassify* the derived anonymised/decrypted data, i.e., to create data without the tags indicating secrecy/privacy.

For example, the encrypting process starts off with a tag such as $\langle government, restricted \rangle$ in S, reads data with tags $\langle government, restricted \rangle$ in its S, encrypts the data, changes its own security context by removing the $\langle government, restricted \rangle$ tag from S (for which it has the privilege) then writes the encrypted data with a tag such as $\langle government, encrypted \rangle$.

More generally, a process A, with the privilege $P_S^-(A) = \{\langle medical, \Delta \rangle\}$ and the label $S(A) = \{\langle medical, * \rangle, \langle medical, anonymised \rangle\}$ is able to declassify to $S(A) = \{\langle medical, anonymised \rangle\}$. The use of Δ privileges allows the trust placed in a certain entity to be precise and is particularly useful when specifying declassifier privileges. Without it, we would have had $P_S^-(A) = \{\langle medical, * \rangle\}$ and no guarantee that the process would not declassify to $S(A) = \emptyset$ (also removing $\langle medical, anonymised \rangle$).

Endorsement. An example concerning integrity labels is *endorsement*. Input data may need to be verified before it can safely be used, e.g., that input into a software library is from a trusted source, such as directly from RedHat. An endorser has an integrity label that allows it to input untrusted data, perhaps adding a tag to its verified output data to indicate that the data is endorsed.

Creation and Privileges. On creation, labels are automatically inherited by a created entity from its creator (rule 3), but privileges are not. If the child is to be given privileges over its labels, they must be passed explicitly. We denote the flow generated by an entity A giving selected privileges $t_X^{\pm}$ to an entity B as $A \overset{t_X^{\pm}}{\hookrightarrow} B$ (for example allowing t to be removed from S, would be denoted $A \overset{t_S^-}{\hookrightarrow} B$).

Definition 4. *A privilege delegation is safe if and only if:*

$$A \overset{t_X^{\pm}}{\hookrightarrow} B \textit{ only if } t \in P_X^{\pm}(A) \tag{6}$$

i.e. a process can only delegate a privilege it owns. Extending for 2D tags, rule (6), becomes:

$$A \overset{t_X^{\pm}}{\hookrightarrow} B \text{ only if } \exists t' \in P_x^{\pm}(A) : t \preceq t'$$

5.4 Conflict-of-Interest Groups

A policy maker may need to specify a separation of duty (SoD) or conflict-of-interest (CoI) between principals and/or roles [28]. An example of SoD is that an auditor may not audit their own actions. A CoI may arise when a principal could give professional advice to a number of competing companies.

CoI support in IFC is unique to FlowK/2. We define a set C of tags that represents some specified conflicting interests. In order for the configuration of an entity A to be valid with respect to C, rule (7) must be respected:

Definition 5. *A process B does not violate a CoI C if and only if:*

$$\left| \left(S(A) \cup I(A) \cup P_S^+(A) \cup P_I^+(A) \cup P_S^-(A) \cup P_I^-(A) \right) \cap C \right| \leq 1 \tag{7}$$

That is, an entity is non-conflicting in this context if the set of its potential tags (past, present and future) contains at most one element from the set of tags within the related CoI group. In detail, by potential tags we mean the tags in its current S and I labels and those tags that it has the privilege to add to $S(A)$ (i.e. $P_S^+(A)$) and to $I(A)$ (i.e. $P_I^+(A)$) or that it may have removed from $S(A)$ (i.e. $P_S^-(A)$) and from $I(A)$ (i.e. $P_I^-(A)$). CoI rules should be checked every time a privilege is granted.

Taking a simple example first, suppose a conflict $C = \{\langle car, fiat \rangle, \langle car, ford \rangle, \langle car, audi \rangle, ...\}$ and some data is labelled $FiatData[S = \{\langle car, fiat \rangle\}, I = \emptyset]$ and $FordData[S = \{\langle car, ford \rangle\}, I = \emptyset]$. The CoI described ensures that it is not possible for a single entity (e.g. a process) to have access to both $FordData$ and $FiatData$ either simultaneously or sequentially, i.e. enforcing that

FordData and *FiatData* are processed separately.

For 2D tags there are three types of policy we must express: constraints applied to whole tags, to concerns and to specifiers. We define, accordingly, three operations on a tag's pair, the identity function id and the two projections π_1 in $\mathscr{C}$, π_2 in $\mathscr{S}$.

$$\begin{array}{rclcrcl} \pi_1 : \mathscr{C} \times \mathscr{S} & \rightarrow & \mathscr{C} & \quad & \pi_2 : \mathscr{C} \times \mathscr{S} & \rightarrow & \mathscr{S} \\ \pi_1(\langle c, s \rangle) & = & c & & \pi_2(\langle c, s \rangle) & = & s \end{array} \tag{8}$$

We extend these operations to sets, such that:

$$\begin{array}{rclcrcl} \pi_1 & : & \wp(\mathscr{C} \times \mathscr{S}) \rightarrow \wp(\mathscr{C}) & \quad & \pi_2 & : & \wp(\mathscr{C} \times \mathscr{S}) \rightarrow \wp(\mathscr{S}) \\ \pi_1(T) & = & \{\pi_1(t) \mid t \in T\} & & \pi_2(T) & = & \{\pi_2(t) \mid t \in T\} \\ & = & \{c \mid \langle c, s \rangle \in T\} & & & = & \{s \mid \langle c, s \rangle \in T\} \end{array} \tag{9}$$

