Integrating Messaging Middleware and Information Flow Control

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

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

***Abstract*—Security is an ongoing challenge in cloud computing. Currently, cloud consumers have few mechanisms for managing their data within the cloud provider's infrastructure. Information Flow Control (IFC) involves attaching labels to data, to govern its flow throughout a system. We have worked on kernel-level IFC enforcement to protect data flows within a virtual machine (VM). This paper makes the case for, and demonstrates the feasibility of an IFC-enabled messaging middleware, to enforce IFC within and across applications, containers, VMs, and hosts. We detail how such middleware can integrate with local (kernel) enforcement mechanisms, and highlight the benefits of separating data management policy from application/service-logic.**

***Keywords*-Information flow control, middleware, cloud computing, distributed systems, policy, security**

I. Introduction

Issues of data management hinder the more widespread adoption of (public) cloud services, especially for regulated sectors such as healthcare, finance and government. Cloud providers and tenants not only have an interest, but often a legal responsibility in ensuring proper data protection [1].

Addressing these concerns requires mechanisms for enabling *control* within cloud services. Specifically, access to data must be protected (including where it flows), in accordance with user-requirements, contracts and regulations, and compliance with these must be demonstrated.

Cloud isolation techniques (*virtual machines* (VMs) and *containers*) separate tenant data/processing. These are cloud provider-managed, without tenant input. Tenants usually have some ability to manage their data, as relevant to the service offering (IaaS, PaaS, SaaS). General access controls operate between tenants and the cloud providers, and between users and cloud-deployed applications, at *ingress-egress* points: as data enters and leaves the cloud. This means tenants/users lose some ability to manage and audit data once it is in the cloud.

It follows that cloud consumers must rely on the provider to properly manage their data. Though we assume a non-malicious provider—they are businesses with legal obligations—data leakage may occur due to bugs, misconfigurations, over-permissive policies and so forth. Further, tenants and providers may have data handling obligations, and thus must often demonstrate that appropriate policy and controls are in place irrespective of whether a leak has actually occurred.

Current control technologies are, of themselves, inappropriate to meet the pressures described. We are therefore exploring the potential for *Information Flow Control (IFC)* in cloud computing [2]. IFC is a data control mechanism where policy attributes (labels) are attached to data to *give control and visibility as data flows throughout the system*. This enhances traditional, principal-centric access controls offering a complementary, orthogonal mechanism that provides continuous enforcement of data-centric control policy at runtime. Such functionality provides assurance, useful for tenants and providers alike.

Towards this, we have designed and implemented a kernel module, FlowK, that enforces IFC constraints on all data flows within an OS [3] (and thus within a VM). We have not attempted hypervisor-level IFC, though would leverage a verified and hardware-signed trusted substrate, see [4].

This paper presents a *messaging-middleware that enforces IFC across systems*, to protect flows within and between applications and services (e.g. storage), be they local to the tenant, VM, cloud provider or external. We are the first, to our knowledge, to explore IFC at this level of abstraction. We detail how the middleware integrates with a local-system's IFC mechanism (FlowK), with a view towards the goal of end-to-end enforcement. Our approach abstracts policy, meaning IFC is enforced regardless of the involvement (or knowledge) of principals, tenants and applications; but provides mechanisms for direct IFC intervention when appropriate. This facilitates system-wide data management, deployment and control.

II. Background

Middleware provides a layer of abstraction for applications over lower-level communication concerns. It aims to simply communication, targeting intra- and inter-application communication across hosts.

In *message-oriented middleware* (middleware), applications communicate by sending and receiving *messages*: data structures encapsulating a set of related data values. In a cloud context, messaging systems can enable communication between applications and services running in the same VM; between different VMs run by the same providers; and with external machines. These flows result a) directly from users of a tenant's application, or b) as a result of a provider provisioning their service (e.g. interacting with shared data storage services, etc). Many cloud providers offer a messaging service of some form; e.g., RabbitMQ is integrated with PaaS platforms such

as Heroku and CloudFoundry, and available in Amazon Web Services, Rackspace and Google Compute Engine.

