FlowK: Information Flow Control for the Cloud

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

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

***Abstract*—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. Our CloudSafetyNet project aims to show that Information Flow Control (IFC) can augment existing security mechanisms and provide continuous enforcement of extended. finer-grained application-level security policy in the cloud.**

We present FlowK, a loadable kernel module for Linux, as part of a proof of concept that IFC can be provided for cloud computing. Following the principle of policy-mechanism separation, IFC policy is assumed to be expressed at application level and FlowK provides mechanisms to enforce IFC policy at runtime. FlowK's design minimises the changes required to existing software when IFC is provided. To show how FlowK can be integrated with cloud software we have designed and evaluated a framework for deploying IFC-aware web applications, suitable for use in a PaaS cloud.

***Index Terms*—IFC, Kernel Module, Security, Integrity**

I. Introduction

A great deal of law and regulation relating to cloud computing has emerged [1], but the technical basis for enforcing and demonstrating compliance lags behind. Security concerns are seen as an obstacle to more widespread adoption of cloud computing. Traditional security practices such as authentication and access control are, of course, used by cloud-deployed applications. The CloudSafetyNet project is exploring the potential of Information Flow Control (IFC) to enhance these traditional approaches by providing continuous enforcement of policy at runtime. Moreover, policy can be finer-grained, capturing data properties as well as principal-based concerns. Here, we present the Operating System (OS) component of the proof-of-concept IFC system we are creating. Note that we use the term IFC to subsume Decentralised/Distributed DIFC (see §III).

Fig. 1 gives an overview of cloud service provision as Infrastructure/Platform/Software as a Service (IaaS, PaaS, SaaS). IFC could be provided independently of cloud-provider software, either as a language feature [2], [3] or through a library [4]–[6]. But this level of deployment assumes the cloud service is trustworthy throughout all the deployed cloud software (see Fig. 1). Even if the cloud provider is not malicious, data leakage could occur within the cloud implementation due to bugs, over-permissive data access policies or attacks of various kinds. We therefore investigated IFC enforcement within the OS (§V). To increase the likelihood of acceptance, our design requires minimum modification to higher software layers (§VI).

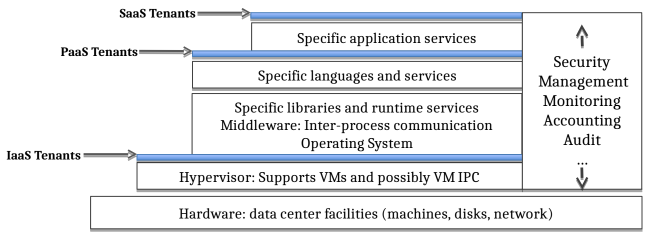

SaaS Tenants
PaaS Tenants
IaaS Tenants
Specific application services
Specific languages and services
Specific libraries and runtime services
Middleware: Inter-process communication
Operating System
Hypervisor: Supports VMs and possibly VM IPC
Hardware: data center facilities (machines, disks, network)
Security
Management
Monitoring
Accounting
Audit
...

Fig. 1: Cloud service provision architecture

Here, we 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 IFC rules on those flows as described in §IV. Our design is based on "policy-mechanism separation", in that the enforcement of IFC in FlowK is separated from any knowledge of principals, users and the management of privileges. This separation ensures maximum flexibility for higher levels of software; we show how FlowK supports application management in IV-E. Any application that does not use IFC can run unmodified and is only affected by a small performance hit on system calls, (see §VII).

This work contributes: (1) providing IFC within a widely used standard OS without requiring monitored processes to be modified, unlike other IFC systems (§III). (2) Support for conflict of interest in the IFC model to meet requirements for isolated processing (§IV-D). (3) As proof of concept, a minimally modified web server architecture for Ruby web applications, to take advantage of IFC (§VI).

We state our threat model in §II and related work in §III. §IV presents the FlowK IFC model. Implementation details are given in §V and §VI demonstrates how applications can be integrated with FlowK by describing our deployment of a webserver workers architecture, enhanced to support IFC policy expression and implementation via IFC. §VII discusses performance measurements. §VIII summarises this work in the context of CloudSafetyNet.

II. Threat and Trust Model

A. Trust Assumptions

We assume the *cloud provider* to be non-malicious and bound through legal requirements to do its best to protect its tenants' data. Indeed, it is in the best interest of large cloud providers (such as Google, Amazon, Microsoft) or governmental organisations (NHS UK) to guarantee high security standards. We believe a reasonable assumption is to rely on the cloud platform implementation to protect applications from unintentional data disclosure by isolating users or groups of users. For this purpose we propose IFC security contexts, see IV. 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 could leak data. In current infrastructure, a bug in the tenant's application could put all the data associated with this application at risk, i.e., via intra-tenant leakage. A misconfiguration in some provider's cloud services could allow applications to access data from other applications, i.e. inter-tenant leakage. IFC security context compartmentalisation allows the sharing of common resources and data, while reducing the risk surface of sensitive data.

