Information Flow Control for Secure Cloud Computing

Jean Bacon, *Fellow, IEEE*, David Eyers, *Member, IEEE*, Thomas F. J.-M. Pasquier, *Member, IEEE*, Jatinder Singh, Ioannis Papagiannis, and Peter Pietzuch, *Member, IEEE*

***Abstract*—Security concerns are widely seen as an obstacle to the adoption of cloud computing solutions. Information Flow Control (IFC) is a well understood Mandatory Access Control methodology. The earliest IFC models targeted security in a centralised environment, but decentralised forms of IFC have been designed and implemented, often within academic research projects. As a result, there is potential for decentralised IFC to achieve better cloud security than is available today.**

In this paper we describe the properties of cloud computing—Platform-as-a-Service clouds in particular—and review a range of IFC models and implementations to identify opportunities for using IFC within a cloud computing context. Since IFC security is linked to the data that it protects, both tenants and providers of cloud services can agree on security policy, in a manner that does not require them to understand and rely on the particulars of the cloud software stack in order to effect enforcement.

***Index Terms*—Cloud, data security, information flow, information flow control (IFC).**

I. Introduction

CLOUD computing has matured into providing inexpensive, practical and on-demand access to computing resources. It is realising *utility computing*—the vision of the Grid and other distributed systems before it. One of the least satisfactory aspects of cloud computing is the lack of assurances about security. Unless *cloud tenants* are able to trust cloud providers, the widespread use of cloud computing solutions will be severely curtailed. The problem of cloud security is challenging due to its wide range of legal and technical facets.

The key technical challenge in cloud security stems from the fact that cloud infrastructures combine heterogeneous software and services written by multiple development teams with no shared approach for guaranteeing data security. For example, a cloud provider may rely on *virtualisation* to isolate the computations of different tenants but still share a single data store across all tenants. Similarly, a data store may provide facilities to isolate the confidential data of different users of an application (e.g. via separate user accounts as supported by most database management systems) but such functionality is not typically exposed to tenant applications. Traditional security practices such as access control [1], [2] Chinese Wall [3] and promising technologies such as homomorphic encryption [4] are already being used or considered in cloud environments, but are unable to achieve the flexibility, generality and efficiency expected by cloud providers and tenants.

As a solution, we argue that *data-centric* security mechanisms such as Information Flow Control (IFC)—and *Decentralised* IFC (DIFC) in particular—have the potential to enhance substantially today's cloud security approaches. We envision future secure cloud computing platforms that support the attachment of security policies to data and use these policies at runtime to control where user data flows.

Such data-centric security mechanisms, which track or enforce information flow, can improve cloud security in many ways. First, developers are given the ability to coordinate with the cloud provider and control how user data propagates in a cloud platform. This facilitates compliance with regulatory frameworks. Second, multi-tenancy, i.e. the practice of sharing services between cloud tenants, becomes more secure because the cloud platform can impose checks to enforce security policies despite flaws in the services themselves. Third, tracking data flows across different services offers the cloud provider a way to log sensitive operations on tenant data rigorously, thus improving accountability.

In this paper we investigate the feasibility of deploying IFC as part of the next generation of secure cloud infrastructures, as proposed in [5]. We review research on information flow tracking and enforcement and evaluate data-centric security models. Our contribution is to show that despite the open challenges that remain to be addressed, IFC models and implementations can lead to practical and more secure cloud computing infrastructures.

Section II gives an overview of cloud computing architectures and an introduction to IFC. We also describe the current status in cloud security, and suggest cross-cutting legal and technical concerns relating to the protection of user data. A discussion of the design space of IFC systems follows in Section III, which introduces various comparison criteria for IFC systems and the relevance of these criteria to IFC in the cloud. In Section IV, we review security threats that IFC

Manuscript received January 31, 2013; revised September 25, 2013; accepted November 12, 2013. The special issue guest editors coordinating the review of this paper and approving it for publication were G. Martinez, R. Campbell, and J. Alcaraz Calero.

This work was supported by the UK Engineering and Physical Sciences Research Council [EP/K011510] and [EP/K008129] "CloudSafetyNet: End-to-End Application Security in the Cloud."

J. Bacon, T. F. J.-M. Pasquier, and J. Singh are with the Computer Laboratory, University of Cambridge, UK (e-mail: {jean.bacon, thomas.pasquier, jatinder.singh}@cl.cam.ac.uk).

D. Eyers is with the Department of Computer Science, University of Otago, New Zealand (e-mail: dme@cs.otago.ac.nz).

I. Papagiannis and P. Pietzuch are with the Department of Computing, Imperial College London, UK (e-mail: jpapayan@gmail.com, prp@doc.ic.ac.uk).

Digital Object Identifier 10.1109/TNSM.2013.122313.130423

may not be able to mitigate. Section V describes existing systems that leverage IFC as suggested by ourselves and others. We summarise the challenges for deploying IFC in cloud environments and conclude in Section VI.

II. Background

In this section we give an overview of the three main categories of cloud service provision (IaaS, PaaS, SaaS). For each, we discuss the typical approaches used to secure them. We then introduce Information Flow Control and discuss cross-cutting legal and security concerns.

A. Cloud Computing and Security

Cloud computing [6] is the latest incarnation of utility computing: the notion that computing services can be provided in a manner that is abstracted away from the computing resource itself. A key aspect is the sharing of resources to increase their utilisation: the consequent economy of scale offered to cloud providers allows them to sell slices of resource on demand in a cost effective manner. Over the years, technology developments and tradeoffs often caused computing provision to switch between centralised and decentralised computing. In the early days, processing machinery was bulky and expensive, so resources had to be shared to make them cost-effective. Users often accessed mainframe computers using shared "dumb" terminal devices. The personal computer shifted processing closer to the user but as communication bandwidth increased the advantages of remote server provision re-emerged.

The Internet had always provided some remote access but increasing bandwidth made it necessary to consider computing beyond firewall-protected local administrative domains, giving rise to new security concerns. Web-based, Service-Oriented Architectures took the provision of computing to a global scale. The Grid [7] explicitly draws an analogy between performing computing and the electricity grid. Users should be able to plug in and do their computing work with little or no attention to how the distributed computing is actually orchestrated. While grid technologies were popular for scientific infrastructure, they did not have great commercial impact. Cloud computing gained significant momentum with widespread user adoption of dynamic websites (e.g. for e-commerce). These were typically hosted on servers with PC-compatible architectures and were decoupled from infrastructure, as the development and deployment of high-efficiency PC hardware virtualisation surged.

Cloud service offerings are typically divided into three broad categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). There has been a recent proliferation of other "*Something*-as-a-Service" varieties, but they have not reached critical mass compared to the three we describe. Fig. 1 illustrates the services of each cloud type, and their relation to tenants.