Middleware is ideally placed for regulating distributed information flows. By providing the application-level communication interface, it desirably allows the separation of management policy from application-logic. Operating across applications/hosts, this allows policy to control, for instance, which data streams can flow between which system components under what circumstances. Further, messages provide a level of data abstraction that is more easily associated with policy and management concerns (cf. a socket or pipe).
IFC-enabled middleware offers benefits at all service levels:
PaaS: Here, the IFC capability can be directly leveraged by application developers. This gives tenants fine-grained control, as application-specific policy can regulate the data flows (at the data-item level) between applications and/or service offerings (storage, processing agents e.g. billing services, etc.), regardless of whether software is tenant or provider deployed.
IaaS: Though IaaS providers strongly isolate tenants, a tenant deploying IFC-aware infrastructure can control and track flows with other instances of their own infrastructure and also with external applications. In practice, the line between IaaS and PaaS can blur, where IaaS offerings provide default VM images and services that tenants can leverage, while PaaS clouds may simply offer a management tier over an IaaS stack.
SaaS: Though the cloud provider offers and manages the application, being IFC aware enables both SaaS tenants (e.g. corporate email underpinned by Gmail) or direct end-users to a) have visibility over their data flows; and b) with the appropriate interface, label data to specify its flow constraints.

Motivations and Assumptions

This work aims to provide infrastructure for tenants and providers to meet their data management obligations, by the ability to define policy and demonstrate compliance.

It follows that we assume the cloud provider to be non-malicious, bound through legal requirements (legislation, contracts) to protect tenants' data (and possibly that of their users) [1]. Similarly, we also assume that *tenants* do not actively try to leak their users' data. Rather, they are most likely legally bound to protect it, e.g. in accordance with data protection and contractual obligations. Even so, a providers' infrastructure could be misconfigured and leaks could happen through shared infrastructure, and in interactions with any sub-providers (those assisting in service provision). And of course, an application provided by a tenant could accidentally leak data, e.g. via bugs. So tenants and providers have a direct interest in infrastructure aiding management and compliance.

We do not directly consider malicious attacks, though IFC audit data could assist digital forensics. Also, IFC compartmentalises risk by confining the effects of such attacks, by restricting the flow of information between system components, and thus can—depending on the attack—limit the effect of a successful attack through its confinement capabilities.

Here we explore the practical aspects of IFC for distributed systems. Our prototype is an exemplar, highlighting the feasibility and benefits of enforcement at this level of abstraction.

III. IFC MODEL

IFC controls information exchange, preventing leakage. We now briefly outline our model (see [3] for more) that provides the general IFC guarantees of secrecy (*no read up, no write down* [5]) and integrity (*no read down, no write up* [6]).

In IFC, entities, such as a processes (representing a source/sink) and data, are associated with *secrecy* (S) and *integrity* (I) labels. A *label* is a set of a tags, each *tag* representing a security concern. The current state of these labels, and the privileges to manipulate them is the entity's *security context*.

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

$$A \to B, \text{ iff } \{S(A) \subseteq S(B) \land I(B) \subseteq I(A)\}$$

For example, the UK "Government Protective Marking Scheme",[1] representing four security classes, would have its hierarchical secrecy policy expressed here as follows:

S Label	Tag set
$S_{top-secret}$	{protected, secret, top–secret}
S_{secret}	{protected, secret}
$S_{protected}$	{protected}
$S_{unclassified}$	$\emptyset$

Thus for two entities A and B such that $S(A) = S_{secret}$ and $S(B) = S_{top-secret}$, we have $A \to B$ and $B \not\to A$.

Entities may be assigned privilege enabling them to modify their labels (change their security context):
Declassification concerns removing secrecy tags, thus making data more accessible. For example, plain-text may have a *secret* tag whereas that 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; i.e. to create encrypted data without the *secret* tag in its S label.
Endorsement involves adding integrity tags, e.g. input data may need sanitisation before it can be safely used. A *certifier* process can have an integrity label allowing the input of untrusted data, and the privilege to create an integrity label for the verified output data to indicate its validation.