For an entity A we note the union of its labels and privileges:

$$SU(A) = S(A) \cup I(A) \cup P_S^+(A) \cup P_S^-(A) \cup P_I^+(A) \cup P_I^-(A) \tag{10}$$

A conflict of interest is denoted $P_{CoI}(f, G)$ where f is π_1, π_2 or id and G is a set of conflicting tags. Rule 7 can be expressed as (where C is a CoI group):

$$P_{CoI}(f, C), |f(SU(A)) \cap C| \leq 1 \tag{11}$$

Intersection and cardinality interfer with subtyping as follows:

- $\{a, b, c\} \cap \{*\} = \{a, b, c\}$;
- $\{\langle a, b \rangle, \langle a, d \rangle, \langle c, d \rangle\} \cap \{\langle a, * \rangle\} = \{\langle a, b \rangle, \langle a, d \rangle\}$;
- $|\{*\}| = \infty$, $|\{a, *\}| = \infty$ and $|\{*, a\}| = \infty$.

As an example, the conflict of interest rule $P_{CoI}(\pi_1, \{medical, private\})$ means an entity can handle the concern medical or private but not both. $P_{CoI}(id, \{\langle private, * \rangle\})$ means an entity can only ever manipulate the private data of a single user.

For the car data example, a issue with previous label models with explicitly enumerated tags is that if a new company was to use the application, a new tag would need to be added to the CoI group. This could prove problematic for a more rapidly changing set.

Using FlowK2, we have processes labeled $[S = \{\langle car, ford \rangle\}, I = \emptyset]$, $[S = \{\langle car, fiat \rangle\}, I = \emptyset]$ etc. and the CoI policy is expressed as: $P_{CoI}(id, \{\langle car, * \rangle\})$. This is simple to read and understand (i.e. a process can manipulate information for only one specifier of concern car) and this policy will not change over time.

6 Kernel Implementation

In §5 we discussed the constraints that must be enforced by FlowK/2 in order to ensure the secrecy and integrity of information within our system. Here we describe how FlowK is implemented, mentioning the differences for tag storage and matching for FlowK2.

In a Linux-like OS, the entities defined in the model are processes, files, pipes and sockets. Information flows $A \rightarrow B$ are usually generated through system calls. If we assume that there is no shared memory between processes, for the four types of entity (process, pipe, socket and file) the only possible information flow is through system calls.

Process, pipe and socket labels are stored in kernel memory in FlowK and follow the lifecycle of their associated entities. File labels are made persistent and stored on disk alongside the file as part of its metadata (using extended attributes in a fashion similar to SELinux).

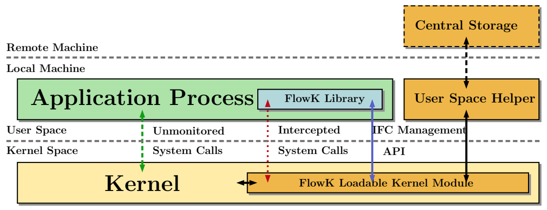

Fig. 1: Kernel Module Architecture Overview.

Privileges as defined in §5.3 are only associated with active entities, i.e. processes. Certain processes have privileges allowing them to change their labels, meaning they are able to change their security context $A \rightsquigarrow A'$. As discussed in §5, label change is an explicit action taken by the entity itself, therefore *passive entities (files, pipes and sockets) have immutable labels and no privileges*.

Files are the only persistent entity; their labels are made persistent through file system *extended attributes*. Any file stored on a file system that does not support extended attributes will be considered unlabelled, and IFC constraints enforced accordingly. How applications can make their labels and privileges persistent across several executions is discussed in the next section.

All labelled entities are allocated their labels when they are created. For a process A creating some entity E the subrules associated with the flow $A \Rightarrow E$ in rule (3) are respected, that is, E inherits A's labels. See §8 for how this could be used in a PaaS-hosted web-based service.

There are several options for providing IFC, through: 1) system call interception [14], [22]; 2) a Linux Security Module (LSM) [14]; 3) a user-space library [19].

FlowK uses system call interception through a Linux Kernel Module [10]. Although this approach has some limitations [12], [34], we selected it for a proof-of-concept exploration of IFC provision. At present, building a dedicated LSM for IFC enforcement has some composition challenges with SELinux [25]. If SELinux is not required, or if current work leads to compositionality of SELinux and IFC, this would become the solution of choice. Little change would be required to FlowK since only the system call interception mechanism would vary. User space interposition, while offering more portability, does not provide as strong security guarantees [24].

6.1 Enforcing IFC by System Call Interposition

In FlowK, we provide IFC to the OS through a kernel module that intercepts system calls [10] to enforce IFC constraints, while relying on the underlying system calls, as shown in Fig. 1. We assume that exchange of information between active entities (processes) occurs through passive entities (files, sockets and pipes) and we prevent shared memory between processes. We present a short overview of our implementation, for more detail see [22].

Our kernel module maintains a map between entity identifiers (processes, pipes etc.) and their respective labels and privileges. In system calls such as write or send, the security context of the process and of the entity associated with the provided file descriptor (i.e. socket, pipe or file) are retrieved, evaluated and compared to decide whether the call is authorised (and respectively for read and recv). System calls not generating information flow are left untouched.

Tags are represented in FlowK as 64 bit integers and labels are a sorted list of tags. In FlowK2 a 64-bit integer is used for each component of the tag, $\langle concern, specifier \rangle$. The overhead on system calls consists of the interception overhead plus the enforcement of IFC constraints. On read or write for example, Definition 2 is enforced.