1) Infrastructure as a Service (IaaS) Clouds: IaaS customers rent *computing resources* directly. This form of cloud computing allows tenants most flexibility over the software they use but requires most effort from them: they are responsible for the configuration and customisation of the resources.

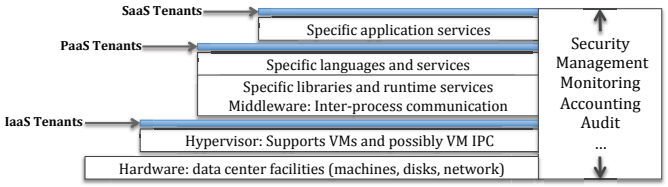

Fig. 1. Cloud service levels and the points of tenant interaction.

IaaS was the first widely available commercial cloud type, initiated by Amazon's launch of their EC2 service,[1] and made possible by the widespread availability of efficient open source hardware virtualisation [8]. Other notable providers include Rackspace,[2] Joyent,[3] Google, and Microsoft. IaaS resources are usually provided to tenants in the form of *Virtual Machines* (VMs). There have been significant recent developments in the management of VM templates, which ease the deployment of new VMs.

In terms of security, the operating systems and software running on the VMs generally need to be managed no differently than on physical, dedicated servers. The exception to this is "paravirtualised" device drivers that are installed into the VMs to increase efficiency. These drivers are necessarily aware of running in a VM. Instead of using expensive emulated device access they typically interact directly with the VM host via some agreed channel. However, there is little that an IaaS user can do other than trusting the paravirtualised device driver authors, or choosing to use much slower virtual hardware via native device drivers. [9], [10] discuss a possible scheme to create a cross-VM side-channel to extract information from a co-resident. As in most systems, an administrator may be a security risk [11].

The key trusted computing base is the hypervisor, or virtualisation host. However, IaaS clouds seldom allow manipulation of the underlying hypervisor configuration. The correctness of the hypervisor has to be assumed, although Microsoft's collaborative efforts to automatically verify a hypervisor [12] are providing significant advances in that area.

2) Platform as a Service (PaaS) Clouds: PaaS customers must develop their applications using languages and service APIs specified by the cloud provider. The supported languages are typically those most popular for web-development. The services provided include facilities such as key-value stores, relational databases, caching systems and various platform-specific functionalities.

For example, the Google App Engine,[4] supports three programming languages (Python, Java and Go) and provides APIs to interact with Google accounts, send e-mail, manipulate images and use various types of persistent storage.

A major incentive for cloud tenants to use PaaS services is that the APIs often give a "scale out" capability, transparently deploying more or fewer resources as required. The tenant need not be concerned with how the PaaS system achieves expansion under high load. Achieving equivalent scalability of infrastructure in an IaaS context would involve setting up services on a large number of VMs.

[1] http://aws.amazon.com/ec2/
[2] http://www.rackspace.co.uk/
[3] http://joyent.com/
[4] https://developers.google.com/appengine/

Achieving high confidence regarding security within PaaS infrastructure is challenging. The services on offer are generally highly heterogeneous, varying in their security offerings, and are often sourced from other projects, which cannot be reengineered. This makes it difficult to achieve consistent security engineering. Further, the scalability of PaaS, and the economy of scale offered to PaaS providers, often mean that software systems are shared by multiple tenants simultaneously. Each service and language environment must separately have its isolation properties verified.

3) Software as a Service (SaaS) Clouds: SaaS customers use applications and/or data hosted by the cloud provider. Often the data being manipulated will remain within the cloud, which avoids the comparatively slow Internet links between the tenant and the SaaS provider.

Google Mail,[5] Google Drive[6] (previously Google Documents), Microsoft Office 365[7] and Salesforce[8] are examples of such services. Unlike IaaS or PaaS offerings, users of SaaS clouds need little technical knowledge. Individual users are unlikely to distinguish SaaS from other types of web-based service. For organisations, a SaaS offering may be customised for the tenant by the SaaS provider. Any further customisation available to the tenant will be using configuration methodologies designed by the SaaS provider.

B. Information Flow Control

Models of secure data access are often classified into Mandatory Access Control (MAC) or Discretionary Access Control (DAC) systems. Traditional and common models such as Access Control Lists (ACLs), capability systems and Role-Based Access Control (RBAC) are DAC systems, meaning that the owner of the data can modify access permissions. DAC systems achieve protection by controlling access to resources. Their implementations often focus on where access control checks are performed in the code of an application. Data is protected as a function of access control checks in the APIs provided to interact with that data. Problems with DAC approaches are that (i) it may be possible to bypass access control checks, especially in web-based systems [13] and (ii) data can propagate or influence system behaviour indirectly in ways that are disclosive, but which access control barriers at discrete points in code do not detect.

MAC systems differ as security policy is defined for the entire system, typically by administrators. *Information flow control* (IFC) is a MAC approach, developed originally from military information management methodologies. IFC is data-centric, and achieves protection by associating *security labels* with data, in order to track and limit data propagation. The labels are also associated with principals in the system. IFC security policy defines permitted relationships between the labels of data and the labels of principals requesting access to data. That is, data protection policy checking can be based on comparing the label(s) associated with the data with the labels held by principals. A simple example from traditional military-style security practice is permitting unprivileged users to pass information to privileged users, but not read privileged information (so-called "no read up, no write down") with matching restrictions on the privileged users. In this case, IFC labels such as *public*, *secret* and *top-secret* would be associated with data items and principals and used to enforce the required security policy.

1) A Model for Centrally Specified IFC: In 1975 Denning [14] proposed a model for secure information flow. The information flow model FM is defined as:

$$FM =< N, P, SC, \oplus, \rightarrow >$$

where $N = \{a, b, ...\}$ is a set of logical storage objects or information receptacles: files, memory segments or program variables depending on the level of detail. $P = \{p, q, ...\}$ is a set of processes, which are the active agents responsible for the flow of information. $SC = \{A, B, ...\}$ is a set of security classes corresponding to disjoint classes of information. A security class $\underline{a}$ is bound to an object a. Processes are also bound to a security class named $\underline{p}$ and are often bound to the security clearance of the user running that process. $\oplus$ is the class combining operator, an associative and commutative binary operator, which specifies for any pair of operand classes, the class in which the result of any binary function belongs. The class of a binary function $f(a, b)$ is $\underline{a} \oplus \underline{b}$. By extension the class of any n-function $f(a_1, ..., a_n)$ is $\underline{a_1} \oplus ... \oplus \underline{a_n}$. A flow relation $\rightarrow$ is defined on a pair of classes. We write $A \rightarrow B$ if and only if information is authorised to flow from class A to class B.

