Big Ideas Paper:

Enforcing End-to-end Application Security in the Cloud

Jean Bacon[1], David Evans[1], David M. Eyers[1], Matteo Migliavacca[2], Peter Pietzuch[2], and Brian Shand[3]

[1] University of Cambridge {firstname.lastname}@cl.cam.ac.uk
[2] Imperial College London {migliava,prp}@doc.ic.ac.uk
[3] CBCU/ECRIC, National Health Serivice brian.shand@cbcu.nhs.uk

Abstract. Security engineering must be integrated with all stages of application specification and development to be effective. Doing this properly is increasingly critical as organisations rush to offload their software services to cloud providers. Service-level agreements (SLAs) with these providers currently focus on performance-oriented parameters, which runs the risk of exacerbating an impedance mismatch with the security middleware. Not only do we want cloud providers to isolate each of their clients from others, we also want to have means to isolate components and users within each client's application.

We propose a principled approach to designing and deploying end-to-end secure, distributed software by means of thorough, relentless tagging of the security meaning of data, analogous to what is already done for data types. The aim is to guarantee that—above a small trusted code base—data cannot be leaked by buggy or malicious software components. This is crucial for cloud infrastructures, in which the stored data and hosted services all have different owners whose interests are not aligned (and may even be in competition). We have developed data tagging schemes and enforcement techniques that can help form the aforementioned trusted code base. Our big idea—cloud-hosted services that have end-to-end information flow control—preempts worries about security and privacy violations retarding the evolution of large-scale cloud computing.

Keywords: application-level virtualisation, information flow control, publish/subscribe, policy, cloud computing

1 Introduction

Complex systems are far from bug-free and distributed systems are inherently complex. This is not only because of the fundamental properties of distribution but also because they blur traditional boundaries of data ownership and administrative responsibility. Current approaches to achieving secure, large-scale, distributed systems are not holistic; often, existing abstractions are imperfectly extended (as is the case, for example, for communication of data across networks [1]). Security is an increasingly critical problem as organisations rush to offload their software services to cloud providers. SLAs with these providers currently focus on performance-oriented parameters, which runs the risk of exacerbating security failures.

Fears about security can come from a lack of isolation. It is understood that cloud providers must isolate each of their clients from others. Traditional server and operating system level virtualisation can help, ensuring isolation as appropriate within a data centre. However, each cloud client provides one or more services, each of which in turn may have a multitude of users. We want to ensure that such a service can guarantee that its users' data remains private, even when managed by threads sharing a service's address space.

To see why this is necessary, consider healthcare data. These may be sensitive for a human lifetime or longer and approaches to security that involve only encryption are insufficient; over long timescales, keys may be compromised or become more easy to break. Additionally, we want a way to control, at a fine granularity and in an end-to-end manner, precisely where information is allowed to flow according to policy.

Our vision is that security in the cloud should be incorporated from the start and achieved end-to-end, even when software components may be buggy or malicious. Only thus can cloud clients guarantee that their users' data is safe from cross-contamination or leakage to unauthorised recipients. Cloud service providers therefore need to offer their clients strong but usable isolation support. In effect we want to start with high-level information flow policies that describe how a cloud client manages the data under its control and transform these policies into mechanisms of data isolation enforcement by the cloud infrastructure. We call this *application-level virtualisation*, emphasising that the goal is to provide the illusion of isolated application instances whose interactions are defined by information flow policies.

As a specific example of where this type of isolation is necessary, consider Cancer Registries within the United Kingdom. They compute aggregate statistics about cancer incidence in different parts of the country. Moving these computations into the cloud has the potential benefit of homogenising the processing involved and the data structures used, ensuring that all Cancer Registries operate in the same way. However, there would be a strong need to isolate processing *within this application*—ideally as if there were separate physical resources devoted to each of the sets of data that should not mix. This would match the physical isolation of the data sets between each Cancer Registry that occurs today.

The isolation provided by application-based virtualisation can also be used to protect commercial interests. Suppose that a city runs a service, hosted by a cloud provider, whereby sensors detecting phenomena about the city can be connected with applications that display, process, or store the readings. A company that owns pollution sensors might sell the readings from those sensors to specific customers. The customers, whose applications run in the cloud, should be prevented from sharing the raw sensor data and be permitted to disclose derived data solely based on their agreements with the sensor owner.

For application-level virtualisation to have useful authority, it must express *all* paths that data may follow in any particular system, including messages transferred over networks, data sent within a machine, and information that is written to disk. To realise this, components of the distributed applications that run in the cloud need to specify their communications with others in terms of not only the structure of the data exchanged (as is commonplace with middleware that supports type systems) but their meaning in terms of legal and other obligations. In other words, we argue that software should make explicit the policy context; the attendant information flows describe both security and privacy concerns. Such policy specification must be separate from application code and the correctness of policy enforcement, provided as part of the infrastructure, should be backed up by proofs made against a formal model.

Alongside support for isolation, these explicit security declarations mean that application-level virtualisation allows the cloud infrastructure to do several desirable things:

1. Data flow may be monitored for the sake of audit. Components' description of messages' meaning and relevance to policy mean that the infrastructure is in the position to gather information that not only describes what happened in the course of an interaction but what the components intended to do.

2. The infrastructure may exercise monitoring of compliance with and enforcement of policies. These can codify the types of data that the components will send and are willing to receive.

3. Negotiation facilities for these policies can be provided by the infrastructure.

Overall this means that the details of information flow control and the monitoring of its integrity can be removed from application logic, being placed within the cloud infrastructure for all clients to share. This allows providers to permit clients only the interactions that the clients specify. Ensuring that all communication between components is done using messages annotated with security concerns, the cloud provider can enable end-to-end security that matches the policy requirements set out by the components that make up the clients' systems. This can eliminate disconnects between policies that specify where data may travel and the software paths that are eventually used.