IV. SBUS-IFC: IFC-ENFORCING MIDDLEWARE

IFC in a cloud-integrated middleware is concerned with controlling flows intra-cloud, inter-cloud and between tenant VMs, and users' end-systems. We now present *SBUS-IFC*, an IFC enabled version of the SBUS middleware.

A. SBUS-IFC Overview

SBUS is a messaging middleware that supports strongly-typed messages; a range of interaction paradigms, including request-reply, broadcast, and stream-based; flexible resource discovery mechanisms; and security including access controls and encrypted communication. Importantly, SBUS supports dynamic reconfiguration, where third-parties (subject to privilege) have the same ability to manage an application's communication as the application itself. This simplifies application development and deployment, as concerns can be abstracted

[1] This has changed since this work began. The current version is here: https://www.gov.uk/government/publications/government-security-classifications

and tailored to the environment, rather than embedded within application code. Here we only introduce SBUS concepts as relevant to IFC; see [7], [8] for detailed specifics.

We present SBUS-IFC to illustrate the potential for, and design-considerations of IFC-aware middleware. Similar mechanisms could be implemented in other middleware; we chose SBUS because we are familiar with its code-base, and we envisage its dynamic-reconfiguration capabilities will be useful for controlling applications within managed (cloud) infrastructure, and in IFC label and privilege management.

In SBUS, a *component* is a dedicated process that is associated with an application to manage its (message-based) communication. Each component reflects the set of labels of its attached application. Components (thus, applications) communicate through messages. Labels are assigned within messages. IFC is enforced as messages move between components.

B. IFC in SBUS Messages

SBUS messages are strongly typed, where a *message type* is defined by a schema describing its set of attributes (see [8] for details, and Fig. 2 for an example). For a message instance, an *attribute* consists of a *name, primitive-type* and *value*. SBUS supports hierarchical message structures, meaning an attribute may contain a number of sub-attributes (children).

IFC within messages is fine-grained, applying to attributes. The labels of an attribute a are no more restricted than those of its child c; $S(a) \subseteq S(c)$ and $I(c) \subseteq I(a)$. All child attributes are subject to the same labelling constraints, with the most top-level attribute being the label of the message type.

Labels can be assigned *statically*, defined within message type schema. This sets the attribute's IFC label for all message instances of the type. For those not statically defined, the application producing the message sets the attribute security labels, directly (through the API), and indirectly (according with the application's security context §VI).

C. IFC Enforcement and Privileges

IFC operates to authorise the message flows between components. Enforcement involves the middleware inspecting all attributes of a message, testing for compliance against those of the application/component. This entails a message transformation removing values for the attributes that do not comply.

Enforcement occurs on: *receiving*, enforced before the message is delivered to the application, preventing improper reads; and *sending*, enforced as a component sends a message, so only the appropriate data is transmitted. On sending, any attributes that do not have labels assigned by the application, are set to the current runtime level of the component.

Declassification is possible if the component has the privilege(s) to alter its labels, and changes its security context correspondingly, before message sending (see §III and §VI).

All flows are audited, capturing the security context of the particular message flow. The audit can be configured to include metadata and/or message content pre- and post-enforcement.

Applications do not see the labels within the messages they receive. This prevents a side-channel; though privileged to see message content, they need not know about sending specifics. However, as labels are 'lost' between hops, this prevents the building of (application-layer) networks, e.g. those with content-based brokers. As such, we define a privilege that allows components to view such labels. Though this entails trust, brokers would likely function within a managed infrastructure (e.g. in a government-wide service), perhaps on behalf of other 'sensitive' (top-secret) components.

D. Reconfiguration and application awareness

Given our middleware focus, here we do not explore privilege management issues. However, the SBUS-IFC mechanisms that embody the privileges of the applications/components can also be dynamically and transparently reconfigured at runtime by third parties (policy engines), where authorised.