Consider a function $f(a_1, ..., a_n)$ and an object b associated with security classes $\underline{a_1} \oplus ... \oplus \underline{a_n}$ and $\underline{b}$ respectively. In order to write the result of $f(a_1, ..., a_n)$ into b then $\underline{a_1} \oplus ... \oplus \underline{a_n} \rightarrow \underline{b}$ must hold true. For part of the military-style policy described above, we can define security classes *public* and *secret* and three authorised flows $public \rightarrow secret$, $public \rightarrow public$ and $secret \rightarrow secret$. Suppose $\underline{b} = public$; then the result of $f(a_1, ..., a_n)$ can be written into b if and only if none of the a_i are *secret*. In general, a process p is allowed to read a if the flow $\underline{a} \rightarrow \underline{p}$ exists.

IFC can be used to enforce more general policies, using appropriate labelling and checking schemes. The labels can be used to manage both confidentiality and integrity concerns, tracking "secrecy" and "quality" of data, respectively, where quality relates to the trustworthiness of the source of any data rather than accidental corruption, e.g. by hardware. Secrecy concerns where data is permitted to flow to, and integrity where it is allowed to come from. IFC implementations must ensure that labels can be allocated to principals but not be forged by them, can be allocated and "stuck" to data, and that label checking enforces security policy regarding all aspects of information flow. We discuss alternative approaches to implementing these aspects of IFC in later sections.

Practical IFC systems usually cannot work with policies that only allow data to become more restrictively labelled, for example *secret* data passed to a principal with *top-secret* clearance becomes *top-secret* when incorporated at that level. There are situations where privileges should be relaxed, for example, to enable the public release of (previously)

[5]https://mail.google.com
[6]https://drive.google.com
[7]http://www.microsoft.com/en-us/office365/
[8]http://www.salesforce.com/

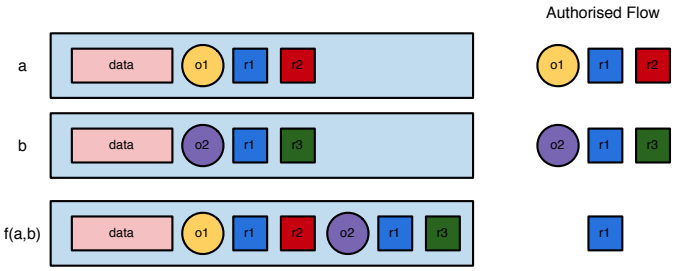

Fig. 2. Labels in Myers' DIFC Model.

classified governmental data, perhaps as a result of some legislation, court order, a verified data desensitising process (e.g. anonymisation), or elapsed period of time. A system of *privileges* operates to introduce carefully controlled additional components into the Trusted Computing Base that can modify labelling contrary to the default restrictions. For example, the privilege to override "secrecy" IFC restrictions is known as the declassification privilege. Nonetheless, IFC remains a methodology that is enforced continuously and is secure by default when compared with DAC schemes. We survey different IFC implementations in later sections.

IFC models for some applications have a fixed set of labels, as in the military example above. For more general IFC systems—perhaps provided by an operating system, middleware platform or language system—new classes of security concern may arise after the IFC system has been deployed, giving rise to the need for new types of label to be introduced dynamically. The representation and enforcement of security policy must be capable of incorporating these new label types.

DIFC models address this need, permitting different parts of the IFC system to introduce new labels into the runtime system, and for the existing security policy to be enforced for these new labels as in the static case.

2) A Model for Decentralised IFC: In 1999 Myers [15] introduced the notion of *security label* to replace the security class of Denning's model [14]. Clearance levels are considered too coarse-grained, permitting unnecessary access and have been replaced by the "need-to-know" principle, also known as Principle of Least Privilege [16]. Therefore, in Myers' model, the owner specifies the authorised readers of the data.

Given an owner o_1 and a list of authorised readers $r_1, ..., r_n$, the label is represented as $L = \{o_1 : r_1, ..., r_n\}$ and the information can only flow between o_1 and the specified readers or between those readers. Consider two objects a and b with respective labels $L_a = \{o_1 : r_1, r_2\}$ and $L_b = \{o_2 : r_1, r_3\}$. These labels are shown in the first two rows of Fig. 2. The "Authorised Flow" group on the right-hand side of each row in the figure indicates the principals that can interact for data labelled with L_a and L_b. Now consider a function $f(a, b)$ that combines data labelled with L_a and L_b. The result of the function $f(a, b)$ will be associated with the label $L_{f(a,b)} = \{o_1 : r_1, r_2 \ ; \ o_2 : r_1, r_3\}$ and therefore will be authorised to flow only towards r_1 (bottom right of Fig. 2).

It seems that a central authority is not needed in such a model since data flow policy is user-specified (discretionary) rather than centrally mandated. However, system support is needed at runtime for the continuous checking of data flows. In order to declassify an information item, all owners must agree to remove their policy. This principle of declassification again appears to remove the need for a central authority, as every owner is responsible for its own policy. But since the processes running on behalf of a principal o_i, or the precise hierarchy of principals, is only known at runtime, declassification also requires runtime support. In this model, users specify data protection policy and rely on the runtime system to enforce it.

C. IFC within Cloud Services

We believe that DIFC is of particular relevance to the cloud, and indeed to any complex distributed system, as in these systems the security infrastructure will typically have an independent and longer life-span than the applications being managed on the platform.

For IaaS, there will be situations that require collaboration across IaaS services, e.g. when tenants wish to share data. IFC provides the means for managing and securing information flows both within and between virtual machines.

For PaaS, IFC potentially provides a security abstraction that is at an ideal level of granularity: mandatory security checks will occur at the interfaces between the software components provided by the PaaS platform, including interactions between tenants. DIFC is even more desirable, as it would allow applications to define their own independent security terminology dynamically.

For SaaS, IFC is of more relevance to the engineering of the SaaS software itself than to the tenants of SaaS clouds. The use of IFC by SaaS providers would increase tenants' confidence that their data is being compartmentalised correctly. The engineering of SaaS-providers' software to use IFC is similar to that for IaaS and PaaS. However, the extra expressiveness of DIFC might not be needed, as the SaaS provider would be able to define the set of security labels in use centrally.

D. General Cloud Security Concerns

Individuals and organisations that use cloud services to store their sensitive data generally rely on the provider to maintain an appropriate level of security. However, it is often the case that agreements (SLAs) between cloud providers and tenants are silent with respect to security guarantees, or even disclaim many types of service responsibility. Further, the global nature of cloud services brings jurisdiction and regulation considerations, which can directly influence the way in which data is managed and governed, in addition to raising issues concerning liability, enforcement, and compensation.