6.2 User-Space Helpers (Ushers)

User-Space Helper processes (Ushers)have three functions: 1) persisting an application's security context across multiple executions; 2) saving the correspondence between local (FlowK) and global tag representations; 3) assisting in inter-process message-passing.

Security Context Persistence: Recall that a process's security context comprises the tag sets in its S and I labels and its privileges over S and I tags (§5). In order to build an effective system it is necessary for such contexts to persist over several executions of the same application/principal. Krohn et al. introduced such a scheme in [14]. A context persistence helper, when invoked, allows a process to save its security context.

On a flowk_save_context call, the kernel module invokes an Usher and transmits to this helper the current security context of the caller. The Usher saves this context in an append-only database where this (immutable) context is associated with a unique security context token. When an application starts and wants to retrieve a particular security context, it calls flowk_restore_context with this token as parameter. FlowK uses this to retrieve the corresponding context and set it up for the caller, provided that the resulting context does not violate CoI constraints.

For our proof-of-concept implementation we assume this context database is associated with the WebWorkers Application Framework. In future work, a context database network will be required within and between clouds, accessible by the Ushers. These helpers may in practice act as local context caches (which is reliable, since a saved context is immutable).

Global to Local Name Mapping: A second function of the Usher is to translate local names to global names. The global name for each tag is a unique flat string, whereas in kernel space, tags are represented as a locally unique 64 bit identifier. The Usher maps between each tag's string and 64 bit representations.

When a new tag is created (by a privileged application-management process) the Usher checks that the string is unique before associating it with a local 64-bit ID. A similar scheme, described in more detail, was introduced by Zeldovich et al. [38]. With the integration of an IFC-enabled middleware with FlowK, we have the capability to create distributed systems with intra-cloud, inter-cloud and end-system-cloud communication. Such a system would require a global naming specification, beyond the scope of the current work. One can envisage that application names are concatenated with application-specific and user-instance-specific names to create application domains, for example. For this reason, we are not yet incorporating FlowK2's naming scheme into our standard FlowK although we have implemented and tested the IFC matching algorithm.

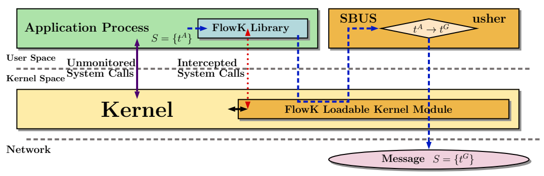

Fig. 2: FlowK architecture, integrating messaging middleware.

Within some application domain, application instances can be spawned within a certain context by a trusted managing process through the FlowK system call flowk_execute_in_context which executes a new application instance in the context specified by a provided context token. If an application has a secret context token, it can restore a previously saved security context. If an application does not have such a token, it can create a context and save it.

Inter-Process Message Passing: Inter-machine communication generally occurs through some form of socket. However, in IFC systems, sockets connected to remote machines are generally considered public and do not carry labels. Zeldovich et al. [38], in DStar, were the first to introduce an inter-machine message passing scheme for distributed IFC systems. Our current project integrates a message-passing middleware layer using a similar approach and providing additional features. A trusted Usher process is attached by the kernel module to each application process that makes a library call to use the external messaging functionality.

For a tag t we note the machine-local representation of this tag (on machine A) as t^A and its global representation t^G. Fig. 2 shows a process p on machine A sending a message to a process q (such that $t^G \in S(q)$) on machine B. For simplicity we assume process p's security context comprises only tag t, $S(p) = \{t\}$. In outline, the messaging helper: 1) receives the message from its attached process; 2) translates the tag from its local representation to its global representation (see §6.2); 3) attaches the label to the message and sends it over the network. 4) Once the message is received the label is translated to its local representation t^B (assuming this already exists for t^G, otherwise is is created) and the message sent to the receiving process. For details of IFC enforcement during send and receive, and any possible security context changes required by processes, see §9.

Our middleware SBUS [32], [33] has been extended with IFC. SBUS-IFC expects structured messages and provides attribute-level labelling. Each attribute a can have its own labels provided that, for message M, $S(M) \precsim S(a)$ and $I(a) \precsim I(M)$. The attributes that cannot flow to a particular process are nullified, e.g. allowing fine-grained medical record policies [15].

7 Support for Audit

Our kernel module is where OS flows are checked and enforced and is therefore the natural point at which audit logs can be generated. Audit logs can be used for several purposes. Firstly, they can be used by the cloud provider to demonstrate compliance with tenant/ user Service Agreements, particularly when third parties may be involved. Allocation of responsibility when a cloud service provider makes use of external services is problematic [16]. Secondly, to check that application-level policy has been correctly specified and enforced. Thirdly, to investigate claims of data leaks by cloud tenants or end-users of tenants' cloud-hosted services. Fourthly, to guard against and investigate general cyber-attacks on the system. Each FlowK component creates an audit log. Our future work will include developing tools to query these logs as outlined below.

FlowK enforces IFC constraints as specified by policy, but this may not reflect the intention of the policy maker, or a misconfiguration may have been made. An audit tool is needed to help determine whether there are circumstances in which information may potentially have flowed from one security context to another. Conversely, if a claim is made that a leak has occurred, an audit tool may be able to determine that such a flow did not happen (at this level of the system).