We expect large-scale systems (and those that are based in the cloud are no exception) to span multiple administrative domains, so anything providing application-level virtualisation will have to operate in this environment. A domain is the starting point for naming and authentication of principals, roles, and communication endpoints. Within a domain, it can be assumed that entities are mutually well-known and accountability and trust are relatively high. A domain anchors the expression and enforcement of policy on the rights, obligations, and expectations of principals, although certain policies may be imposed externally by law or as a result of national agreement. A domain must recognise and fulfil its responsibilities for safeguarding data, including the control of data flowing into and out of it.

When a domain is part of a public service, such as in healthcare or policing, public policy can be brought to bear on data protection. With the advent of cloud computing, the protection of the data entrusted to services has not yet been addressed. Indeed, the owners of the services are most likely not the owners of the systems on which the services run. If a major leakage of data occurs, who should be held responsible? Even when there is commonality between stakeholders, such as with the UK Government's cloud computing initiative [2], the scale of aggregation would require inter-departmental responsibilities for data management and security to be made clear.

Paper outline: in section 2, we introduce the key supporting technologies required for our vision: cloud infrastructure, asynchronous messaging and data labelling, information flow control, role-based access control (RBAC), and expression and trusted enforcement of security policy. Section 3 shows how information flow control should be augmented by RBAC and policy to realise application-level virtualisation. We have experimented with this methodology in a number of domains, from healthcare and co-located financial services to social networking, as discussed in section 4. Section 5 continues with an analysis of the major questions still to be answered and provides some proposals for addressing them. Section 6 concludes: we believe that our "Big Idea" could lead to a new standard model for cloud computing, meeting developers' needs for seamless data integration and usability without compromising security.

2 Background

Our plan for application-level virtualisation rests on several pre-existing technologies. In section 2.1, we first describe the infrastructure that cloud providers offer for hosting applications and providing communication between them. For robustness, cloud applications that are large-scale, distributed systems are most easily built using asynchronous messaging (section 2.2). Information Flow Control (IFC) (section 2.3) can sit atop asynchronous communication to curtail leaks of sensitive information and also to prevent inappropriate trust of outside data. Alongside this, parametrised Role-Based Access Control (RBAC) provides a scalable framework for structuring permissions and interactions in large systems (section 2.4). Finally, the security policy must be expressed in a format suitable to the application and requires enforcement by an independent infrastructure, such as a trusted policy module (section 2.5), using IFC to prevent application security breaches.

2.1 Cloud Infrastructure

Cloud services may provide clients with operation at multiple levels of system abstraction and their interfaces come in a consequent variety of shapes and sizes. Some common types of cloud services, and large-scale examples of them, are summarised in this section.

Infrastructure. In this model, clients of the cloud service rent entire computing nodes, such as virtual machines. Although particular operating system templates may be available, the client is responsible for maintaining the operating system and the software that is to run above it. Examples of this model include Amazon's EC2, Rackspace, and Nimbus.

Platform. This model presents a level of abstraction where clients develop their software using a restricted set of programming languages against a particular set of services provided by the cloud. Some platforms may include cloud-hosted incarnations of traditional software such as relational databases but others may provide less familiar functionality such as custom document stores—required when there is necessary coupling between the service and properties of the cloud infrastructure. Google's App Engine and Microsoft's Azure are examples of the platform model.

Application. At the highest level of abstraction, the cloud provides application hosting where clients configure the application on offer but do not have programming-level involvement. Salesforce has been employing the application level of abstraction.

Our focus is on addressing security at the "platform" level. Clients have to redevelop their software against cloud services in any event, and we believe that the additional effort to provide security policy alongside the code would be unproblematic.

2.2 Asynchronous Communication and Data Labelling

Any non-trivial application hosted in a cloud is a large-scale distributed system. To build these, and to provide links between applications, an asynchronous communication model is needed, allowing participants to interact without requiring them to be simultaneously online. Even for cloud services that are stateless (with respect to the cloud servers), message passing is useful for achieving robustness in the face of service reconfiguration and other downtime.

Synchronous communication interfaces can be superimposed on this asynchronous model. At the same time, asynchrony provides a much more accurate view of the realities of distributed operation, with unpredictable failures and delays, distributed multicast used to optimise content delivery, and partial local information. Furthermore, a truly synchronous interface is undesirable because it can act as a covert channel—imagine a message encoded using communication response times.

In our approach, we treat all communicated messages as multi-part structures [3] in which each part has its own data and security label. This allows effective distributed processing because services can annotate the data as needed without altering the security labels attached to the unchanged fields. For example, suppose a pathology laboratory processes a healthcare record and annotates it with diagnostic information. This annotation (a new part) would be of high confidentiality because producing it required access to patient data. However, other parts of the message, such as the identity of the originating pathology laboratory, could remain at their original, lower confidentiality levels and remain accessible to subsequent processors.

2.3 Information Flow Control (IFC)

Information Flow Control (IFC) uses the data labelling within messages to enforce where data may go. Consequently, it is a security technique that guarantees strong protection of data confidentiality and integrity [4].

In IFC, all data are tagged with *security labels* that limit where the information can flow. Each label consists of a set of *confidentiality* tags, describing the "secrecy" of the data, and a set of *integrity* tags that attest to the data's provenance. Data can only flow to processes with compatible labels and data released by a process must be compatible with the process's label. Normally, information can only increase in confidentiality and decrease in integrity as it is processed, unless special *declassification* or *endorsement* privileges are exercised. For example, if a "top-secret" label is more confidential than "secret", then information labelled "secret" can be handled by processes with "top-secret" clearance but not vice versa. "Top-secret" data is therefore confined and can only be declassified to "secret" by trusted processes with the right declassification privilege. A static set of labels is clearly not enough for large distributed systems. *Decentralised information flow control* (DIFC) [5] addresses this, by permitting applications to create their own tags on the fly and allowing privileges over these tags to be assigned dynamically.