B. Mitigated Threats

The 2014 Cost of Data Breach: Global Analysis [7] reports a cost of \$145 per record lost, and up to \$349 for medical records. The cost to companies depends on national legislation e.g., \$3.6M per company p.a. in the UK, \$4.2M in France, \$5.9M in the US. The study shows that European nations are willing to support preventative cost, while the US is heavily weighted towards post-leak litigation and reputation loss cost.

We cannot prevent all types of data leakage but we can mitigate many. The Web Hacking Incident Database [8] reports that in 2009, data leakage represented 28% of attacks. We cannot prevent bad application design (which accounts for the majority of vulnerabilities), but we restrict the scope of such attacks to a limited number of security contexts; e.g. common attacks such as SQL injection and OS command injection would be limited to a specific IFC context and would not allow leakage of all users' data. Furthermore, our general approach encourages strong authentication (11% of attacks are due to insufficient authentication) and our use of the Principle of Least Privilege (PoLP) means that individual application instances do not have access to the full application's dataset.

Therefore, depending on the security context defined by the application developer, the amount of data involuntarily disclosed through an application's bugs or vulnerabilities can be greatly reduced. A developer creating a separate security context per user would limit leakage to a single user. Our medical portal [9] would limit disclosure per user and per usage. In such circumstances, it becomes more complicated for an attacker to gain access to significant amounts of data. This reduces the risk of data being incorrectly disclosed, and would reduce the post-leakage cost.

In the web server architecture in §VI, 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.

Our aim is that applications with a relatively high requirement for data confidentiality and integrity, such as those handling medical records [9] could be safely cloud-hosted.

III. Related Work

A. IFC Models

It has long been argued that standard security techniques, such as firewalls and access control mechanisms, are not enough to prevent information leakage [10]. Indeed, it is beyond the scope of such mechanisms to determine whether, after the controls they impose, the information is used correctly. For example it is difficult to determine if the confidentiality of decrypted data is respected [11]. We therefore need to protect information flow end-to-end, i.e., when information is transmitted within and between applications.

In 1976, Denning [10] 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 [12] 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 the permitted *reading down* and *writing up*, as proposed by Biba [13]. Using this model we are able to control and monitor information flow to ensure data secrecy and integrity.

In 1997 Myers [14] 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 fluid system, able to capture the needs of different applications. Each entity is associated with two labels: a *secrecy* and an *integrity* label, to capture respectively the privacy/confidentiality of the data and the reliability of a source of data. These labels are composed of tags which represent 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. The FlowK model follows this general idea, see §IV.

B. IFC for Operating Systems

Some research projects have implemented IFC at the OS level, most notably Flume [15]. Here, a model similar to Myers' [14] is used and the entities are files, pipes, sockets and processes. In Flume, monitored (labelled) processes have

Description	Notation	Rule
Data Flow	$A \to B$	(1)
Creation Flow	$A \Rightarrow B$	(2)
Security Context Change	$A \leadsto A'$	(3)
Privilege Delegation	$A \overset{t^{\pm}_{X}}{\hookrightarrow} B$	(4) and (5)

TABLE I: Types of flow

access to a restricted set of system calls, and some (such as fork or 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 lifecycles.

Asbestos [16]is a rewritten OS rather than an imported module, requiring substantial changes in software that uses it. In Aeolus [17] and Laminar [18], 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.

C. IFC for Distributed and Cloud Systems

DStar [19] enables IFC in distributed systems by leveraging the labelling mechanisms of IFC-compliant OS such as Flume, Asbestos and HiStar [20]. DStar defines globally meaningful labels and each OS provides an *exporter* that maintains the correspondence between local representations and global labels, supporting inter-process communication as message-passing. CloudSafetyNet has made a general middleware IFC-compliant (SBUS-IFC) [21] and we are in the process of integrating it with FlowK. Current cloud platforms do not support IFC, although [22] discusses how IFC techniques could be provided as part of cloud services. The OS-level projects on providing distributed IFC, such as HiStar [20] or DStar [19] (which are based on Asbestos [16]) are closest to potential cloud deployment. The DStar system appears to be a suitable base on which to build an IFC-aware PaaS cloud, but we are not aware of such efforts.