We highlight below some cross-cutting technical security concerns: multi-tenancy, access control enforcement and accountability. We first present some current regulatory issues.

1) Regulatory Framework: Governments have shown concern about widespread use of cloud services. Here, we give a selection of current recommendations by several nations.

Enforcement of the **data protection** policies of cloud tenants is of great importance for companies wishing to push their data or services to the cloud, as they are ultimately responsible for the use and security of their users' data [17]; it is their responsibility to ensure that the policies they define are correctly enforced [18]. In the US, it is a requirement for a company storing medical data to ensure its availability, integrity and

privacy [19], [20], including any medical-related information a company may have about its employees. Privacy-preserving laws within the EU and elsewhere have caused companies manipulating private information (e.g. Google and Facebook) to be faced with the threats of numerous lawsuits.

Another key issue is **compliance** with respect to the rules imposed by a regulatory agency [21], [22] or law [17], [23]. A company hosting data should be able to provide documents concerning data (logs, etc.); this is known in computer forensics as e-discovery [24]. Furthermore, proposals for European Regulation [22] specify that a cloud provider should *comply with this regulation and demonstrate this compliance, including by way of adoption of internal policies and mechanisms for ensuring and demonstrating such compliance*. The report by CNIL (French National Board on Information and Liberties) [21] suggests that corporate rules should be bound to the data.

There are also issues concerning the physical **location of data**. That is, certain information may be required to be stored and/or processed in a location that falls under a particular jurisdiction. Locality requirements may derive from the data owner, who seeks to operate under a specific regulatory regime. Alternatively, such requirements may be legally mandated.

The US Patriot Act [25] and EU legislation [21], [22] restrict where data can flow. French regulation enforced by CNIL forbids copy or transfer of private information outside the European Union, with the exception of a limited number of countries that enforce similar laws (Argentina, Canada, Israel, etc.) and to US companies enforcing Safe-Harbor. Furthermore, CNIL [21] specifically recommends that cloud providers limit flow of private information to locations that have been agreed through a contract with the tenant.

Currently there is no generally available technical mechanism for a user to specify constraints on where his data can be stored, or even to know where it is stored.

DIFC provides a means to control and monitor data flow continuously, according to policy. We therefore believe that DIFC may be a useful tool to help cloud providers comply with regulation—and audit this compliance—more easily in future. Later sections expand on these possibilities.

2) Multi-Tenancy: Multi-tenancy, where a number of tenants share the same infrastructure, may be used by any of the three forms of cloud described above. As such, there is a need to ensure that information from a given customer cannot flow to another, whether or not tenants are actively seeking to view others' data. Note that a tenant may be hosting a multi-user service on cloud infrastructure. IFC supports isolation of individual users' data, not just inter-tenant isolation.

It is possible that a bug or error in the infrastructure design inadvertently allows a user to access the private data of another. Secondly, a user may observe a change in public data while knowing that another user is running concurrently, allowing inference about the concurrent user's data. This case is addressed by non-interference policy [26], which states that private data should not affect (or interfere with) public data. IFC mechanisms can help enforce non-interference policies.

3) Access Control: A security challenge faced by companies and institutions wishing to delegate the hosting of sensitive data to a cloud provider is the management of access rights. The cloud interface may have a less subtle and comprehensive view of access control than is required for the application. Examples are many and varied, including medical health records, customer records of companies, personnel records, and commercial design documents. The cloud provider's staff may be granted access to perform maintenance on the database or in order to provide technical support, which could cause privacy issues [11], [27]. A way to avoid this problem might be to encrypt the data stored in the database. Although some progress has been made towards performing queries on encrypted data [4], [28], [29], existing solutions either require the database to know the encryption key to perform the query [28] or impose limitations on supported queries [4], [29]. Until this issue is resolved, there is perhaps unacceptable risk involved in storing personal or commercially sensitive data on public clouds. IFC would allow applications' access control policy to be carried through to runtime and to provide a self-contained expression of a tenant's data management requirements.

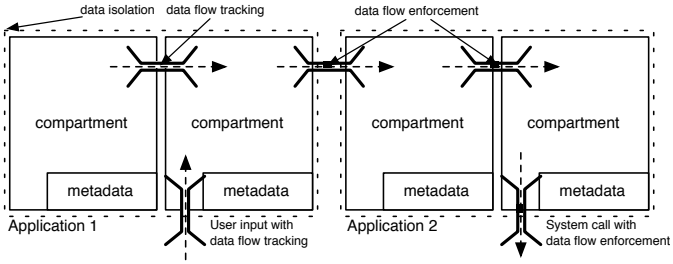

Fig. 3. IFC data isolation, data flow tracking and enforcement between two applications.

4) Accountability: There is a lack of accountability on what operations are performed on tenants' data in the cloud. A tenant should have the right to know if its data has been misused, mishandled by the provider, or transmitted to third parties without its consent. In general, the tenant cannot ensure that the policy agreed with the provider is respected. Even if a tenant trusts its cloud provider, it has no contract with or control over the actions of any third party cloud providers. There is a lack of standards relating to interoperability between cloud providers.

Since IFC tracks all data flows in order to detect policy violations it also has the potential to provide detailed logs for audit purposes.

III. Information Flow Control System Design

In this section we examine and classify the IFC design space with a view to cloud deployment. Cloud computing has particular needs in terms of information flow security. Fig. 3 illustrates possible requirements for two cloud-hosted, interacting applications. Data isolation must be provided between compartments of the applications, and data flows tracked and/or enforced on input, output and inter-compartment and application communication.

We discuss how these can be achieved by using the following four criteria as a basis for comparing IFC provision.

A. When the system operates, (static, runtime, hybrid)

B. How the system isolates data, (e.g. hardware-assisted OS and virtualisation mechanisms, programming language and library mechanisms)

C. How the system tracks data flow across isolated data, (e.g. domain level, process level, variable level, message level) and

D. How the system uses the output of data flow tracking to enforce data flow (how policy is specified, the structure of label metadata, and the declassification of security data)

Security engineering involves tradeoffs between security and efficiency. The designers of IFC systems will select their threat models to inform any compromises they need to make within the IFC design space. In the following sections we discuss threats against which IFC systems provide protection, followed by a discussion in §IV of threats that are not covered by most IFC systems.

A. Options for When an IFC System Operates

1) Static Methods: Static methods for data flow analysis, while not directly relevant to the runtime enforcement of IFC in the cloud, can be used for certifying that cloud software components and their interactions are safe before their deployment. In the case of PaaS for example, this may allow a verified trusted base to be developed on top of which cloud tenants can build their applications.