Many application domains have shown the value of IFC and DIFC, including military multi-level security [6], operating system process isolation [7], and our own work on event-based distributed systems [8].

2.4 Role-Based Access Control (RBAC)

Distributed applications in general, and effective use of IFC in particular, require allocation, maintenance, and checking of privileges. Role-based access control (RBAC) has been demonstrated as an effective technology for the large scale [9]. RBAC is now in common use across a wide variety of infrastructures, including operating systems, databases, and web-based software. Specific definitions of RBAC are provided by the American National Standards Institute (ANSI) [10], although many software systems implement a simpler version of the RBAC concept. In all interpretations of RBAC, the notion of a *role* is introduced in between principals (e.g., users and processes) and privileges (e.g., method calls and filesystem access requests). Used in this way, roles are essentially a form of grouping. Database and operating system infrastructures often employ this form of RBAC.

The simplest ANSI standard is $RBAC_0$. In addition to the grouping function of a role, the concept of a *session* is included in the model. Sessions are designed to collect a set of active roles that pertain to a given work task. This focuses security management on related sets of roles that are being used, as opposed to having to consider all of a user's potential roles whenever an access control decision needs to be made. This mirrors the way access control tends to work with human principals in a workplace; either functional or organisational roles are activated in appropriate contexts. The activation and deactivation of roles leaves an audit trail as to the intentions of the user over the duration of their session.

For scalable, fine-grained access control, these RBAC approaches are not manageable due to the need for large numbers of roles and the static, simple method of policy specification. The ANSI $RBAC_2$ standard makes steps in the right direction by allowing the specification of additional constraints over the role activation relationships. However, it does not go far enough to effect large-scale, distributed RBAC because its specification of constraints is not sufficiently fine-grained.

Parametrised RBAC mitigates this problem by allowing the connection of dynamically-assigned attributes to roles. The RBAC infrastructure becomes more complex, however, as there is now the need for some inference system to bind the values of parameters during the evaluation of access control rules. Having said this, we have demonstrated that fairly straightforward parametrised RBAC rule specifications with Horn clause form and simple inference semantics are useful and sufficient [11].

2.5 Policy Expression and Enforcement

The techniques of IFC and RBAC are tools that can be used to build a security infrastructure but details of their integration may be below the level at which applications are developed. To solve this mismatch, we need to decouple security specification from the concerns that it protects. What we need is a means of expressing high-level information flow policies and transforming those into appropriate roles and IFC labels. There is usually a trade-off between processing speed and the ability to make sense of the security policy being applied. Policy enforcement requires control flow to cross into specific access control software subsystems, which can add latency on the critical path of some software operations.

Increasingly, software systems are looking to concentrate security management into a focused part of the infrastructure, rather than having security logic scattered throughout the code-base. SELinux and AppArmor provide this sort of policy expression and enforcement at an operating-system kernel level, while the desire to work in environments such as user web-browsers led to the design of Java's system of access control.

One notable policy language that is gaining acceptance is XACML [12]. For application-level concerns, as opposed to operating system or language-level security, XACML has the advantage of security engines having been implemented in a number of programming languages, accompanied by a standardised, expressive policy language.

Regardless of the particular policy languages employed, scaling up the reach of policy requires addressing concerns that cross multiple administrative domains and needs distributed policy enforcement mechanisms; the need to handle cooperating but distinct organisations is common in healthcare scenarios [13].

Small-scale distributed enforcement mechanisms include Kerberos [14], and the widely-deployed Microsoft derivatives of it. At larger scale, short-lived certificates can be used to help distributed enforcement [15], although certificate-based techniques have unavoidably inelegant problems when privilege should be revoked before the certificate expires.

The surge of Web 2.0 applications is starting to beget technology that can effectively manage security in dynamic, heterogeneous environments. The Security Assertion Markup Language (SAML) [16] aims to simplify trust establishment between sites for the sake of user convenience features such as single-sign-on. A related authentication technology is OpenID [17], which provides an approach to managing distributed naming for user identification. Recently, OAuth [18] has emerged as a protocol for evaluating distributed authorisation of privilege.

Many of the above technologies are closely suited to, or even tightly coupled with, specific domains (e.g., web-based authentication or privilege management within a LAN environment). Our aim is to provide security tools for the use of client software, to embed the client software within our security framework, but presume as little as possible about the application's environment.

3 Towards End-to-end Security

Recall that we achieve security by requiring that *all data and communication be protected with Information Flow Control (IFC) constraints.* If these constraints are enforced consistently, the security context is indelibly linked to the data—the security labels can change only when trusted software components deliberately exercise special privileges.

3.1 IFC Compared to Boundary Security

Adding IFC labels to all data allows end-to-end tracking and protection. In contrast, traditional boundary security restricts information only as it enters or leaves a systemic boundary such as a domain. Figures 1 and 2 illustrate the distinction between security controls at the boundaries of a distributed system and continuous, end-to-end security tracking with IFC. In both figures, we see two different events move across two domains. The paths of Event A and Event B are shown as solid and dashed lines, respectively. In figure 1, explicit boundary access control checks prevent Event A from reaching an unauthorised recipient. In figure 2, it is the IFC labels that cause Event A to be blocked before it reaches the unauthorised recipient; Event B is allowed. Even if the recipient republishes