IV. FlowK IFC Model

IFC augments authentication and authorisation by enforcing dynamically, that only permitted flows of information can take place. Flows are enforced by means of labels. We assume that labels are associated with entities after successful authorisation and here define how labels are used to control information flow. We introduce a notation for the different IFC operations, to describe succinctly the behaviour of an IFC-aware system. Table I summarises the rules controlling the flows that are defined in this section.

A. Enforcing Information Flow via Labels

A label is a set of tags. A tag represents a security concern for a category of data, e.g. encrypted, anonymised, clinical, personal, etc. The FlowK IFC model associates two labels with an entity: a *secrecy label* and an *integrity label*; $S(A)$ and $I(A)$ respectively for an entity A. The current state of these two labels (their sets of tags) is the *security context* of an entity. In the IFC world, *read* is equivalent to an incoming flow and *write* to an outgoing flow.

A flow of information from an entity A to an entity B, denoted $A \to B$, is allowed if the following rules are respected:

$$A \to B, \text{ iff } S(A) \subseteq S(B) \wedge I(B) \subseteq I(A) \tag{1}$$

The subrule of rule (1) concerning secrecy labels ensures that an entity only passes information to an entity that is allowed to receive it. The subrule concerning integrity labels further constrains the permitted flow to enforce data quality. Traditional security requirements can thus be enforced.

B. 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:

$$\text{if } A \Rightarrow B, \text{ then } S(B) := S(A) \text{ and } I(B) := I(A) \tag{2}$$

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

C. Privileges for Managing Tags and Labels

In our model certain entities (processes) have privileges as well as labels. These privileges represent the right for an entity to modify its labels. 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 $t_i \in P_I^+(A)$.

For an entity A, a label X (S or I) and a tag t, a change of the label is authorised if the following rule is respected:

$$\begin{array}{lcl} X(A) & := & X(A) \cup \{t\} \text{ if } t \in P_X^+(A) \quad \text{or} \\ X(A) & := & X(A) \setminus \{t\} \text{ if } t \in P_X^-(A) \end{array} \tag{3}$$

In order to receive information from an entity B, an entity A will need to set its labels (if it has the privilege) so that the flow constraints expressed by the tags associated with B are respected, that is such that the flow $B \to A$ respects rule (1). We do not allow implicit/automatic label change, to prevent a certain type of covert channel [18], [20] and to help programmers reason about their programs' behaviour.

A process changes its security context by changing its S and I labels. This may be done prior to creating an entity that will inherit that context (rule (2)). We denote (A, S, I) as a process and its labels, and on label change denote the flow $(A, S, I) \leadsto (A, S', I')$ or in short $A \leadsto A'$.

Declassification: An example of changing security context is *declassification*; e.g. plain-text data may have a *secret* tag whereas the same data when encrypted may flow more freely. A process that encrypts data must be trusted to have the privilege to *declassify* the derived encrypted data, that is, to create encrypted data without the *secret* tag in its S label.

In outline: the encrypting process starts off with the *secret* tag in S, reads the file with tag *secret* in S, encrypts the data, changes its security context by removing the *secret* tag from S (for which it has the privilege) then writes the encrypted data, e.g. as public data.

Creation: Note that on creation, labels are automatically inherited by a created entity from its creator, but privileges are not. If the child is to be given privileges over its labels, these must be passed separately. 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 $A \overset{t_S^{-}}{\hookrightarrow} B$). In order for a process to delegate a privilege to another process it must own this privilege itself. That is,

$$A \overset{t_X^{\pm}}{\hookrightarrow} B \text{ only if } t \in P_X^{\pm}(A) \tag{4}$$

As for other IFC systems, once an entity has been given a privilege, i.e., is trusted to perform some label manipulation, that privilege cannot be taken back. This is important for enforcing the separation of duties constraint described below.

D. Separation of Duty and Conflict-of-Interest Groups

A policy maker may need to specify a Separation of Duty (SoD) or Conflict of Interest (CoI) between principals [23], [24]. An example of SoD is that an auditor may not audit their own actions. A CoI may arise when a principal gives professional advice to a number of competing companies.

We define a set C of tags that represents some conflicting interests. For the configuration of an entity A to be valid with respect to C, rule (5) must be respected:

$$\left|\Big(S(A) \cup I(A) \cup P_S^{+}(A) \cup P_I^{+}(A) \cup P_S^{-}(A) \cup P_I^{-}(A)\Big) \cap C\right| \leq 1 \tag{5}$$

That is, an entity is non-conflicting in this context if the set of its potential tags contains at most one element from the set of tags within the related CoI group. 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. $P_I^{-}(A)$).