Any monitored system call issued by a labelled process is logged. Operations on labels are also recorded. Operations from unlabelled processes are only recorded if they represent an interaction with a labelled entity. We have defined several different types of flow as part of our IFC model (see §5), namely data flow, creation flow, security context change and privilege delegation. These different types of flow may be important in security forensics and are recorded as part of the audit log.

A log entry is as follows:

Elements	Information
Outgoing Flow Entity	identifier, secrecy label, integrity label
Incoming Flow Entity	identifier, secrecy label, integrity label
Flow Operation	timestamp, system call name, permitted?

This recorded information allows us to understand better the behaviour of the system. For example, the log can be searched to find whether a suspicious number of forbidden flows is recorded for some entity. If a leak is claimed or suspected, the logs record all flows involving labelled entities that have taken place.

To investigate information leaks, we propose an audit tool that processes the logs in response to an authorised user's query. The information contained in the log allows a directed graph of permitted flows to be built. Nodes are composed of **[entity identifier, security context]** and the edges are composed of **[system call name, timestamp]**. This is illustrated in Fig. 3.

All public entities are represented by a single node representing the public domain, as our system is not able to provide guarantees on information flow between unlabelled entities.

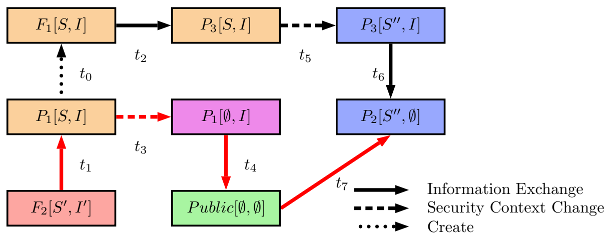

Fig. 3: Audit directed graph (the disclosure path is shown in red between the contexts S', I' and $S'', \emptyset$).

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 important 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 strictly monotonically increasing timestamps from the first node to the last. Fig. 3, shows in red 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. If the data disclosure was unintentional, it could be a sign that a laundry attack has occurred (the attacker gained control of P_1's declassification mechanism through, for example, a vulnerability in the software). All P_1's declassifications should then be checked, and the application in general.

An example of the use of IFC audit is that adherence to a "cease and desist" order on some principal such as a cloud-hosted service provider is hard to demonstrate in current systems. Similarly, a user may request that their data is deleted. In our system, every principal has a unique label and demonstrating compliance with this rule is simple: it must be shown that there is no edge with a timestamp greater than the time of the "cease and desist" enforcement that connects the principal's nodes to other nodes.

The issue of derived data (metadata) is problematic in current systems. A cloud-hosted application or cloud service provider may gather data on its users and sell it to advertisers. This could even continue for users who have asked to be deleted from the system. When IFC is used, advertising data could only be derived explicitly through some form of declassifier. This would have to be part of the service-providers' policies and its contract

S Label	Atomic tag set	FlowK2 tag set
$S_{topsecret}$	{protected, secret, topsecret}	$\{\langle govt, protected\rangle, \langle govt, secret\rangle, \langle govt, topsecret\rangle\}$
S_{secret}	{protected, secret}	$\{\langle govt, protected\rangle, \langle govt, secret\rangle\}$
$S_{protected}$	{protected}	$\{\langle govt, protected\rangle\}$
$S_{unclassified}$	$\emptyset$	$\emptyset$

TABLE 1: Atomic and FlowK2 tag sets corresponding to the "Government Protective Marking Scheme".

with users. Compliance with the users' wishes could more easily be tracked and enforced.

Allocation of responsibility for data is also highlighted by IFC audit. The cloud provider might demonstrate that application policy was enforced as defined, and that if a leak occurred it was through a legitimate channel (such as a declassifier). That is, the data leakage is the responsibility of the tenant-provided software and not the cloud provider. The tenant's software may have been penetrated, did not correctly respect declassifying requirements, or tenant-provided policies did not provide the intended effects for their end-users. We believe that the possibility of investigation via an IFC audit system should allow better allocation of responsibility between the parties involved in cloud applications.

8 Application-Level Integration

In this section we describe application-level integration with FlowK. First, persistent storage is needed. We have already seen how a file service interface is provided by FlowK. Here we discuss briefly how databases, key-value stores etc. can be made IFC-compliant. We then outline how we have modified a standard webworkers framework for running web applications to make it IFC-aware; details are in [22].

8.1 Integrating with Persistent Storage

A first technique to provide IFC with data stores comes directly from work on library-provided IFC [4], [7], [8], [22]. Here, the tags are stored within the database alongside the data and a trusted software component ensures that when information is read from the database, the corresponding labels are applied. In FlowK [22] we propose a similar system where we interface with a key-value store through a trusted FUSE interface (see Fig. 4). In Flume [14], a trusted application provides the interface between untrusted applications and the database.

More recent work has seen the emergence of databases that natively understand IFC concepts and can enforce IFC policies [29]. Our SBUS-IFC work allows the integration of messaging middleware with database queries which, coupled with IFC-aware databases, could provide a clean implementation with a minimum TCB. With the recent resurgence of interest in IFC at the OS and application level, it is likely that new and better solutions will emerge in the near future.

8.2 A Cloud-Deployed Web Service Framework

In [22] we presented the design of a framework for web applications to run above FlowK, in order to demonstrate application-level integration with FlowK, see Fig. 4. Cloud applications are run by IFC-constrained workers. End-user clients send their requests to a reverse proxy, the Security Context Router (SCR). The request may contain a security context token that is used by the reverse proxy to 1) instantiate a new worker if needed; 2) route the request to the appropriate worker.