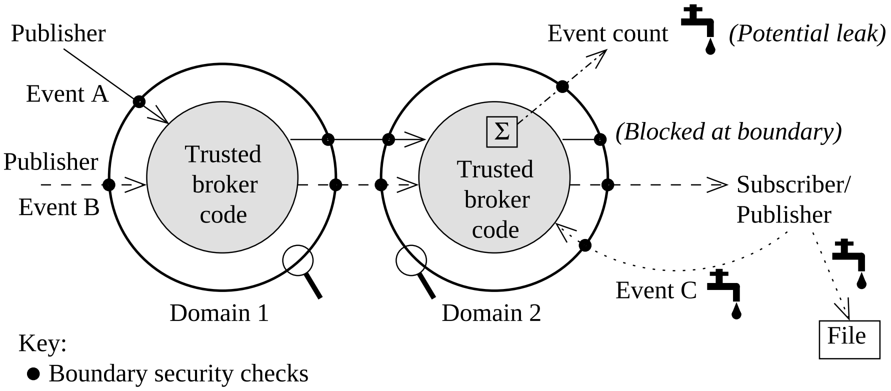

Fig. 1. Boundary security between event publishers, brokers, and subscribers

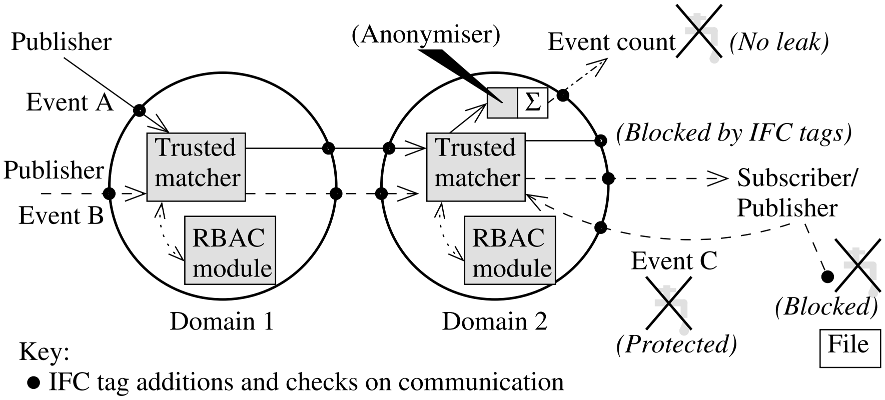

Fig. 2. End-to-end security with Information Flow Control

data derived from Event B as Event C, the same security restrictions are still enforced.

Figure 2 also shows an anonymiser module, which has additional trust in the form of declassification privileges. It uses these to publish an anonymised event count. Other portions of the infrastructure, such as the RBAC module, operate entirely within the IFC framework—all of their communications and data are subject to its restrictions. In contrast, the boundary access controls in figure 1 are insufficient to prevent the event counting component (Σ) from potentially accumulating identified data.

This approach provides an end-to-end model of data security that is the precise definition of *application-level virtualisation*. In effect, each user of an application is isolated from every other user except where trusted code explicitly bridges the boundaries. Here, examples of a "user" could be (1) a single service request, (2) healthcare data related to a single patient, or (3) stock trades executed for a single trader in an investment bank on behalf of a third-party client.

This whole scheme hinges on effective use of IFC. The challenge, therefore, is to make IFC-based systems practical and flexible to use in building complex, large, real-world computer systems. IFC is most straightforward in relatively static, strictly hierarchical environments [6]. As systems become more complex and include interconnections between organisations and departments, themselves constantly in flux, the management of IFC security labels becomes an increasing burden. At some point, the meaning of the IFC-enforced security rules is lost and the code is simply tweaked to make it work. This comes from undisciplined label allocation and use strategies.

Instead, we believe that IFC label enforcement should be linked to organisational security policy, expressed using Role-Based Access Control. This has three major advantages:

Simplified management. Policy can be changed without modifying the deployed code and services can be re-engineered without changing the policy. Furthermore, policy analysis tools can be used to validate the overall application structure at a high level.

Appropriate security. Security policy can be represented using concepts of organisational importance—this is useful for both technical and legal reasons. Technically, this ensures that policy operates at the correct granularity and can consistently follow changes in organisational structures. A large organisation can enforce overall security policy, while allowing additional policy refinements by departments. Legally, it means that data security is linked to concepts important to the organisation; thus data security can feature in SLAs and organisational agreements, and data that leaves the secure processing environment (by being printed, say) can be protected by other legal means such as employment contracts.

Independent enforcement. Policy is translated into IFC labels by a small trusted computing base. Bugs in application code cannot violate the policy.

3.2 Programmer-Friendly, Domain-Specific IFC

Programmers need to be given straightforward ways to develop software that uses application-level virtualisation. We have shown that this can be achieved with simple modifications to a standard Java runtime, enforcing IFC restrictions at the communications API and associating IFC labels with each process's local state [8]. This approach lets programmers write code as usual, with the programming and computational overheads of label checking being systematically localised to the communication boundaries. Furthermore, programmers need only write code in terms of policy, not in terms of individual IFC labels. This simplifies programming, providing a higher-level interface to label management—the RBAC module translates policy expressions to IFC tags, and grants appropriate privileges to the receiving processes. Our experience of this approach is summarised in section 4.

This IFC/RBAC model is well suited to a multi-domain architecture. In IFC terms, all data originating from an organisation has one or more confidentiality and integrity labels by default. Thus, data is always embedded within a security context unless explicitly made public by privileged code.