Suppose a conflict $C = \{Fiat, Ford, Audi, ...\}$ and some data (e.g. files) are labelled $FiatData[S = \{Fiat\}, I = \emptyset]$ and $FordData[S = \{Ford\}, I = \emptyset]$. The CoI C 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 Ford-owned data and Fiat-owned data are processed in isolation.

This rule is enforced on every $A \overset{t_X^{\pm}}{\hookrightarrow} B$ flow. Correctness depends on CoI groups being configured correctly before process labelling occurs. To our knowledge, CoI has not previously been enforced in IFC systems; e.g. while drawing on OS work for its design, [25] is purely language-based. We believe we can solve the problem posed in work on Chinese Wall policy [24] for the cloud [26], providing similar CoI isolation policy.

E. Application Startup—Trusted Entities

So far we have seen how active entities exchange information, create new entities, change their security context and delegate their privileges. We have not yet seen how tags and privileges are set up for the application instances.

We define a trusted entity to manage an application, denoted A^* for a trusted entity A. At startup, A^* has access to all the tags required for the application and all the privileges to manage them. It spawns all application instances, with appropriate security contexts, and sets up any CoI groups needed by the application; A^* is the only entity entitled to create CoI groups for the tags it manages.

In our current implementation such trusted processes are configured at startup with the tags for the application's naming domain, see §VI for an example. Work is in progress on integrating FlowK with an IFC-aware middleware [21], a function of which is to transfer tags as well as data.

In the example in §IV-D, the application manager $CarManager^*$ is set up with the privileges to add and subtract all the tags in the naming domains for S and I labels for the application. For simplicity, consider $I = \emptyset$ for this application. Suppose the naming domain for S comprises the tags $\{Fiat, Ford, Audi, ...\}$. $CarManager^*$ creates a conflict of interest $C = \{Fiat, Ford, Audi, ...\}$ and then creates the processes to manipulate information for each car type in isolation.

F. Applying the Model in FlowK

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).

Privileges (as in §IV-C) are only associated with active entities, i.e. processes. Certain processes have privileges, allowing them to change their labels; that is, those processes are able to change their security context $A \rightsquigarrow A'$. *The labels of passive entities (files, pipes and sockets) are immutable.*

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 (2) are respected, that is, E inherits A's labels. See §VI for how this could be used in a PaaS-hosted web-based service.

V. Implementation

FlowK is a minimal trusted code base running in kernel space and enforcing IFC constraints. FlowK is agnostic to label definition and the management of roles and principals. Trusted processes (§IV-E) running in user space manage labels on behalf of applications, i.e., managing policy and assigning

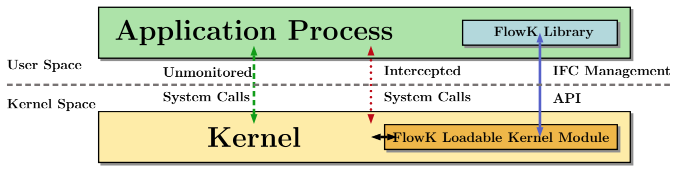


Fig. 2: FlowK kernel module architecture overview.

the appropriate privileges to processes. FlowK is therefore only tasked with *enforcing* the IFC constraints described in §IV. See §VI for an integrating example.

As described in our model (§IV), an entity cannot alter the labels of another. Any change of security context occurs through the explicit action of the process itself. This prevents a number of covert channels [10], [20] and allows the programmer to reason more easily about program behaviour.

A. Architecture

Fig. 2 shows the overall system architecture. We associate each entity with two sets of tags (for their S and I labels), each represented as a 64 bit integer. Processes are further associated with privileges over their tags. System calls creating flows are intercepted and IFC constraints are applied, enforcing IFC according to the labelling; other system calls are left unintercepted. Label manipulation through the FlowK API is done through a static FlowK library connecting to an interface equivalent to a standard device driver, i.e. communication with user space is as for any other kernel module.

To achieve system call interposition, the system call table is patched and IFC system calls replace standard system calls [27], while relying on the underlying standard implementation. Our kernel module maintains a map between entity identifiers (processes, pipes etc.) and their respective labels and privileges. On 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 operation is authorised (and respectively for read and recv).

B. Files

Files are the only entities with persistent labels. Labels are made persistent through extended attributes i.e. only files in a file system supporting extended attributes can carry labels; otherwise, files are considered unlabelled and information flow restricted accordingly.