An implication of the design is that "untrusted" applications can be supported on an IFC-enabled cloud. To achieve this we allow end users to create new, persistent security contexts (see §6.2). When sending a request, the user may ask for the token associated with a context it owns. The request will be executed within the persistent context associated with the token.

In Fig. 4, we see that the SCR replaces the load balancer present in a more traditional architecture (although, we could envisage a load balancer upstream to several SCRs). Each worker is constrained to work within a well defined security context as specified by the end-user request. The SCR is implemented as an Apache httpd server used as a reverse proxy running a custom-made module. The SCR runs as a trusted process and thus can use the flowk_execute_in_context FlowK system call to spawn a new worker for a user-specified security context.

Following the IFC definitions given in §5 and assuming that the cloud provider guarantees that no declassification occurs without user consent, we ensure that the data sent through a server request associated with a security context are bound by this security context. The end-user is thus able to specify data-bound policy that will be enforced by the cloud provider, regardless of the application implementation.

9 Middleware Integration

SBUS [32], [33] is a messaging middleware in which data, events, etc., are encapsulated within messages. IFC in a cloud-integrated middleware is potentially concerned with flows intra-cloud, inter-cloud and between clouds and users' end-systems. The support in SBUS-IFC for IFC within messages is fine-grained, in that individual attributes are labelled. Below, we outline how flows are authorised according to the labels of the messages and the processes between which they flow.

We have seen how IFC labels are allocated to processes as part of their initialisation. In addition, any process that requires external communication will, implicitly, make a library call (that includes a system call to FlowK) which in turn instantiates its SBUS Usher, see §6.2 and Fig. 2.

9.1 A System-Wide IFC Model

To demonstrate a system-wide IFC labelling scheme with corresponding flow rules we chose the hierarchical scheme unclassified, protected, secret, top-secret used by the UK government until April 2014: "Government Protective Marking Scheme".[1] The label model was chosen for its wide applicability to many Governmental and other large organisations. Such schemes can be captured in

1. The current version is available via: https://www.gov.uk/government/publications/government-security-classifications

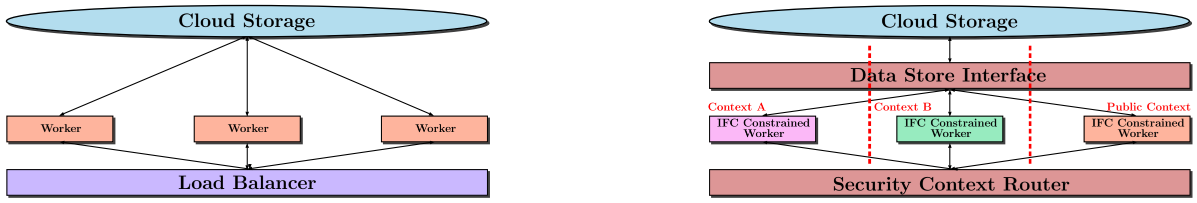

Fig. 4: A scalable webserver worker process design running multiple identical instances of an application. We reduce the attack surface by running the multiple instances in different security contexts.

SELinux hierarchies. Equally, we could have defined a label set for healthcare and lifestyle or environmental monitoring with associated flow rules, as in §5. For simplicity of exposition we consider only the secrecy labels S and assume the integrity labels I are null.

We define tags protected, secret, topsecret. In order to communicate, processes must hold a label with appropriate tags from the above set to describe their current runtime privilege level. Translating this hierarchical IFC security classification into the (D)IFC label models defined in §5, process labelling for atomic tags and FlowK2 2D tag model are illustrated in table 1.

The rules of §5 then enforce the *no read up, no write down* policy described in §3 and §5. Note that if we wanted to define a scheme with an arbitrarily tall hierarchy with new layers added over time, the topmost layer, equivalent to top-secret in this example, could be expressed using the 2D tag model of FlowK2 as: $S = \{\langle govt, *\rangle\}$

The assignment of labels within messages is defined below, followed by the rules for IFC enforcement when messages are sent between components.

9.2 IFC in SBUS Messages

The role of IFC in middleware is to manage the flow of messages, and the data encapsulated within. SBUS messages are strongly typed, comprising a set of attributes. In a message, an attribute comprises a *name*, *primitive-type*, and *value*. Each attribute is assigned an IFC label S, as defined above (see Fig. 5 for example message instances). This assignment is determined by:

Static definition: The attribute label can be defined statically within the message type schema. This sets the attribute's IFC label for all message instances of the type,[2] i.e. applications cannot change this.

Applications: The application producing a message can set the security labels for the attributes. If the application does not assign a label to an attribute, the middleware sets this label to the component's current label, subject to any static definition.

9.3 IFC Enforcement

IFC enforcement requires the evaluation of the message against the receiving component's S label, in accordance with the *no read up, no write down* policy. That is, for:

2. It is encoded as a first-class definition within the message type.

IFC Enforcement on Receive: If the receiving component's S label is below that required to read an attribute, the attribute value is removed from (made null in) the message. This is enforced by the SBUS Usher when a component receives a message, before it is delivered to the application.

IFC Enforcement on Send: A sending component cannot emit a value for an attribute where the attribute's S label is at a lower level than the S of the component. This is enforced by the SBUS Usher when an application attempts to send a message, by removing the values for any attributes violating this policy, before message propagation.

This enforcement involves the SBUS Usher inspecting all attributes of a message, testing for compliance, on receiving and sending.