It follows that cloud software and services need not be IFC aware. Policy is enforced by the middleware without application intervention, including label assignment on sending, where applicable. Further, SBUS' reconfiguration capabilities enable connections between components to be defined and managed at runtime, providing another mechanism for control.

Clearly, this facilitates the development and deployment of cloud software and systems. But importantly it also provides the possibility for management policy to be adapted, when and where necessary, at runtime without application involvement.

V. Cloud Scenario: Monitoring and Admin

We now present a scenario concerning cloud system monitoring and administration to: (1) demonstrate the capability that an integrated IFC-middleware brings to a cloud context (2) indicate the associated performance overheads.

This example uses the system-wide IFC model described in §III, consisting of the security contexts: unclassified, protected, secret, top-secret. To simplify explanation, we only consider secrecy labels S, assuming the integrity labels I are null. Components hold a label with appropriate tags from the above set to describe their current runtime privilege level.

A. System monitoring and administration

Production systems must be properly monitored and managed. Syslog is a common standard[2] for system logging, where syslog messages can encapsulate statistical, debugging and error information produced from applications, services, and devices. These messages give insight into system state, enable better system administration and management.

Logging is a real issue in cloud computing; there are commercial offerings to assist.[3] A cloud service entails a number of VMs, many of which with operating systems running a syslog server (daemon) to which running applications push relevant information about their state, including potential issues or faults. Often, such information must be accessible from remote machines, particularly for data centres, cloud providers, and in distributed systems in general. Thus, many syslog implementations provide remote logging capabilities.

[2]http://tools.ietf.org/html/rfc5424.

[3]For example: https://logentries.com/ and https://www.loggly.com/.

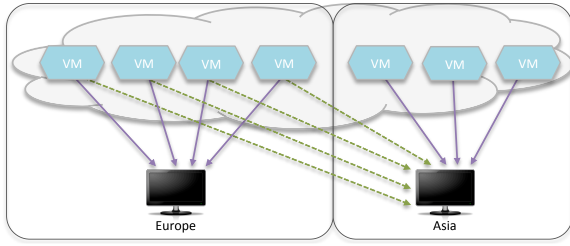

Fig. 1. Syslog information flows from various VMs to administration consoles. Each VM will run a number of applications and processes, some of which will deal with personal and/or sensitive information. There will be circumstances in which log information may flow to other jurisdictions (dashed lines).

Syslog messages can be sensitive. The more detail encapsulated in such messages, the greater the sensitivity of the log. Machines and applications that are used for sensitive tasks, e.g. for billing purposes, or healthcare, are likely to produce sensitive log messages. Thus, syslog data flows require control.

B. System monitoring scenario

Providers and tenants may be subject to data management obligations. For instance, the EU Data Protection Directive states that personal data is considered sensitive, imposing certain requirements, e.g. that personal data must not leave the EU (subject to exceptions [9], [1]). We feel that IFC can help meet and demonstrate compliance with such obligations.

Many cloud services will deal with personal information, e.g. customer data and billing information, and many sensor-driven applications. Log data may come from tenant applications (particularly in IaaS), or from provider-managed infrastructure, such as databases or other shared resources. Some log data will encapsulate sensitive, personal information.

To manage the runtime environment, log information must flow to various locations; e.g. hardware-related issues (CPU/ memory issues) are relevant to system administrators, while application-specific logging may need to flow elsewhere to particular tenants. This must all be properly isolated. To illustrate, we consider the flow of log information to administration consoles.[4] The global nature of cloud means the consoles are geographically spread, in order to meet local legal obligations and also for business/operational concerns (e.g. timezones, financial incentives). In some circumstances, systems may be managed from remote 'jurisdictions', so logs may flow there under certain circumstances. Fig. 1 presents this scenario.