Unlike other systems [15], we enforce IFC not only on file opening (therefore including creation), but also on other file operations such as read and write. Recall that files have immutable labels (see §IV-F); the labels are assigned to a file when it is created, following rule (2). When a process modifies its security context during its lifetime it may no longer be able to read from or write to a file it created, or could previously access, and should therefore be returned an access error. A process that does not change its security context should never lose access to a file it could previously access. In practice, processes that need to change their security context are rare and have a specific role, such as declassifiers (anonymising, encrypting) and certifiers (validating/verifying input).

C. Inter-Process Communication (IPC)

In this work we are not addressing shared-memory IPC, supported at OS level, and have disabled shared memory for monitored processes in our evaluation. We therefore consider two kinds of IPC: pipes and sockets.

A pipe is created by a process, and therefore the pipe inherits the labels of that process according to rule (2). Pipes have immutable labels; no external factor can change a pipe's labels once they have been created. Suppose a second process is forked, inheriting the pipe's descriptors and labels. The two processes use the pipe to transfer bytes, one writing and the other reading. Rule (1) applies for the flows $process \rightarrow pipe$ and $pipe \rightarrow process$.

Only *UNIX* (*local*) *sockets* support labels since otherwise the other end of the socket might be over a network link, where the remote system does not enforce IFC constraints. Therefore, all other sockets are considered unlabelled and information flow constraints applied accordingly. Communication across systems can be achieved via IFC-enabled middleware, work-in-progress in CloudSafetyNet.

As for pipes and files, *UNIX sockets* carry immutable labels and the labels of the processes using the *UNIX sockets* are checked against those immutable labels. When a process interacts with a socket, information flow policy is applied according to the direction of the flow (i.e. from the process to the socket or from the socket to the process).

For practical reasons, trusted process can mark certain passive entities as trusted. When a passive entity is marked as trusted, information flows between the trusted process and the passive entity are unmonitored. This facilitates e.g. building the security context router described in §VI.

D. Creating New Processes

FlowK is not concerned with enforcing IFC constraints in shared memory, so all threads of a process share the same labels. If one thread changes its (shared) security context, this will change the security context of all threads.

When calling fork or clone to create a new process or thread, the labels of the caller are copied to the newly created entity. The mapping between processes and their labels is guaranteed by the uniqueness of their pid (in the POSIX sense). When a process is closed, the associated labels are dereferenced and freed.

The execve system call allows execution of a program passed by its filename. Suppose the process P calls execve and F is the file to be executed. The labels of the process after a successful execve are such that $S(P) := S(P) \cup S(F)$ and $I(P) := I(P) \cup I(F)$. We fail the execve system call if the resulting label would violate any of the existing CoI groups (see rule (5)). We consider this flow to be a $F \Rightarrow P$ flow.

From a higher level perspective a process P would generally start a new executable by forking a new process Q using vfork, then the child would execute a file F using execve. Using the notation from §IV, we have the following flows: $P \Rightarrow Q$ when the child is vforked by the parent process, and $F \Rightarrow Q$ when the executable is loaded into memory at the time that the child process calls execve. This is the only case in which a process is considered to be created by two parents, and such creation is necessarily restricted by CoI rules. We denote the creation as: $(P, F) \Rightarrow Q$.

VI. Securing Webserver Worker Processes

In this section we show how FlowK can be integrated with applications by creating a framework for possible PaaS-cloud deployed web services. For example, the framework could be used to implement a medical web portal [9] or to provide user isolation as described in [28].

The intent of the proposed architecture is to allow the dynamic creation of Webserver Worker instances (WWs) to perform particular tasks. A WW is subject to IFC constraints reflecting the privileges of the principal on whose behalf it is running, and on the nature of its tasks. We believe this approach can greatly reduce the risks of data leakage.

Fig. 3b shows a simple architecture for IFC-constrained WWs. We aim to minimise the engineering work required to implement applications using this system. The Security Context Router (SCR) is an Apache server running a module we developed; the WWs are standard Ruby-on-Rails servers, enhanced by a library to communicate with the Worker Manager (WM). The Data Store Interface and the WM are bespoke components, but consist of only a few hundred lines of Ruby code. We kept the SCR and WM separate in order to avoid information being unintentionally exchanged between security contexts.

We first describe a single tenant service. In outline, when a user connects to the server for the first time, the SCR routes the request to the Public WW which holds the application's authentication and authorisation policy. Once the user is authorised by the Public WW, it asks the WM to create a new WW for this user's request and specifies the security context in terms of IFC labels/tags (note that several users can share the same WW, if their security context is identical). The user's WW then runs within this specified security context. Once the WW is created, the WM returns a token to the Public WW which requested the creation of this new security context. The Public WW then stores this token within an HTTP cookie for that user. Further requests from this user will be routed to the correct WW depending on the parameters specified in the token. If the token is invalid or has expired, the SCR routes the request to the Public WW.