As described in §5, there are situations where a process should be able to *write down*, after **declassification**. This is possible if the process has the privilege to remove the tag(s) from its S label, and changes its security context correspondingly, before sending the message.

Components cannot in general see the original labels for the messages they receive. This prevents side-channels, in that though they are privileged to see the content, they need not know (and thus be able to infer) specifics about the sending process. However, this prevents more complex networks from being built, e.g. those with brokers making routing decisions in the application layer, since the labels are effectively lost between hops.

Therefore, we define an additional label-visibility privilege that components can hold, which enables a component to see the labels for the messages it receives. We use this to facilitate functionality as in content-based routers. Such components would probably be part of managed infrastructure (e.g. in a Government-wide service), perhaps operating on behalf of other "sensitive" (topsecret) components.

9.4 SBUS-IFC Experiments

We used SBUS-IFC in a system monitoring and administration scenario: (1) To demonstrate the functionality of an integrated IFC-middleware. (2) To indicate the performance overheads that IFC functionality imposes. A summary of the overhead imposed on message processing during send and receive are described in §10.3. An example of the transmission of syslog messages is given in Fig. 5 showing attributes being removed after security context checking.

Message sent by syslog process running at secret level, where all fields except for process and content are declassified to level protected

```
<syslog>
  <priority lbl="P">4</priority>
  <timestamp lbl="P">
      23:17:59.00, 15/04/2014
  </timestamp>
  <host lbl="P">morena</host>
  <process lbl="PS">locd[69]</process>
  <content lbl="PS">
      SSLHandshake failed 192.168.1.43:22412
  </content>
</syslog>
```

Message received by remote process running at protected level

```
<syslog>
  <priority>4</priority>
  <timestamp>
      23:17:59.00, 15/04/2014
  </timestamp>
  <host>morena</host>
  <process />
  <content />
</syslog>
```

Fig. 5: Example of a message, receive-mediated by an SBUS Usher, for a process running at level protected.

System Calls	Native Linux	FlowK	Difference	FlowK Multiplier	Flume Multiplier
open (r/w)					
–create	1.9μs	15.8μs	13.9μs	8.3	16
–exists	1.4μs	15.4μs	14μs	11	34.5
–does not exist	5.1μs	18.9μs	13.8μs	3.7	23.6
close	0.8μs	0.9μs	0.1μs	1.1	1.3
write (file)	1.2μs	7.2μs	6μs	6	NA
read (file)	0.3μs	6.4μs	6.1μs	21.3	NA
fork	28.7μs	225.4μs	196.7μs	7.9	NA
pipe latency	4.4μs	7.8μs	3.4μs	1.8	8.2

TABLE 2: Some system call overheads compared with Flume as reported in [14]

10 Evaluation

The CloudSafetyNet project is at an early stage and work is in progress. Implementations tend to be for proof-of-concept and performance optimisation has not yet been done. This section aims to give an order of magnitude idea of performance compared with equivalent solutions. Performance evaluation of FlowK has been carried out on a quad-core 2.2Ghz Intel i7 with 6GiB of RAM running Fedora 20 (kernel version 3.14).

10.1 System Calls Interposition Overhead

We used a micro-benchmark to measure the performance of our system compared to standard Linux. The results are presented in Table 2. We also provide the performance of Flume as reported in [14] for comparison. In Flume, all data transfers are via an intermediate process, leading to multiple context switches. FlowK only uses an intermediate process (the SBUS Usher, see §6.2) for external message passing.

Another question is the influence of the label complexity over performance. In Fig. 6, we present the overhead on pipe latency against the number of tags composing a label. The performance overhead increases linearly with the label complexity. However, in practice it is unlikely that the most complex label will exceed ten or so tags.

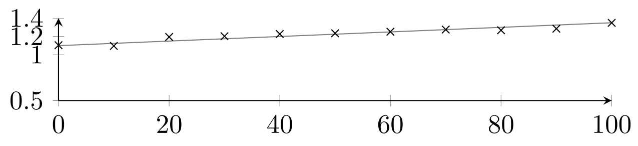

Fig. 6: Pipe performance multiplier (for unmonitored and 0–100 tags, normalised over native (y=1.0) performance)

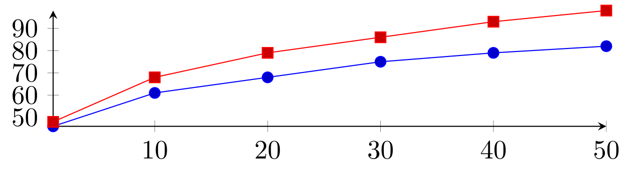

Fig. 7: Performance of our worker architecture (§8) with (red/square) and without (blue/circle) FlowK running. Y-axis: latency in ms for 90th percentile, X-axis: number of concurrent requests. Results are averaged over 10,000 requests.

10.2 Web application interface

We created a situation similar to the one presented in [21]. An application allows patients to retrieve their medical records from a database on request. They are then placed in a temporary data store. We have 50 records of around 9KB in a key-value store from which records are selected at random. We measure the latency in ms as a function of the number of concurrent requests, see Fig. 7. A single security context is used for all requests in order to allow a comparison against a system with FlowK switched off. Our aim is to outline the interposition cost induced by FlowK.

The overhead measured from 1 to 50 concurrent requests varies from 5% to 23%. The overheads measured are of the order of magnitude expected for similar OS-level IFC implementations [11], [14].