In the cloud, this protection not only insulates co-hosted organisational applications from each other but it isolates users of shared services because the users can employ disjoint sets of security labels. For example, the security policy can use parametrised RBAC to effectively manage separate IFC labels for each user or role set as in the following policy snippet [11]. Avoiding discussion of the denotational semantics, this policy basically states that being able to acquire the role of a "treating doctor" of a particular patient, involves the role acquirer being a "doctor" that has an active ward-round on the ward that contains that patient. This example demonstrates the relevance of fine-grained management of roles: simply being a qualified doctor is not enough to justify interactions with any patient. It is difficult to use non-parameterised RBAC to encode this policy.

$$\begin{aligned}&\mathsf{duty\text{-}doctor}(\textit{doctor-ID},\ \textit{hospital-ward-ID}),\\&\mathsf{current\text{-}inpatient}(\textit{patient-ID},\ \textit{hospital-ward-ID})\\&\quad\vdash \mathsf{treating\text{-}doctor}(\textit{doctor-ID},\ \textit{patient-ID})\end{aligned}$$

Figure 3 shows a component that provides controlled access to Electronic Health Records (EHRs). Each EHR is assumed to have associated IFC security labels, with each label being a set of tags. Doctor P requests to read the EHR of Patient A; Doctor Q requests that of Patient B. The doctors hold treating-doctor roles with doctor and patient identifiers as parameters. On a read invocation, the service checks the validity and applicability of the requester's active roles. If they are satisfactory, the service assigns labels that allow access to the patient's EHR to the thread that executes the service for that doctor. This means that the threads are isolated—they have no access to data having incompatible labels.

To complete our model of end-to-end security, we need secure persistence and storage for data—real world applications do not maintain their state entirely within events. IFC and RBAC support the integration of database access.

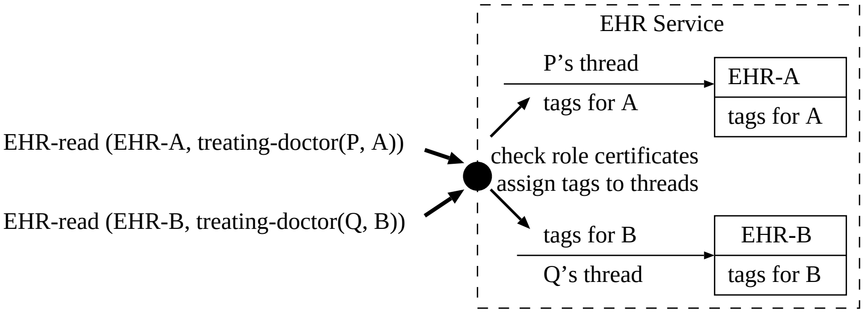

Fig. 3. Isolated threads can access different data

3.3 Integration of RBAC and IFC for Database Access

Databases need to offer fine-grained isolation of information in order to support application-level virtualisation. Otherwise they trivially become channels for unauthorised cross-contamination of data. We express database access restrictions using role-based security policy, using the same policy expression as we apply to asynchronous messaging. This has two main advantages: (1) database security is expressed in common terms with operational data security and (2) roles provide long-lasting security labels that naturally are independent of security policy changes. This provides strong protection for persistent data that may be sensitive for decades. In effect, roles act as bridges, allowing IFC-based information security restrictions to provide end-to-end security both in the cloud and for associated long-term data storage.

This approach extends our earlier work on linking database access control with publish/subscribe systems [19]. By protecting the data in this way, rather than exposing encrypted data to application code, we can provide long-term protection while minimising the risks of key loss or cracking of encryption schemes. Furthermore, it is possible to store data using trusted hardware, preventing disclosure even to superusers and system administrators.

3.4 Current Cloud Offerings

Current cloud offerings do not provide facilities for application-level virtualisation, be it by use of of IFC and RBAC as we have advocated or using other techniques. Here we describe the security mechanisms in Amazon's Elastic Compute Cloud (EC2), and the associated S3 storage service, and Google's App Engine as a representative sample of what is currently available.

Amazon EC2 and S3. Amazon's EC2 is a cloud infrastructure as defined in Section 2.1; it provides facilities to create and destroy virtual machines (VMs) with ease. All of the VMs under the control of one EC2 user (what we have called the "cloud provider client") are linked to that user's Amazon ID and great care is taken to ensure that there is no cross-contamination of data, via memory or disk, between VMs owned by different users. Communication between VMs is constrained using a network firewall. No mechanism is provided to control data flow within a VM.

The storage system associated with EC2, Amazon S3, offers more sophistication. It is a key/value store, where each object has a value that is an arbitrary byte stream. A *bucket* is the container for one or more objects; each object resides in precisely one bucket. Again, Amazon IDs are used as principals, though they are augmented by patterns such as "anonymous" and "any authenticated user."

Access Control Lists (ACLs), which are a subset of simple RBAC as outlined in Section 2.4, may be attached to buckets or individual objects [20, chapter covering access control]. More expressive are *bucket policies*, which are used to specify access rules for all objects in a given bucket. Such a policy is a set of statements, each of which describes a permission in the form of “A is/isn’t allowed to do B to C where D applies” [20, appendix]. A is a principal, B is an S3 operation (such as “put object”), C is a bucket identifier, and D is a set of conditions. These conditions are composed using a simple language combining the expected operators (`NumericNotEquals`, `StringEqualsIgnoreCase`, and so on) with attributes of the request (the time, the source IP address, etc.), content of the request (such as HTTP Referrer), and properties of the target bucket.

This approach is expressive enough for an implementation that uses static IFC labels. For example, these labels could be translated into Amazon users who could then be incorporated into bucket policies. However, this is unsuitable for a dynamic environment because it is awkward to create and delete users frequently. Furthermore, ACLs are low-level constructs and bucket policies may not be modified but only replaced. This means that an intermediate infrastructure would be needed to add and remove security concerns from policies. It is doubtful that constant replacement of policies, and creation of new buckets when a unique policy was required within the application, would provide high performance or be scalable.

Google App Engine. Google’s App Engine [21] provides a platform (in the terminology of Section 2.1) that allows execution of Java[4] and Python programmes within a limited environment. As with Amazon EC2, care is taken to ensure, through effective sandboxing, that one programme does not interact with another, even if those programmes are associated with the same cloud client. Again no facilities are provided to limit the flow of data within each sandbox. Furthermore, the data store provided does not appear to support any access control, permitting the account owning the database complete access to all data.