The lifecycle is managed as follows: the WM sets an expiry time for the security context based on the application's parameters, terminating the WW when the time is reached. Alternatively, an application may know that a context for files was generated to serve some request (or set of requests); once the task they were created for is finished, the WW can terminate itself.

Importantly, security contexts do not need to be pre-defined, and new tags can be generated as needed by any application. For example, user-specific tags can be created as new users register with the application. Tag combinations may exist provided they do not violate CoI constraints (new CoI rules can be defined during the system lifecycle, but obviously cannot be enforced before being declared).

1) Webserver Worker Processes (WWs): In our implementation, WWs are implemented in Ruby and can use any Rack-compatible framework, such as Ruby-on-Rails and Sinatra. The Rack App is augmented to allow applications to communicate with the WM so that they can create new security contexts.

2) Worker Manager (WM): The WM is a trusted process, see §IV-E, responsible for creating new WWs on request. When requesting a new WW, the Public WW specifies the required tags as a string. The WM maintains a local persistent store mapping this string representation of the tags to their local kernel integer representation, see Fig. 3b and §V. All WMs running on a single machine share the same persistent label store. When setting up new WWs, WMs also transmit parameters specified by the application. These can be used to control the WW behaviour, e.g. by specifying the user the new WW is expected to serve.

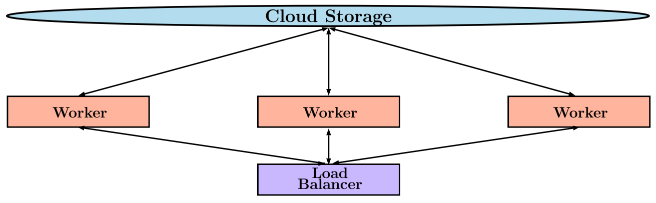
3) Security Context Router (SCR): The SCR is a reverse proxy that routes requests according to a security token. The SCR receives all requests from end-users. These requests contain a cookie representing a token which is verified by the SCR, and the request routed to the corresponding WW. This is transparent to the end-user and the WW. It is possible, but not mandatory, for the WW to verify the token's validity for further security.

4) Data Store Interface: We connect to a key-value store service through a pseudo file interface. The kernel module enforces the same constraints for key/value pairs as it would for a file. We do the same for a memcache instance. Note that in a more mature IFC implementation we might interface with an IFC-compliant database [29] or use an IFC-aware messaging layer based on middleware such as SBUS [21].

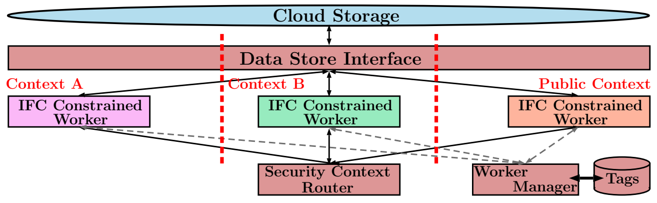
5) Multi-Tenancy: In a multi-tenant environment, the WM will hold the security policies of the tenants to authenticate and authorise their users. The WM also knows the tags that can be allocated to authorised users of each tenant and can check any user request before setting up a WW for the user.

Allowing multiple tenants to share tags would allow them to share data more safely, using the IFC constraints described in §IV. Taking the UK cancer registry (see §II) as an example [9], certain information might be available to all users of the registry, e.g., clinicians may have access to their current patients' medical histories; medical researchers may have access to anonymised data sets, perhaps for geographical regions. Other information may be restricted e.g., to the Brain Cancer Portal service.

The aim of this work is to support protected sharing of data, with isolation when needed and cross-tenant sharing when appropriate, all under strict control and with audit.



(a) Standard webserver worker architecture



(b) IFC-aware webserver worker architecture

Fig. 3: A scalable webserver worker process design running multiple identical instances of an application in different security contexts.

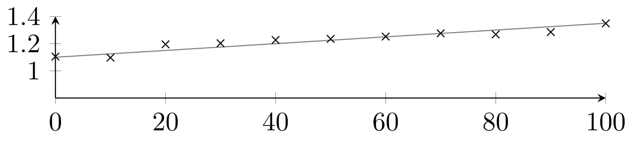


Fig. 4: Pipe performance multiplier (for unmonitored and 0–100 tags, normalised over native (y=1.0) performance)

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 II: Some system call overheads compared with Flume [15]

6) Discussion: Importantly, we do not require large-scale code changes in order to provide IFC within a cloud environment, e.g., our architecture could relatively easily be added to CloudFoundry, by: (1) replacing App by our proposed architecture; (2) modifying the router to understand the security context (SCR) and routing to machines where this security context exists; (3) making the messaging layer IFC-aware; (4) allowing, through the cloud controller, certain service instances to run within a specified security context.