10.3 SBUS-IFC Evaluation

To indicate the overheads that IFC imposes, we compared the SBUS-IFC implementation with the non-IFC (standard) SBUS implementation. Using a workload of 5000 syslog messages, 20% of which were randomly subjected to attribute declassification (in line with that of Fig. 5), over 20 trials, we measured (refer to Fig. 8):
(1) The standard SBUS implementation with no IFC capability on send or receive, resulting in both read and write data leakage.
(2) SBUS-IFC message processing on **receive**, where some attribute values of incoming messages are not delivered to the receiver.
(3) SBUS-IFC message processing on **send**, where certain attribute values are nulled before sending because the sender has insufficient privileges to send them.
Note that though IFC is enforced both on message sending and receipt, for (2) all messages sent were authorised

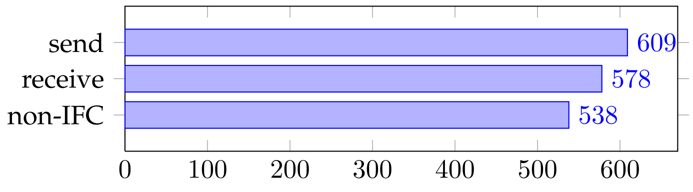

Fig. 8: Performance Comparison between SBUS and SBUS-IFC (x-axis time in ms)

by IFC policy (the messages were not perturbed on sending), and for (3) all messages received were authorised (thus not perturbed on receipt).

In summary, the results indicate that IFC enforcement introduces ∼**13% overhead in performance time** for the workload over standard SBUS. For the three overhead scenarios, the **workload transmission in bytes** for the receive IFC (where all message content is transmitted) was approximately 1.91MB, and for non-IFC was approximately 1.84MB for all trials. This means that attaching IFC labels to message attributes introduced 3.6% extra traffic. The write scenario prevents certain attribute values from propagating—thus transmission in that scenario was ∼1.52MB (a reduction in traffic of ∼17% over the non-IFC scenario for this workload).

11 Conclusions and Future Work

Our CloudSafetyNet project is investigating the potential and feasibility of providing IFC as part of cloud computing. If massive reengineering by cloud providers and/or cloud tenants was necessary for IFC to be incorporated into cloud services, it would be unlikely to be adopted. Therefore, our design choices aim to minimise any changes to existing systems when IFC is deployed, while adopting current best-practice where possible.

As proof-of-concept of the feasibility of IFC deployment end-to-end, we have designed, implemented and evaluated (1) an OS kernel module (FlowK) to enforce IFC, (2) a standard, application-level web services architecture adapted to use IFC via FlowK, and (3) an IFC-enabled messaging middleware (SBUS-IFC) integrated with FlowK, giving the potential to create IFC-aware distributed systems comprising clouds, multi-clouds and end-systems. Our middleware is capable of integration with other security systems including SELinux, giving the potential for heterogeneous IFC-enabled systems.

FlowK is solely concerned with IFC enforcement, system calls are unchanged, and unmonitored processes are not affected by its existence, apart from a small performance overhead. The limitations of providing IFC at the OS level in the cloud must be made explicit. We cannot guarantee that users' data could not be leaked through techniques based on VM co-residency [26], [39]. Any research in this area, would strengthen our solution.

FlowK has privileged User-Space Helper processes (Ushers) to assist in starting up applications and application instances from saved security contexts. A dedicated SBUS Usher is created for every process that participates in external communication to mediate sent and received messages according to the processes' security contexts.

Since public cloud services are provided by relatively few large companies, and applications are myriad, we believe the former are more likely to be trusted. Incorporating IFC into public cloud services would make them more trustworthy; we suggested a mechanism whereby cloud-hosted web-application end-users could establish security contexts in which to run on public clouds.

We have carried out preliminary work on application policy expression. We proposed FlowK2 with two-component tags, to allow more precise expression of required policy and trust, especially for "big data" processing. The space and time overhead of 2D tags over atomic tags is negligible for up to a dozen or so tags and in our experience to date, the need for more is unlikely.

Further work is needed on global tag naming before we finalise our tag design. To extend IFC to whole clouds, multi-clouds and distributed systems in general, many issues have to be considered in detail. There is a wealth of law and regulation regarding cloud service provision [16] that would need to be incorporated into any naming scheme, for example to include jurisdiction and geographical location.

We believe that IFC provides an important means to assist cloud service providers to demonstrate compliance with regulations on cloud computing, thereby increasing the trust of potential cloud users. CloudSafetyNet has carried out some initial work to demonstrate that IFC provision is feasible within clouds, between clouds and between clouds and end-user systems.

References

[1] Bacon, J., et al. Big Ideas paper: Enforcing end-to-end application security in the cloud. In *ACM/IFIP/Usenix Middleware* (2010), pp. 293–312.

[2] Bacon, J., Moody, K., and Yao, W. A Model of OASIS Role-based Access Control and its Support for Active Security. *ACM TISSEC* (2002), 492–540.

[3] Bell, D. E., and LaPadula, L. J. Secure Computer Systems: Mathematical Foundations and Model. Tech. rep., The MITRE Corp., Bedford MA, 1973.

[4] Bello, L., and Russo, A. Towards a taint mode for cloud computing web applications. In *PLAS '12* (2012), ACM, pp. 1–12.

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

[6] Cheng, W., et al. Abstractions for Usable Information Flow Control in Aeolus. In *Proc. USENIX Annual Technical Conference* (2012).

[7] Conti, J., and Russo, A. A Taint Mode for Python via a Library. In *Information Security Technology for Applications* (2012), Springer, pp. 210–222.