Based on the above, in its current form App Engine is not suitable for providing application-level virtualisation. In fact, its sandboxing and hence bespoke Java Virtual Machine (JVM) and Java libraries mean that the analysis of the Java libraries that we have done as part of building the DEFcon prototype [8] would not be applicable without further examination.

4 Application Case Studies: Progress to Date

We have demonstrated application-level virtualisation in a number of application domains [22], including health record transfer, cooperative co-location of algorithmic trading systems, social networking, and IFC-enforced policy specification. Here we summarise these experiences to illustrate that (a) application-level virtualisation is feasible and (b) we have identified the necessary prerequisite technologies.

[4] Technically, App Engine supports the Java Virtual Machine, so any language having that as the target execution environment can be made to work.

Health records. When health records are exchanged, long-term data security is critical—for a lifetime or more. We have adapted distributed event-based systems to this challenge, by tightly integrating RBAC with publish/subscribe messaging [13,23] along the lines suggested in section 3. Organisational security policy dictates which data flows are allowed; we assume that each domain has at least one trusted event broker node for secure exchange of events, and uses point-to-point communication between domains. This particular application does not demand high throughput, and health record exchange must follow agreed protocols, so content-based event routing is not required with its sharing of communication paths.

Application-level virtualisation builds on this work and allows a natural deployment of the system in the cloud: using IFC for strong end-to-end security allows security policy to govern not only data exchange but also data processing. Furthermore, enforcing policy with IFC reduces the footprint of trusted code in the messaging middleware and leads to better performance: new IFC tags are needed only when roles are activated to gain privileges, so IFC tags effectively cache policy evaluation.

Stock trading and pairing. We used application-level virtualisation to develop a stock trading application, as part of a high performance event processing platform prototype called DEFCon [8]. Financial applications have stringent performance requirements in terms of throughput and latency. Co-locating clients with investors and sharing facilities among investors is extremely useful for server consolidation in heavily-demanded hosting facilities, and trading systems aim to be co-located with stock exchanges to minimise latency [24,25]. At the same time financial applications are subject to important security policy requirements, including:

1. flow integrity of market data information;
2. strict control of information flows between investment strategies of different clients of a bank and between client and bank investors;
3. confidentiality of orders in Dark Pool trading, used to move large quantities of equities without revealing the trader's identity; and
4. auditing by regulatory authorities on completed transactions.

To satisfy these requirements, we first made the ideas presented in section 2.3 concrete by developing an IFC label model suitable for event-based processing. The model is inspired by OS-level IFC approaches, which provide IFC security among processes in separate address spaces [7]. We applied IFC to processing components that run in the same address space and saw a significant improvement in performance.

An important requirement for financial scenarios is the usability of the target language. We chose Java since it supports high throughput and low latency processing with a tunable garbage collector. Unfortunately, achieving application-level virtualisation is difficult in Java; its security model was designed for isolation between application code and the host system, but does not provide good support for isolation between sections of application code. Crucially, we provided an isolation methodology that can easily be maintained in the face of the continuous evolution of the language runtime.

Our experimental results show that the overhead of checking labels is low: in common applications, labels are quite small and can be checked efficiently without imposing global locking overhead. In comparison, the overhead of using separate Java Virtual Machines to isolate users, both in terms of processing latency and resource consumption, is unacceptable even when the number of users is small.

Social networking. Social networking applications have challenging security requirements: they need to protect their users' individual privacy while still allowing information exchange between these users. As an example, we used IFC tags to protect a Twitter-like microblogging application [26]. Examples of policies that could be enforced are:

1. subscribers to a message topic are guaranteed to have their identities hidden from each other;
2. publications are received only by authorised subscribers and only the publisher of a topic can see the corresponding subscription requests; or
3. publications are received only by authorised subscribers and subscribers' identities are hidden from the publisher.

Our implementation achieves isolation between requests using Erlang's inexpensive process model, with IFC-based policy checks added to Erlang's built-in message passing mechanism. We use IFC restrictions to enforce partitioning of subscription state per subscriber in the message dispatcher. Even in a lightweight application such as this, having essentially no processing overhead, the cost of IFC enforcement is reasonable, being approximately 0.2 ms (of 0.7 ms) per message delivered in an underloaded system. In a distributed deployment, this is small compared to the expected communication delays and computation times.

This example demonstrates two important results: (a) IFC can offer effective privacy protection for users of shared services without preventing appropriate data exchange; in other words, IFC restrictions are usable in practice and are not simply a secure sink for data; and (b) IFC restrictions are compatible with the securing of publish/subscribe communication systems for which the publisher and subscriber sets change dynamically—with microblogging as the special case of one publisher per topic. Whilst we do not expect cloud providers to adopt Erlang, our results show that IFC functionality layered atop an asynchronous message passing facility is sufficient to provide the communication necessary for application-level virtualisation.

Policy specification and compliance monitoring. As we argued in sections 2.4 and 2.5, effective expression of roles and policy is crucial for application-level virtualisation. As a consequence, access control policy for large systems needs to reflect organisational structure, allowing the blending of high-level organisational policy with local policy-based restrictions. Our work has used parametrised RBAC to allow fine-grained enforcement and compact specification [11]; we have shown that this approach can support large and complex organisations, such as the UK National Health Service (NHS). The resulting separation of policy specification and enforcement/compliance monitoring is well suited to cloud computing infrastructure because the deployment platform only needs to enforce policy faithfully but plays no part in its authoring.

In our latest work [22], we explore a flow control policy language in which policy authors explicitly chart how information can flow through IFC-enforcing distributed environments; these span multiple hosting organisations, ranging from corporate intranets to cloud providers. The language uses parametrised flow specifications to link privileges to deployed units of code.