From an economic viewpoint, the pricing approach could be the same as for elastic scaling, i.e., paying on a worker/time basis. An application relying on a large number of different security contexts will cost more than one relying on fewer, while paying only when those instances are needed.

VII. Evaluation

In this section we present evaluations of the FlowK kernel module and our webserver worker process architecture. Although we present performance measurements this work is for proof of concept and has not been optimised.

A. Kernel Module

To measure performance we used a quad-core 2.2Ghz Intel i7 with 6GiB of RAM running Fedora 20 (kernel version 3.14). In order to provide reproducible results and to ease comparison with other projects, we used elements of a standard benchmark, David Nieimi's UnixBench 5.1.3, that we run under IFC constraints.

In Fig. 4, we display the results of a set of tests. Two processes read from and write to a pipe respectively using a 512 byte buffer. The processes' secrecy labels contain between 0 and 100 tags. The results of the test are then used to calculate the multiplier of our system compared to native performance (1.0). At first, the interception cost is an overhead of around 10%, growing with the number of tags. For a complex policy of 100 tags, the overhead is around 35%. Since the test application mostly performs system calls, these results represent a worst case scenario.

We also evaluated our system using local/Unix sockets and file reading. Since the results were effectively identical to those for pipes we do not present them here. Indeed, the interposition cost is almost identical between system calls and the flow constraints being verified are the same. The performance overhead of interposition itself can be estimated at around 10% (the performance difference between the native system and an unmonitored process running on FlowK). The performance hit when using labels is due to the verification of IFC rule (1), see §IV-A. The worst case complexity is $\mathcal{O}(n)$, with a complexity of $\mathcal{O}(1)$ when information flows from a *public* entity to a labelled one.

Table II gives the results of a micro-benchmark for some system calls in order to compare our performance relative to native Linux with that reported for Flume in [15]. System calls were repeated 1,000,000 times (with the exception of the `fork` system call, repeated 1,000 times). The process tested under FlowK was labelled with twenty secrecy tags; the number of tags used in the Flume evaluation is not reported. The performance measured relative to native Linux appears to be better than that reported for Flume.

B. Webserver Worker Processes

We created a similar scenario to [9]. A server retrieves patients' medical records from a database, on request, and places them in a temporary datastore. We have 50 records of around 9kB within a key-value store from which records are selected randomly. We measure the latency in ms as a function of the number of concurrent requests, see Fig. 5. 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 highlight the interposition cost of FlowK.

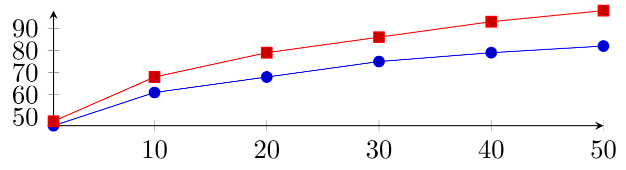


Fig. 5: Performance of our WW process architecture (§VI) with (red/square) and without (blue/circle) FlowK running. Y-axis: latency in ms for 90th percentile, X-axis: number of concurrent requests. Results are averaged over 10,000 requests.

The overhead measured from 1 to 50 concurrent requests varies from 5% to 23%, of the order of magnitude expected for similar OS level IFC implementations [15], [16].

VIII. Conclusions and Future Work

We have described part of the CloudSafetyNet project, to demonstrate the feasibility of providing IFC for cloud computing. If significant reengineering by cloud providers and/or tenants were to be necessary for IFC to be incorporated into cloud services, it would be unlikely to be adopted. Our design choices have therefore aimed to minimise any required changes to existing systems when IFC is deployed. FlowK does not require any changes to system calls so unmonitored processes are not affected by the existence of FlowK as an OS module, apart from a small performance overhead. The security context manipulation is done through a small, well defined set of API calls.

The FlowK kernel module is concerned only with *enforcement* of IFC, following policy-mechanism separation. The management of end-users and policies, leading to principals, privileges and labels is the concern of higher-levels. Our model includes privileged, trusted processes (§IV-E) for this purpose. We envisage such a process per cloud tenant, privileged to set up entities in security contexts to represent the end-users of the tenant's cloud-hosted service. The challenge is to provide both protection (isolation) and information sharing as needed between the users.