We manage syslog data flows, by using SBUS-IFC to balance the adherence to data protection (location-based) principles, and the need for distributed logs. Our focus is on the interaction between a syslog server for a machine dealing with sensitive data and an administration console, which may belong to the cloud provider (for SaaS or PaaS management) or a tenant (e.g. IaaS).

Scenario Implementation

Each syslog component operates at $S_{top-secret}$ to enable the trusted syslog process to properly view, manage, control and protect the flow of that information. It also holds declassification privileges enabling a relaxation of the constraints for any *(critical)* messages considered important or urgent, that require some active intervention or response.

A critical event is determined by the syslog component, which can be configured to account for various factors, including priority (a syslog field), the application's purpose, the labels assigned by tenants/users, etc. For example, failures in a customer billing service may be particularly important. Note that message criticality is not determined solely by message content. The syslog component actively sets the IFC labels for messages; the console is not so trusted and privileged.

C. Data protection demonstration

We now demonstrate the ability to capture different data disclosure policy, considering an administration console in the EU, and one in Asia that should *not* receive EU personal information (Fig. 1). The syslog server component runs at $S_{top-secret}$. Non-critical event attributes remain at this level, assuming there is no reason for them to be disclosed.

The server declassifies critical events, setting non-specific fields *(priority, timestamp, host)* to $S_{protected}$, and those detailed *(process, content)* to S_{secret}. This makes critical events more visible, though the sensitive attributes are still more protected. We consider four situations:

Scenario 1: The console operates at the lowest privilege $S_{unclassified}$, and thus sees no data. This represents a non-EU (Asia) console that lacks the privileges to view EU syslog data.

Scenario 2: A console running at $S_{top-secret}$ is able to receive all messages, critical or not. This represents an EU console.

Scenario 3: The console runs at $S_{protected}$, to be notified of critical messages, without detail on the sensitive attributes *(process, content)*. This enables the administrators in Asia to be informed of potential issues without violating data protection rules.[5] We illustrate this in Fig. 2.

Scenario 4: The console runs at S_{secret}, thus receiving all critical message details. This may be baseline EU-console policy, to ensure a focus on important events, or may be a deliberate violation of data protection rules, where messages flow to Asia to enlist extra administrative support for mitigating potential loss, e.g. when a system is under attack.

These scenarios show IFC controlling data disclosure, as a mechanism orthogonal to that of access control (which is principal focused) and content-based filters (which focus on data values cf. semantics) [10]. We also see the power of combining IFC privilege management with dynamic reconfiguration. Changing the labels of the console, which can occur at runtime through a single SBUS control message, directly determines the data transmitted/disclosed. This demonstrates the power of the application-logic/policy separation, where simple reconfigurations can realise different functionality, at runtime, without application involvement. Further, this approach paves the way for tenants/users to specify data sensitivity levels, for which the infrastructure should enforce and audit the flows.

[4]These could equally be some aggregation service, storage service, etc.

[5]Perhaps to inform the EU administrators 'on call' outside business hours.

Message as published by syslog server

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

Message read by console running at $S_{protected}$.

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

Fig. 2. Example of a declassified critical message, read-enforced for a process running at level $S_{protected}$ (as per Scenario 3).

D. Performance Overheads of SBUS-IFC Integration

To indicate IFC overheads, we compared the SBUS-IFC with the non-IFC SBUS implementation. Our concern was the relative performance. We used two Macbook Airs (one a Core Duo 1.3 running OSX 10.6, the other a 1.7GHz i7 running OSX 10.9), connected by 100BASE-T Ethernet.

Using a workload of 5000 syslog messages, sent in immediate succession, 20% of which critical, we measured:

1. The IFC read measurement, representing Scenario 3 above, where only some attributes of the critical messages are visible to the receiver. This prevents an unauthorised read-leakage.

2. IFC send considers data leakage at source. This differs from the scenarios above as the server no longer holds the declassification privilege; thus, when sending critical messages, the attribute values are removed before message transmission.