For example, the following is a rule that allows a patient's treating doctor to send the patient's data to a pathology laboratory, which may then process it and send the results back. The pathology lab may also send the data on to a cancer registry for further research studies:

```
NHS.patient-data[doctor-ID, patient-ID]: {
  -> treating-doctor(doctor-ID, patient-ID) ->, NHS.pathlab,
  NHS.cancerregistry ->
}.
```

The effect of this policy is to establish an IFC-protected domain with its own confidentiality and integrity tags. The integrity tags prevent untrusted outside data from being passed off as patient data; the confidentiality tags prevent patient data from being released except as authorised through the policy.

Linking the policy to RBAC privileges has two advantages: (a) privileges can change over time without any change to policy and (b) security policy can be linked to physical security, starting with IFC labels themselves [27,28] and moving towards roles. For example, one can ensure that patient data is released only to a doctor or only displayed on terminals in a physically secure location. This bridge between security policy and physical security allows end-to-end data security to extend beyond electronic data into the real world.

In these flow policy specifications, structured naming is used. For example, `NHS.cancerregistry` could refer to all cancer registries in the NHS, while `NHS.cancerregistry.ecric` could refer to a specific cancer registry and `NHS.cancerregistry.ecric.dropbox` could refer to ECRIC's dropbox service for secure incoming data. This allows high-level organisational policy and low-level operational policy to be specified in the same terms. Multiple policy specifications are automatically combined to determine which flows are allowed; in the above example, the high-level `NHS.patient-data` rule would allow ECRIC's dropbox to receive patient data. Additional low level rules attached to the dropbox could prevent it from releasing the data to third parties, either by specifying additional flow types or by further restricting `NHS.patient-data`.

We are still exploring how best to support policy authoring. At the simplest level, we have established rules to detect clashes between policy entries but further work is needed to provide policy visualisation tools, to establish common policy idioms for organisational data exchange, and to verify policy compliance in large, distributed environments that encompass both cloud providers and corporate intranets.

5 Future Work: Open Research Challenges

Our experiences show that application-level virtualisation is a viable design strategy for providing end-to-end security. Our work to date has illustrated the efficacy of necessary pieces of infrastructure. Moving towards a complete, principled methodology that is usable for building cloud-hosted services will require that the research community address the following issues.

Expression of security concerns. So far we have used labelling of data to convey meaning in terms of security. When operating across multiple administrative domains, we have suggested that the tags within labels be interpreted in the context of the data use agreements between participating organisations [27]. However, it is unclear that this is an ideal mechanism. Tagging of data may feel unnatural to the developer and if data are incorrectly identified, enforcement of security will be incorrect. We need mechanisms, for example, integrated into programming languages, whereby security tags about data can be defined and used as naturally as the declarations of the structure of those data.

Roles as macros for policy. We have argued that many of the tags should be constructed based on role definitions. We must define how this can be done in terms of the desirable RBAC primitives and, potentially, provide a “role toolkit” that developers can use to effect end-to-end security quickly. Furthermore, if we are to map tags to data use agreements expressed using deontic logic [29], we must be comfortable with the advice that we give concerning how to write those agreements. To date, we have begun to provide the mechanism to do this [30], and we have started the process of constructing these agreements based on real-world concerns [28].

Secure mechanisms. How should tags and labels be made secure throughout a distributed system, ensuring that they are not altered and are bound permanently to data as appropriate? How can we be certain that data use always respects tags? So far we have assumed that a small and secure trusted code base exists on each host. Is this realistic and, if not, what are the viable alternatives? What are the implications when high performance is needed for large numbers of clients of a shared service with low latency requirements? How much can be achieved by static checks at the language level?

An alternative is to use a single secure host per domain rather than having a trusted component on each host. We have done this for OASIS [11]. This has the advantage of concentrating domain-knowledge of policies and issuing of tags and labels (such as for role activation or on role use) and, as we have argued, is a natural fit with inter-organisation IFC [27].

Confidentiality of tags and labels. So far we have considered tags and labels to be essentially public in that the receiver of an event may look at the security concerns of the data contained, even if the data themselves are hidden. However, what should be done if tags and labels themselves should be secret? If they are protected using encryption, how should one manage key distribution? How would this key distribution affect efficient event dissemination, for example using content-based routing? Whilst checking back with the issuer of tags raises immediate fears about scalability, are there application areas where it is acceptable because correctness is vastly more important than performance? For parametrised label systems, it may be possible to adopt a hybrid approach, for example by identifying to the receiver that events carry `patient-data` parts without revealing the `patient-id` parameter value.

Policy quality. IFC is only as useful as the policy used to define it. If you specify bad policy, you can release your data. We need a disciplined approach to (a) review policy to check for correctness, and (b) formalisms so that high-level policy can restrict the scope for local policy errors.

Performance. We do not as yet have a comprehensive understanding of the performance penalty that must be paid for end-to-end security in a widely-distributed system. However, our results so far [8] indicate that IFC schemes on a single host can be built into an existing language runtime moderately easily and need not incur a significant performance penalty (particularly not given that network costs are likely to dominate IFC checks). We need to confirm that negotiation of policy concerns between hosts can be done off the critical path used for IFC enforcement.

Special requirements of the cloud. Remember that our primary goal in this paper is to highlight techniques that cloud providers can use to effect end-to-end security. Whilst doing so, we must not lose sight of the additional challenges of deployment within cloud environments. For example, design decisions that focus on a particular operating system or hardware, or require a globally-administered naming system, are unlikely to be applicable to cloud infrastructures.

6 Conclusions

Security in the cloud must be included from the start. This demands a new approach to end-to-end security that supports strong isolation of data, even when business processes are outsourced into the cloud. Cloud processing needs isolation between users of shared services, as well as isolation between services. Our vision of *application-level virtualisation* provides this, by integrating (a) event-based communication for robust service interconnection; (b) strong end-to-end security with Information Flow Control; (c) role-based policy specification that bridges data processing, persistent storage, and physical security; and (d) trusted policy enforcement.