To demonstrate this we have designed, implemented and evaluated a simple IFC-aware WW architecture, requiring minimal re-engineering of applications and suitable for cloud deployment. A trusted process manages tenant applications' policies and sets up IFC contexts for the WWs that service users' requests. This is a step towards showing how application policy can be integrated with IFC. Both FlowK and our WW architecture could be deployed in a PaaS cloud with minimal re-engineering.

In FlowK we have a straightforward and efficient starting point for IFC enforcement in cloud computing. Future work will explore application policies in more detail and their enforcement via IFC. We are integrating FlowK with an IFC-enabled messaging middleware (SBUS-IFC) [21] to create more general application structures for distributed and cloud computing. 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.

Acknowledgements

This work was supported by the UK Engineering and Physical Sciences Research Council under grant EP/K011510 CloudSafetyNet. We acknowledge the support of Microsoft for work on Cloud Law with QMUL colleagues.

References

[1] C. J. Millard, *Cloud Computing Law*. OUP, 2013.

[2] A. C. Myers, "JFlow: practical mostly-static Information Flow Control," in *Proc. POPL'99*. ACM, pp. 228–241.

[3] V. Simonet, "Flow Caml in a Nutshell," in *APPSEM-II*, 2003.

[4] I. Papagiannis, M. Migliavacca, and P. Pietzuch, "PHP Aspis: Using partial taint tracking to protect against injection attacks," in *USENIX WebApps*, 2011.

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

[6] A. Yip *et al.*, "Improving application security with data flow assertions," in *Proc. ACM 22nd SOSP'09*, pp. 291–304.

[7] Ponemon Institute, "2014 Cost of Data Breach: Global Analysis," http://www.ponemon.org/.

[8] "Web Hacking Incident Database," http://projects.webappsec.org/.

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

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

[11] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE JSAC*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[12] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model," *MITRE Corp. Tech. Rep. 2547*, 1973.

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

[14] A. C. Myers and B. Liskov, "A Decentralized Model for Information Flow Control," in *Proc. 17th SOSP*. ACM, 1997, pp. 129–142.

[15] M. Krohn *et al.*, "Information Flow Control for Standard OS Abstractions," in *21st ACM SOSP*, 2007, pp. 321–334.

[16] P. Efstathopoulos and E. Kohler, "Manageable fine-grained information flow," in *EuroSys*. ACM, 2008, pp. 301–313.

[17] W. Cheng *et al.*, "Abstractions for Usable Information Flow Control in Aeolus," in *Proc. USENIX ATC*, 2012, p. 12.

[18] I. Roy *et al.*, "Laminar: Practical Fine-grained Decentralized Information Flow Control," *SIGPLAN Not.*, vol. 44, no. 6, pp. 63–74, 2009.

[19] N. Zeldovich *et al.*, "Securing distributed systems with information flow control," in *Proc. USENIX NSDI*, 2008, pp. 293–308.

[20] N. Zeldovitch *et al.*, "Making information flow explicit in HiStar," in *Proc. 7th USENIX OSDI*, 2006.

[21] J. Singh and J. Bacon, "SBUS: A generic, policy-enforcing middleware for open pervasive systems," *Cambridge Univ. Computer Lab. Tech. Rep. TR847*, 2014.

[22] J. Bacon *et al.*, "Information Flow Control for Secure Cloud Computing," *IEEE TNSM*, vol. 11, no. 1, pp. 76–89, March 2014.

[23] R. Sandhu, "Separation of Duties in Computerized Information Systems," in *Database Security IV*, 1990.

[24] D. Brewer and M. Nash, "The Chinese Wall security policy," in *IEEE Symposium on Security and Privacy*, 1989, pp. 206–214.

[25] D. Stefan *et al.*, "Flexible Dynamic Information Flow Control in Haskell," in *Proc. ACM Haskell Symp*, 2011, pp. 95–106.

[26] T.-H. Tsai *et al.*, "A practical Chinese Wall security model in cloud computing," in *IEEE 13th APNOMS*, Sept 2011, pp. 1–4.

[27] N. Dhanjani and G. Rodriguez-Rivera, "Kernel Korner: Loadable Kernel Module Programming and System Call Interception," *Linux Jnl.*, 2001.

[28] R. Cheng *et al.*, "Radiatus: Strong User Isolation for Scalable Web Applications," *Univ. Washington Tech.Rep.*, 2014.

[29] D. Schultz and B. Liskov, "IFDB: Decentralized information flow control for databases," in *EuroSys*. ACM, 2013, pp. 43–56.