We outline two representative examples: taint analysis, and security-typed languages, presenting references for further detail. This is not an exhaustive coverage of static methods; e.g. we do not cover symbolic execution techniques [30]–[32], nor the use of theorem provers to verify IFC policy [33]–[35], which can be useful for ensuring information flow control within applications [15], [36], [37]. See [38] for a survey of languages for information flow.

a) Taint Analysis: Taint analysis [32], [39]–[41] is a method for identifying the illicit use of untrusted data. Its techniques are similar to those of source code data flow analysis [42]. The original notion of 'tainting' relates to the risk of potentially damaging data being introduced into software by malicious users. It is a common programmer error to treat user input as 'safe' internal data whereas it is potentially 'tainted' data which must be sanitised before it can be treated as safe. This can lead to serious security problems. Taint analysis usually works at the granularity of programming language variables, and tracks whether or not each variable is potentially 'tainted', (e.g. it has incorporated potentially dangerous user data). Taint analysis can help ensure unsanitised data is never used directly in system calls, or other constructs that might facilitate user attacks via tainted data. For example, SQL queries must be considered potentially tainted in order to prevent SQL injection attacks.

The main limitation of taint analysis is the lack of runtime information. This means that static taint analysis needs to be pessimistic regarding program structures, such as conditional branches. This pessimism often leads to an over-conservative analysis of where tainted data propagates within software.

In a cloud context, cloud tenants' software is generally not amenable to static taint analysis, since it is often deployed dynamically. Also, a protocol would be needed for reporting back taint analysis results, coupled with a means for tenants to provide code path assertions to improve the usefulness of the taint analysis results.

b) Security-Typed Languages: Security-typing has been the subject of several research projects [43]–[46]. In their seminal work Volpano and Smith [47] suggested augmenting a traditional language type system with data flow annotations. This allows developers to express confidentiality and integrity data flow policies that are enforceable by the compiler. This work inspired Jif [15], [48], which uses the JFlow policy language. Jif is described as a *security-typed language*, as data flow requirements are explicitly declared as part of the type of each variable. This enables the enforcement of *non-interference* [26]—where data belonging to one security category cannot interfere with another—to ensure confidentiality and integrity as appropriate. Further, as programs are *composable*, they can be combined into a larger program that also enforces non-interference [38]. Jif has influenced a number of languages, such as FlowCaml [37], and systems [49]–[51] that we discuss in §V.

Jif [15], [52] extends Java by adding DIFC labels to its type system. The compiler examines all program statements based on the DIFC labels of the variables involved and the semantics of each Java operation. The compiler ensures that data associated with a certain label does not reach a variable or a communication channel with a more permissive label. If this happens the compiler generates compilation errors. The compiler then generates a program with additional runtime taint tracking mechanisms—Jif is therefore a **hybrid** system by our classification criteria. The generated program propagates data according to the data flow requirements stated by the labels. It is able to run on an unmodified Java Virtual Machine (JVM).

Adoption is a major issue for security-typed languages because developers would have to use a novel programming language, with static typing and low-level, label-based data flow policies, for writing or rewriting applications. For this to be feasible, such languages must offer sufficiently rich libraries, and users must have access to training and support or specialist, trained developers.

Static techniques have their place in IFC systems, for example for verifying the correctness of long-lived IFC system components. Ensuring that the data of dynamically arriving application components is accessed and transferred as specified by policy is more challenging.

2) Runtime IFC Methods: Runtime enforcement of IFC in the cloud is the main focus of this paper. First, we outline a simple runtime technique, taint tracking.

a) Runtime Taint Tracking: It is a minimal form of IFC; a technique for analysing and enforcing data flow in applications [53]. It assumes only two types of data: tainted/untainted which might be secret/public (to capture confidentiality) or untrusted/trusted (to capture integrity). The tainted data often represents user input data which is untrusted and should not be used in sensitive sections of the code. Other tainted data could be sensitive information that should not leak out of the application or should flow only through well-defined channels.

Taint tracking became of interest to the research community through its use to prevent cross-site scripting or SQL-injection [54]–[59] or to detect suspicious information flows in applications [60]–[66]. When such a flow is detected, the system could either generate a report, perform automatic data sanitisation or terminate the execution depending on the

purpose of the application concerned.

Runtime taint tracking is a simple technique for developers to understand and use. While it can involve a specific security language, conceptual simplicity has motivated research on systems that expose taint tracking to developers as a stand-alone security mechanism [49], [50], [58], [67]. Such systems use developer input to devise a data flow policy and then restrict processing as a result of that policy. Assuming that the part of the system that enforces data flow policy has been implemented without error, correct data flow is ensured in any application built on top of it. Thus, in contrast to security-typed languages, this allows developers to continue to use familiar languages, only having to learn how to interact with the taint tracking system.

b) Runtime Label Tracking: Taint-tracking systems can be seen as the simplest form of runtime IFC. More general runtime IFC methods will manage many different, and possibly orthogonal, notions of data security in their label metadata. For both taint and more general label tracking at runtime, the program statements that an application executes, as well as their execution order, are known. These include statements generated dynamically, e.g. when fetching code at runtime from remote locations. At runtime, the results of program statements are also known to the tracking system. The analysis therefore can focus on the current execution and not on alternative paths, which may never affect data flow (but noting implicit flow—§IV-A). These properties render runtime tracking suitable for data flow analysis in large systems written in dynamic languages, which are hard to analyse statically.

B. Data Isolation

Data isolation is a prerequisite for effective data flow tracking. It prevents the application from exchanging data using mechanisms that are not explicitly controlled or monitored by the runtime IFC system. When data exchange is monitored by the IFC system, the results of data flow analysis do not contain false negatives (i.e. data flow that occurs in the application but is not detected by IFC tracking).

To achieve data isolation, the system first separates the analysed compartment from its environment. All outside communication must be inspected by the IFC system, if it is to prevent any unmonitored access to external data.

A cloud-specific isolation concern is the need not only to isolate tenants' use of resources, but also to support tenants' abilities to provide multi-client services. Here, isolation of individual clients' data is required, while allowing the sharing of data when appropriate for the application needs.

There are many ways in which isolated compartments can be created, a selection of which we cover below.

1) Hardware-Assisted Isolation by the OS: All common, modern operating systems provide at least some support for isolation of software systems. At the very least, there is a separation between kernel and user-space. For a long time many monolithic kernel designs caused the whole kernel to be a trusted computing base (however technologies such as SELinux are beginning to change this). This isolation is usually hardware-assisted, e.g. memory protection mechanisms and 'ring levels' at which code runs within CPUs.