Our research to date has demonstrated the feasibility of this approach in the context of large-scale distributed systems. However, more work is needed to extend this into a standard model for cloud computing that can meet developers' requirements for seamless integration and usability without compromising security.

Our big idea—cloud-hosted services that have end-to-end information flow control—preempts concerns about security and data use violations that are holding back the evolution of large-scale cloud computing. With this, we can reassure cloud users who are worried about cross-contamination within their applications and, at the same time, are reluctant to share a system with other companies and therefore refuse, at the moment, to entrust sensitive data to a cloud.

Acknowledgments

This work was supported by grants EP/C547632, EP/F042469, and EP/F044216 from the UK Engineering and Physical Sciences Research Council (EPSRC).

References

1. Dierks, T., Allen, C.: The TLS protocol version 1.0. RFC 2246 (January 1999)
2. Smith, A.: Open source, open standards and re-use: Government action plan. `http://www.cabinetoffice.gov.uk/media/318020/open_source.pdf` (2009)
3. Pietzuch, P., Eyers, D., Kounev, S., Shand, B.: Towards a Common API for Publish/Subscribe. In: Proceedings of the Inaugural Conference on Distributed Event-Based Systems (DEBS'07), ACM Press (June 2007) 152–157 Short paper.
4. Bell, D.E., La Padula, L.J.: Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA (May 1973)
5. Myers, A., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology **9**(4) (2000) 410–442
6. Department of Defense: Trusted computer system evaluation criteria (orange book) (1983)
7. Krohn, M., Yip, A., Brodsky, M., et al.: Information flow control for standard OS abstractions. In: SOSP '07, New York, NY, USA, ACM (2007) 321–334
8. Migliavacca, M., Papagiannis, I., Eyers, D., Shand, B., Bacon, J., Pietzuch, P.: High-performance event processing with information security. In: USENIX Annual Technical Conference, Boston, MA, USA (2010) 1–15
9. NHS Connecting For Health: RBAC Statement of Principles (July 2006) NPfIT Access Control (Registration) Programme.
10. American National Standard for Information Technology: Role-based access control. ANSI INCITS 359-2004 (2004)
11. Bacon, J., Moody, K., Yao, W.: A model of OASIS role-based access control and its support for active security. ACM Transactions on Information and System Security (TISSEC) **5**(4) (November 2002) 492–540

12. OASIS eXtensible Access Control Markup Language (XACML) Technical Committee: eXtensible Access Control Markup Language (XACML) v2.0. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml (2005)
13. Singh, J., Vargas, L., Bacon, J.: A model for controlling data flow in distributed healthcare environments. In: Proceedings of Pervasive Health 2008: 2nd International Conference on Pervasive Computing Technologies for Healthcare, Tampere, Finland 30 (January 2008) 188–191
14. Neuman, C., Yu, T., Hartman, S., Raeburn, K.: RFC 4120: The Kerberos network authentication service (V5). Technical report, USC-ISI and MIT (2005)
15. Chadwick, D., Zhao, G., Otenko, S., Laborde, R., Su, L., Nguyen, T.A.: PERMIS: a modular authorization infrastructure. Concurrency and Computation: Practice and Experience **20**(11) (2008) 1341–1357
16. OASIS Security Services TC: Security assertion markup language (SAML) V2.0 technical overview. Committee Draft 02 (March 2008)
17. OpenID Foundation: OpenID authentication 2.0 (December 2007)
18. Hammer-Lahav, E.: RFC 5849: The OAuth 1.0 protocol. Technical report, Internet Engineering Task Force (April 2010)
19. Singh, J., Eyers, D.M., Bacon, J.: Controlling historical information dissemination in publish/subscribe. In: MidSec '08: Proceedings of the 2008 workshop on Middleware security, New York, NY, USA, ACM (2008) 34–39
20. Amazon: Amazon Simple Storage Service developer guide (API version 2006-03-01). http://docs.amazonwebservices.com/AmazonS3/latest/dev/ (Retrieved August 25 2010)
21. Google: App Engine Java overview. http://code.google.com/appengine/docs/java/overview.html (Retrieved August 25 2010)
22. Migliavacca, M., Papagiannis, I., Eyers, D.M., Shand, B., Bacon, J., Pietzuch, P.: Distributed middleware enforcement of event flow security policy. In: 11th International ACM/IFIP/USENIX Middleware Conference, Bangalore, India (2010) (to appear).
23. Bacon, J., Eyers, D.M., Singh, J., Shand, B., Migliavacca, M., Pietzuch, P.: Security in multi-domain event-based systems. it - Information Technology **51**(5) (2009) 277–284 DOI 10.1524/itit.2009.0552.
24. Duhigg, C.: Stock traders find speed pays, in milliseconds. The New York Times (2009)
25. London Stock Exchange: Exchange hosting. http://www.londonstockexchange.com/traders-and-brokers/products-services/connectivity/hosting/hosting.htm (Retrieved May 23 2010)
26. Papagiannis, I., Migliavacca, M., Eyers, D.M., Shand, B., Bacon, J., Pietzuch, P.: Enforcing user privacy in web applications using Erlang. In: Web 2.0 Security and Privacy, Oakland, CA, USA (May 2010)
27. Evans, D., Eyers, D.M.: Efficient policy checking across administrative domains. In: Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks. (July 2010)
28. Evans, D., Eyers, D.M., Bacon, J.: Linking policies to the spatial environment. In: Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks. (July 2010)
29. Meyer, J.J., Wieringa, R.J.: Deontic Logic in Computer Science. John Wiley & Sons Ltd (1993)
30. Evans, D., Eyers, D.M.: Deontic logic for modelling data flow and use compliance. In: MPAC '08: Proceedings of the 6th international workshop on middleware for pervasive and ad-hoc computing, New York, NY, USA, ACM (2008) 19–24