3. Non-IFC is the standard SBUS implementation. As there is no IFC enforcement, there is leakage on send and receive.

Fig. 3 shows the overhead of IFC enforcement. The difference in numbers between the IFC scenarios is because in the send scenario the server lacks the declassification privilege. Enforcing send policy entails analysing the attributes of each message before transmission; this is bypassed when the declassification privilege is held (as in the read scenario).

The results indicate that IFC enforcement introduces **∼13% overhead in performance time** for the workload over standard SBUS. The workload transmission in bytes for the read IFC (where all messages are transmitted) was approximately 1.91MB, and for non-IFC was approximately 1.84MB for all trials. Thus attaching IFC labels to message attributes **introduced 3.6% extra traffic**. The send IFC scenario prevented certain attribute values from propagating, thus transmission was ∼1.52MB (a reduction of ∼17% cf. the non-IFC scenario).

VI. IFC-Enforcing Kernel Integration

We have described middleware that protects flows across machines. It is also important that once data is delivered to

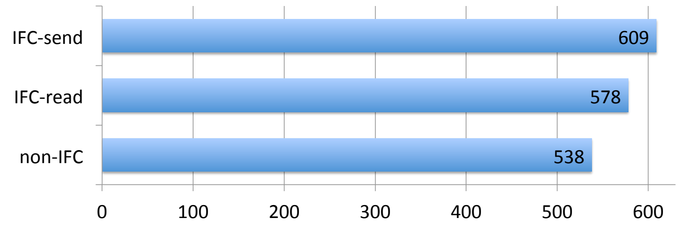

Fig. 3. Performance evaluation between SBUS and SBUS-IFC for the entire 5000 message workload (x-axis time in ms).

the application layer, it remains protected. As such, we have integrated SBUS-IFC with FlowK, which enforces IFC in the kernel space and provides an API for user-space applications. The goal is to enable an end-to-end IFC capability, whereby FlowK enforces I/O within a machine, and the SBUS-IFC protects communication between machines.

In a Linux-like OS, the IFC entities are processes, files, pipes and sockets. Information flows between these are through system calls (FlowK prevents shared memory). FlowK intercepts system calls,[6] enforcing IFC based on the security contexts of these entities, while accounting for (process-level) privilege management. For more on FlowK, see [3].

Inter-machine communication is via sockets. In IFC systems, sockets connected to remote machines have been considered public, thus unlabelled [11], [3]. IFC-aware middleware rectifies this, by enforcing IFC across machines.

Middleware typically uses system calls to provide its functionality. Further, a distributed IFC-middleware must enforce labels consistent with those of the local machine. Thus, to manage the middleware-kernel interplay, we have defined *User Space Helper* (Usher) processes (Fig. 4). An Usher is a FlowK trusted process, privileged with respect to system-calls and the label-management concerns of the application it represents.

Ushers allow FlowK to delegate remote enforcement aspects to the middleware. To integrate with FlowK, the SBUS-IFC process is an Usher. This: (1) prevents FlowK from enforcing IFC constraints on the system calls between the application and its SBUS process, and between SBUS components (leaving IFC enforcement to SBUS); (2) allows SBUS to access and manage the application's FlowK privilege and label allocations, as well as local-to-global label mappings to enable IFC enforcement across machines.

Note that FlowK imposes an overhead of ∼10% [3], similar to the ∼13% observed for SBUS-IFC (§V-D).

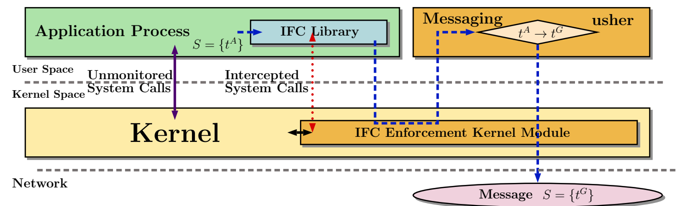

Fig. 4. FlowK architecture, integrating messaging middleware.

[6] We are currently reworking FlowK into a Linux Security Module.