With large trusted computing bases, it is difficult to adhere to the Principle of Least Privilege (POLP) [16]. POLP would ideally require applications to be programmed as components with appropriate privilege separation. Runtime IFC instead allows for the POLP notions to exist in the IFC policy, and unmodified applications checked against that policy.

Most operating systems provide additional isolation facilities such as Unix-style "chroot" tools, to limit the accessibility of the filesystem to contained processes.

It has been observed [68] that application authors generally find working with such mechanisms too cumbersome, so they create application code that is significantly over-privileged. Any vulnerabilities can then have significant impact on the applications. It is argued that the OS should support IFC directly instead of offering only general features on which to build POLP.

2) Isolation via Virtualisation: Data isolation can be achieved using virtualisation technology by placing the IFC system "below" all the software and hardware running unmodified application code. The virtualisation framework modified for IFC support need not be that used by the cloud hosting provider. For example, in Argos [69] the QEMU [70] virtualisation framework is modified to extend the target code so that it defines isolation regions and checks information flow metadata. This is useful for analysing unmodified binary code, such as malware, but has the downside of executing 15-30 times slower than the original code.

In contrast, the Flicker project descends below the OS and the hypervisor to form an independent but minimal trusted computing base supported by new hardware features [71]. The CLAMP system [72] works in a similar way, using lightweight VMs to add isolation to unmodified web applications. Similarly, Payne et al. [73] control data flows between different VMs running on top of Xen [8].

3) Isolation in Programming Languages and Libraries: For IFC systems built with static methods, data isolation is usually defined by attaching IFC metadata to existing compile-time software elements.

More dynamic methods of language-based IFC will rely on features of the runtime system, for example the taint-tracking facilities in the Perl and Ruby languages.

Finally, it is possible to gain many IFC benefits without requiring a modified runtime environment, simply through disciplined use of libraries—although isolation will not be strictly enforced in such cases. For example, shared-nothing actor frameworks [74] and the disciplined use of specific APIs to interact with the environment can provide isolation.

C. Data Flow Tracking

Assuming the isolation of data can be achieved, the next key aspect of IFC systems is tracking how data propagates throughout the application. IFC systems generally have a mechanism to classify each isolated component with respect to its required security properties.

The data flow tracking part of any IFC system is responsible for maintaining accurate metadata as data moves between the isolated components. Other parts of the IFC system will be informed about the tracked flows, and can react appropriately.

One key design decision in any data flow tracking system is the granularity at which the tracking operates. Since IFC systems are designed for a certain granularity of tracking defined by the IFC model, in most cases it is not possible for code using the IFC system to alter this granularity. There is a predictable design tradeoff: using a finer granularity provides more precise information about the data movements in the application, but causes higher overhead.

Operation of a data flow tracking system involves:

1) intercepting the exchange of data,
2) inspecting the intercepted data, and
3) updating information flow metadata as necessary.

Since the tracking operations must be invoked at every data exchange, it is crucial that their operation is efficient. Further, it is particularly important to track data flows into and out of the information flow system—e.g. when software interacts with external storage or other processes.

IFC tracking can be made more efficient simply by reducing the number of interaction points that are monitored. Working at a **domain level** allows most interactions between software components to proceed: an example might be kernel and user-space domains. This matches the common need to track interaction between application software and the underlying operating system.

A more fine-grained approach is **process level** tracking, e.g. as used by Asbestos (§V-B2). However, this will not track the fine-grained interaction of data within software in the manner of the **variable level** tracking within Jif (§II-B2). Our work on the DEFCON (§V-C1) and SafeWeb (§V-C2) systems shows that event-based systems provide another level at which to track data flow: at the **message level**.

As discussed in §IV, there are still likely to be potential side-channels of communication that are available to malicious processes regardless of the data flow tracking granularity.

D. Data Flow Enforcement

The enforcement part of IFC systems involves IFC policy being checked, and action taken if such a policy is violated by a given data flow. Note that checking policy may be more resource intensive than just tracking data flows, see §III-C. This is why many IFC systems choose to perform enforcement only on key isolation boundary crossings, even if the policy restrictions could theoretically have been enforced earlier.

Any enforcement approach must first allow a means to **specify policy**. One common approach is to define a "can-flow-to" relation among DIFC labels, for example using JFlow, or tag-based approaches (see §V-B1). For IFC systems that aim to assist cooperative developers, such as Resin (§V-D1), an alternative is to allow these developers to write their own policy checking code, as part of their application.

In systems with explicit security labels, there are still many options for the **structure of label metadata**. At one end of the spectrum are systems with single-bit taint metadata. At the other end are DIFC label systems that maintain integrity and confidentiality information.

IFC systems **enforce data flow policy** at a particular granularity. For static IFC systems, compile-time errors can indicate that developers' software violates IFC policy. For example, if potentially tainted data input from outside the compiled software was being used in sensitive system calls, the software needs at least to ensure sanitisation of the potentially tainted data.

Simple runtime taint tracking systems can enforce domain-specific policies with little, if any, input from the application administrator or user [13], [75], [76]. Such a 'fixed policy' approach allows cloud providers to ensure a base level of security, and is particularly useful for the administrators of IaaS and SaaS cloud services, as the respective application developers and users have minimal input into or control over the platform. IFC approaches that allow applications to manipulate labels (to various degrees) [49]–[51], [58], and control the actions of checking operations [77], suit PaaS services as they enable application developers to customise the management of their data flows.

Finally, the data flow enforcement within many IFC systems provides a means to **declassify** data to lower levels of secrecy, or **endorse** data to higher levels of integrity. As discussed above in §II, most practical applications cannot have data only increasing in secrecy while losing integrity. Declassification (or endorsement) privileges provide software components with a way to modify the security labels attached to data in a manner that reduces security. However, they also mean that, for example, a system can produce public output, when sensitive intermediate data has been vouched as being appropriately desensitised. Clearly the use of such privileges expands the trusted computing base, and must be done with care.

IV. Threats to IFC Systems

IFC is a robust access control methodology, but not a security *panacea*—this section explores some threats that IFC does not typically protect against. For many IFC systems, though the environment is considered hostile, the application code is generally considered not to be explicitly malicious, even if the threats may be caused by implementation errors [59]. For instance, taint tracking systems commonly assume a *benevolent developer* [56], [77], that is, a non-malicious developer that does not actively try to evade tracking. This assumption reflects the fact that runtime taint tracking systems cannot guarantee that all data flows are monitored effectively. This assumption may or may not be safe to hold in multi-tenant cloud contexts, depending on the relationship between tenants and the cloud provider. We now explore some threats that are not commonly addressed in IFC systems.