[8] Davis, B., and Chen, H. DBTaint: cross-application information flow tracking via databases. In *Proc. 2010 USENIX conference on Web application development* (2010), pp. 12–12.

[9] Denning, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[10] Dhanjani, N., and Rodriguez-Rivera, G. Kernel Korner: Loadable Kernel Module Programming and System Call Interception. *Linux Journal* (2001).

[11] Efstathopoulos, P., and Kohler, E. Manageable fine-grained information flow. In *ACM EuroSys* (2008).

[12] Garfinkel, T. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS* (2003), vol. 3, pp. 163–176.

[13] JAYARAM, K. R., ET AL. Trustworthy Geographically Fenced Hybrid Clouds. In *ACM/IFIP/Usenix Middleware* (2014), ACM.

[14] KROHN, M., ET AL. Information Flow Control for Standard OS Abstractions. In *21st ACM SOSP* (2007), pp. 321–334.

[15] LI, M., YU, S., ZHENG, Y., REN, K., AND LOU, W. Scalable and secure sharing of personal health records in cloud computing using attribute-based encryption. *Parallel and Distributed Systems, IEEE Transactions on 24*, 1 (2013), 131–143.

[16] MILLARD, C. J. *Cloud Computing Law*. OUP, 2013.

[17] MYERS, A. C. JFlow: practical mostly-static information flow control. In *Proc. 26th ACM SIGPLAN-SIGACT POPL'99* (1999), ACM, pp. 228–241.

[18] MYERS, A. C., AND LISKOV, B. A decentralized model for information flow control. In *Proc. 17th SOSP* (1997), ACM, pp. 129–142.

[19] NIU, B., AND TAN, G. Efficient user-space information flow control. In *Proc. 8th ACM SIGSAC* (2013), pp. 131–142.

[20] PAPAGIANNIS, I., MIGLIAVACCA, M., AND PIETZUCH, P. PHP Aspis: Using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development* (2011), p. 13.

[21] PASQUIER, T., SHAND, B., AND BACON, J. Information Flow Control for a Medical Web Portal. In *e-Society 2013* (March 2013), IADIS.

[22] PASQUIER, T. F. J.-M., BACON, J., AND EYERS, D. FlowK: Information Flow Control for the Cloud. In *submitted, CloudCom* (2014), IEEE.

[23] PASQUIER, T. F. J.-M., BACON, J., AND SHAND, B. FlowR: Aspect Oriented Programming for Information Flow Control in Ruby. In *13th International Conference on Modularity* (April 2014), ACM.

[24] PROVOS, N. Improving host security with system call policies. In *USENIX Security* (2003), vol. 3.

[25] QUARITSCH, M., AND WINKLER, T. Linux Security Modules Enhancements: Module Stacking Framework and TCP State Transition Hooks for State-Driven NIDS. *Secure Information and Communication 7* (2004).

[26] RISTENPART, T., ET AL. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. 16th ACM CCS* (2009), pp. 199–212.

[27] ROY, I., ET AL. Laminar: Practical fine-grained decentralized information flow control. *SIGPLAN Not. 44*, 6 (2009), 63–74.

[28] SANDHU, R. Separation of Duties in Computerized Information Systems. In *Database Security IV: Status and Prospects* (1990).

[29] SCHULTZ, D., AND LISKOV, B. IFDB: decentralized information flow control for databases. In *EuroSys '13* (2013), ACM, pp. 43–56.

[30] SFAXI, L., ABDELLATIF, T., ROBBANA, R., AND LAKHNECH, Y. Information flow control of component-based distributed systems. *Concurrency and Computation: Practice and Experience 25*, 2 (2013), 161–179.

[31] SIMONET, V. Flow Caml in a Nutshell. In *APPSEM-II* (2003).

[32] SINGH, J., AND BACON, J. SBUS: A generic, policy-enforcing middleware for open pervasive systems. *University of Cambridge Computer Laboratory Technical Report* (2014).

[33] SINGH, J., EYERS, D., AND BACON, J. Policy Enforcement within Emerging Distributed, Event-Based Systems. In *ACM Distributed Event-Based Systems* (2014).

[34] WATSON, R. N. Exploiting Concurrency Vulnerabilities in System Call Wrappers. *WOOT 7* (2007), 1–8.

[35] YIP, A., ET AL. Improving application security with data flow assertions. In *Proc. ACM 22nd SOSP'09* (2009), ACM, pp. 291–304.

[36] YOSHIHAMA, S., ET AL. Dynamic information flow control architecture for web applications. In *ESORICS 2007*. Springer, 2007, pp. 267–282.

[37] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proc. 7th USENIX OSDI '06* (2006), pp. 19–19.

[38] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIÈRES, D. Securing distributed systems with information flow control. In *Proc. 5th USENIX NSDI'08* (2008), pp. 293–308.

[39] ZHANG, Y., ET AL. Cross-VM side channels and their use to extract private keys. In *Proc. ACM Conference on Computer and Communications Security* (2012), CCS '12, ACM, pp. 305–316.

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

Thomas Pasquier is a PhD student and a Research Assistant at the University of Cambridge. His MPhil from Cambridge completed a project on Prevention of identity inference in de-identified medical records.

Jatinder Singh is a Senior Postdoctoral Research Associate at the University of Cambridge. His expertise is in policy-controlled middleware.

Olivier Hermant is a researcher at MINES ParisTech, France. His expertise is formal methods and their foundations. He is leading the development of Dedukti, a universal proof checker.

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