VII. Related Work

Cloud-based access controls typically protect at the point of provider-tenant interaction, but not within the cloud. Encryption allows tenants/users to protect data 'out of their hands', but this brings key-management issues, and generally precludes cloud-processing services. Tenant isolation mechanisms are blunt, provider-centric, and do not account for flexible application-level data sharing. We explore these issues in [12].

A survey of IFC implementations is given in [2]; few consider communication. *Component Information Flow* [13] 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. *DIFCA-J* [14] is an approach that 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* [15] 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* [16], each machine has a dedicated exporter component, through which all inter-machine communication occurs. Leveraging the IFC-compliant OS, the exporter translates between local machine and global labels, to regulate flows across machines.

Our approach differs as it integrates IFC functionality into a general, distributed communications middleware. Further, the work on IFC in networked environments tends to impose constraints on system design, architecture, implementation and/or the operating environment. We made a deliberate design decision not to impose a structure on system design; but rather to integrate IFC functionality into the kind of communications infrastructure already common in enterprise and cloud systems. Moreover, current cloud platforms do not support IFC, which is the key motivator for our work. We have discussed how IFC techniques could be provided as part of cloud services [2]. Integrating SBUS-IFC with FlowK moves us further towards this goal.

VIII. Conclusion

There is real need for cloud users/tenants to be able to specify policy to manage and control their data within a cloud service. To this end, we have been working towards bringing IFC-enforcement capabilities to cloud infrastructure.

In this paper we show the practicality of an IFC-capable middleware. Specifically, we demonstrate the means for IFC control in distributed environments, across applications, services, containers, VMs, tenants, providers, etc. The policy specifying how software, services and applications are used can be separated from their code, thus facilitating deployment and management within the cloud. We showed how IFC-aware middleware can be integrated with a local enforcement mechanism (FlowK), enabling protection both within and across systems. Initial results indicate an IFC enforcement overhead of approximately 10-13%, which we anticipate would be acceptable for many cloud-based application scenarios.

This work moves us towards our goal of end-to-end IFC enforcement. However, naming remains a major consideration: how to design globally unique names for distributed systems is well-known, but to incorporate such a scheme for the tags of applications world-wide requires a global naming specification and a service support framework akin to DNS. Another consideration is structured data. While the middleware can naturally enforce at the level of message attributes (data fields), IFC work generally considers channels and bytes. More work is required on enabling this more fine-grained IFC policy, end-to-end. We also intend to explore IFC for the emerging 'internet of things', where mobile devices, sensors, actuators interact with applications, cloud services and each-other.

Acknowledgement

This work was supported by UK EPSRC grant EP/K011510, and Microsoft (via MCCRC) for the cloud-legal issues.

References

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

[2] J. Bacon, D. Eyers, T. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch, "Information Flow Control for Secure Cloud Computing," *IEEE TNSM, Special Issue on Cloud Service Management*, vol. 11, no. 1, pp. 76–89, March 2014.

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

[4] K. R. Jayaram, D. Safford, U. Sharma, V. Naik, D. Pendarakis, and S. Tao, "Trustworthy Geographically Fenced Hybrid Clouds," in *ACM/IFIP/Usenix Middleware*. ACM, 2014.

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

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

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

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

[9] "European Commission: Proposal for a General Data Protection Regulation, 2012/0011(COD), C7-0025/12, Brussels COM(2012) 11 final," 2012.

[10] J. Singh, D. M. Eyers, and J. Bacon, "Disclosure Control in Multi-Domain Publish/Subscribe Systems," in *Proc. ACM Distributed Event-Based System (DEBS'11)*. ACM, 2011, pp. 159–170.

[11] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information Flow Control for Standard OS Abstractions," in *21st ACM Symposium on Operating Systems Principles*, 2007, pp. 321–334.

[12] J. Singh, J. Bacon, J. Crowcroft, A. Madhavapeddy, T. Pasquier, W. K. Hon, and C. Millard, "Regional Clouds: Technical Considerations," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-863, 2014. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-863.pdf

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

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

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

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