A. Implicit Flow

One of the weaknesses of runtime information flow control is the difficulty of tracking *implicit* information flows [78]–[81]. *Explicit* flows from x to y, noted $x \Rightarrow y$ are caused by passing data between variables, for example: `y := x mod 2`.

An *implicit* flow of information arises from the control structure of the program, for example

```
x := x mod 2
y := 0
if x = 1 then y := 1
```

illustrates the *implicit* flow $x \Rightarrow y$ equivalent to the *explicit* flow `y := x mod 2`.

It is possible to track such an assignment by introducing a process sensitivity level [82], in which case the assignment of y can be detected at runtime. We could consider that any variable modified within the **if** statement (or any function called from it) must be assumed to create an information flow. However, in the case $x = 0$, no value is assigned to y and therefore no flow is detected even if it exists.

It is possible to prevent such flows remaining unnoticed by applying the label from the **if** to any assignment happening after the **if** statement. However, this means that the number of labels assigned to variables will increase [81] often unnecessarily. This leads to data with higher sensitivity than intended, known as *label creep* [1]. The following example illustrates this phenomenon:

```
x := x mod 2
z := z mod 2
y := 0
w := 0
if x = 1 then y:= 1
if z = 1 then y:= 1
w:= x mod 2
```

From the Denning model, described in §II-B, we expect that $\underline{w} = \underline{x}$; that is, x and w are of the same security level. However, if we enforce process sensitivity levels, we have $\underline{w} = \underline{x} \oplus \underline{z}$ even if we know there is no $z \Rightarrow w$.

To address the concerns brought by the benevolent developer assumption, it has been suggested that an implicit flow can be prevented by the preemptive halting of program execution [79], [83], [84]. However, this could prevent legitimate applications from terminating [80]. Therefore, to deal with potentially malicious code, variable-level runtime taint tracking can be combined with static analysis techniques [76].

B. Other Covert Channels

A *covert channel* is an unintended method of communication and thus violates, or operates outside of, a security policy. With respect to IFC, any unmonitored data flow is considered a covert channel [85]. Clearly, implicit flows may give rise to a covert channel, though arguably these are more easily identified than those that arise from the applications' runtime behaviour. *Storage* channels [86], [87] are covert channels where communication occurs through (reading/writing to) a shared resource, while *timing* channels [86], [87] concern communication through the timing of particular operations. In the case that an attacker is able to observe program termination or non-termination, a program of the type **do** something() **while**(h=1) is not safe [88]. This is known as a *termination* channel. *Probabilistic* channels can be defined as follows: interference rules [26] stipulate that private data should not interfere with whether or not a given event occurs; however, this does not stipulate that such private data cannot interfere with the probability at which such an event occurs [89], [90].

Side channels refer to the leakage of information through sanctioned use of the system. In contrast to covert channels, which are illicit, side channels refer to the unintended leakage of information through use of the system in a manner consistent with data flow policy. For instance, knowing when/where data of a particular classification is communicated might reveal sensitive information, even if one cannot access the data itself.

We highlight these risks to ensure a false sense of security is not bestowed on DIFC approaches: IFC is just one piece of the cloud security puzzle. There are approaches that can help address the above security threats, but many are so highly disruptive (e.g. synchronisation approaches to reducing timing channels) they are infrequently used. If they are necessary, they can be applied in parallel to DIFC models.

V. DIFC System Implementations

Here, we summarise selected work on implemented DIFC systems that could contribute to adoption of DIFC within the cloud. Their features relevant to cloud deployment are compared in Table I. We cover IFC systems that operate in hardware then some implementations of IFC within operating systems. All share or have gained inspiration from Myers' Jif DIFC label model (§II-B). IFC implementations at the middleware level and as language libraries are then described. Finally, we consider IFC provided at these system levels for possible integration with the IaaS, PaaS and SaaS cloud architectures described in §II-A.

A. IFC Protection in Hardware

Some IFC schemes target custom hardware. RIFLE [91] translates normal binary code to run on hardware that supports IFC tracking. To avoid the pitfalls of implicit flow inherent to all dynamic systems, see §IV-A, all implicit flows are translated to explicit flows. Suh et al. [92] present a hardware mechanism to track information flow. The authors modify how standard instructions behave to propagate tags and add an additional cache to store those tags. CPU registers have an additional bit to denote tagged data.

B. IFC Enforced by Operating Systems

When IFC is enforced by Operating Systems, IFC tracking is typically done at the process level. Processes and persistent data are labeled, and labels are propagated when persistent data is accessed and when inter-process communication occurs. Asbestos (2005) [49], [93] is a fully IFC-capable OS, albeit with a non-standard interface. Flume (2007) [51] runs on top of a slightly modified Linux OS and intercepts system calls to enforce IFC. DStar [94] enables IFC in distributed systems, by translating the security labels between instances of IFC-enabled OSs. Aeolus runs Asbestos [95] across a distributed system by providing cross-host communication and IFC tracking.

1) Flume: Flume [51] runs on top of a modified Linux OS, intercepting system calls to enforce IFC. Its label model draws on that of Jif [48] but is less complex than Asbestos.

Flume uses tags and labels to track data in the system. A tag t simply serves to carry some information; e.g. tag a may represent Alice's private data. A label L is a subset of the tags available in the system. Each Flume process has an associated label for integrity I and for secrecy S. If a process p has seen Alice's data associated with tag a then its secrecy label S_p is such that $a \in S_p$. In order for p to publish any data to

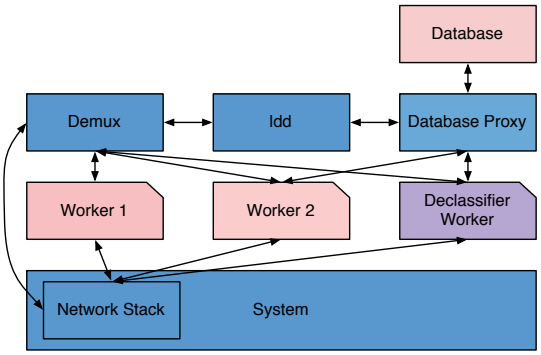

Fig. 4. The Processes of the Asbestos Web Server [49]. Blue, pink and purple represent trusted, untrusted and semi-trusted processes respectively.

the public, it requires the agreement of Alice (and any other tag owner in S_p). If a process has an integrity label I_p such that $a \in I_p$, this process is only allowed to read data with a label I_d if $a \in I_d$. Objects (such as files or directories) are represented as processes with immutable labels.

Each process has a set of capabilities. Two types of capability are associated with a tag t: t^+, which allows a process to add the tag to its labels, and t^-, giving the privilege to remove it. For secrecy, t^+ allows a process to receive data associated with t and protect data sent with that tag, while t^- allows processes to declassify data. Regarding integrity, t^+ enables a process to endorse itself as high-t-integrity, while t^- allows it to receive low-t-integrity data.

Although Flume has the advantage of supporting a standard OS and programs, this also means that it is vulnerable to any security flaws inherent in those systems. It cannot establish the same trusted computing base as a dedicated DIFC OS.

Laminar (2009) [96] takes a similar approach to Flume at the OS level, adding a security module to standard Linux. In addition, the JVM is customised to support thread isolation and object protection. Airavat [97] is an example of cloud services running above Laminar; it makes the Hadoop file system and MapReduce DIFC aware.

2) Asbestos: Asbestos [49], [93] is an OS prototype that provides labelling and isolation in order to bound exploitable software flaws. Each process P has a $send$ label P_s and a $receive$ label P_r. The $send$ label represents the current contamination of the data being sent, and the $receive$ label the contamination a process is willing to accept from another. Labels are composed of a default access/privilege level and a number of handles (h), each handle comprising a unique identifier and an access level. Access levels can take values 0, 1 (untainted), 2 (partially tainted), 3 (tainted) and $\star$ which indicates the declassification privilege.

Process P is able to send data to Q only if $P_s \sqsubseteq Q_r$, that is, iff $\forall h \ P_s(h) \leq Q_r(h)$. When the process Q receives data from P its send label is changed such that $Q_s \leftarrow Q_s \sqcup P_s$. $\sqcup$ is the least upper bound operator, i.e., $(Q_s \sqcup P_s)(h) = \max(Q_s(h), P_s(h))$. Similarly when a process reads a file from the trusted file system or from the network it is contaminated by the file and can only send a message to another process which accepts this level of contamination. For a process to receive data with a certain label the system must raise the $receive$ label of this process to the appropriate level.

A process can create a *decentralised compartment* and can create other processes with labels such that they can only reveal information to other processes in that compartment. The process that created the compartment can declassify information in that compartment or delegate the right to do so. This enables the implementation of POLP [16].

A web server was built on top of Asbestos to illustrate its ability to enforce DIFC in highly concurrent applications [49]. Its architecture, shown in Fig. 4, is relevant to cloud deployment, as we will discuss in §V-E. The server is composed of a number of dynamic untrusted *workers*, each with a particular task such as logging, retrieving data, or changing passwords. The *demux*, a trusted process, analyses incoming requests and directs them to the relevant worker. The *idd* is a trusted process that verifies user credentials and system state. Each worker, maintains an isolated—through labels and event processes—cache for each user. A trusted proxy manages database access, associating each table row with a particular user. The *declassifier worker* is semi-trusted; if compromised it could leak the compromised user data, but could not access uncompromised data.

HiStar (2006) [50] extended Asbestos to avoid the possibility of covert channels caused by the implicit modification of taint levels. HiStar also provides a user-space level library emulating a Unix-like OS interface.

3) DStar: DStar [94] enables IFC in distributed systems by leveraging the labelling mechanisms of IFC-compliant operating systems. To facilitate tracking across machines, DStar involves translating a local machine's security labels into a set of globally-meaningful labels. Each machine maintains an *exporter* that tracks the labels of the local processes, and enforces data flow policy when interacting (via messages) with other machines. DStar can operate over a range of OSs, including HiStar, Flume and (trusted instances of) Linux.

4) Aeolus: Aeolus [95] deploys a common trusted computing base (TCB) on every node of a distributed system. It is based on Asbestos, extending the Asbestos design for distributed communication. Each application thread runs on behalf of a principal. Applications run on top of the trusted base, which filters I/O, inter-thread and external communications, enforcing the policy associated with the data.

C. IFC at the Middleware Level

Integrating IFC mechanisms within middleware means that policy can be enforced against all applications' interactions using that middleware. IFC security can be exposed as an explicit service provided by the middleware, and/or the middleware may use IFC internally to act as a safety-net to mitigate against erroneous or insecure application behaviour. As the role of middleware is to mediate between applications and lower-level system (OS/Network) concerns, it is more amenable to managing distributed applications, as heterogeneous OS support is often a development goal.

1) DEFCON*:* In previous work, we explored data-centric security mechanisms in several domains. The DEFCON system [98] adds strong object isolation to Java without impacting the efficiency of object sharing. In particular we introduced the notion of *Decentralised Event Flow Control* (DEFC), which focuses on the IFC requirements of event-based systems. Our

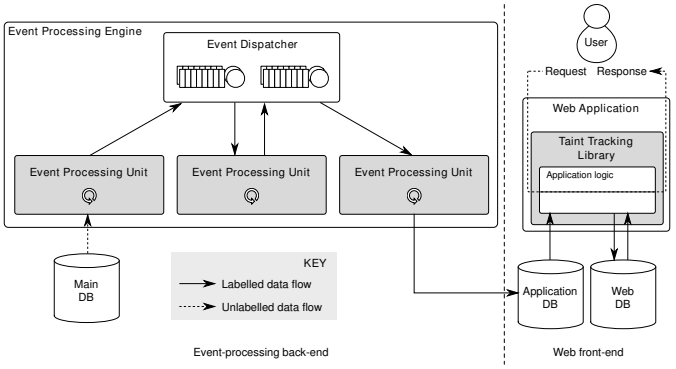

Fig. 5. The SafeWeb Architecture.

DEFC model uses labels similar to Flume, but applies them in parallel to different parts of messages. These multipart messages can then be passed between isolates using software developed using an event-driven paradigm.

DEFCON is implemented in Java, and runs on an unmodified JVM. It provides efficient inter-isolate communication using a combination of static and runtime techniques. As a middleware, DEFCON provides an API that applications can be developed against. However, to strongly enforce isolation, the system goes further than providing middleware: additional runtime data flow containment instrumentation can be installed using AspectJ, an aspect-oriented weaver. A static analysis phase ensures that isolates cannot communicate using channels such as the many thousands of `static` variables maintained by the Java runtime environment.

We show that DEFCON can be used to provide a secure, low-latency, centralised event processing middleware for financial trading applications.

Our DEFC approach generalises beyond Java: a similar approach to the message-based information flow control in DEFCON was implemented in the Erlang language [74].

2) SafeWeb: SafeWeb [59] is a middleware that aims to mitigate against policy violations in multi-tier web applications. It uses IFC to track data flows through all tiers of the web application infrastructure, in order to ensure end-to-end data confidentiality and integrity. SafeWeb consists of an *event processing backend*, that deals directly with the processing of confidential data, and a *web frontend* that manages application (or client) requests. By decoupling web requests from the processing of data, implementation issues in the logic handling web requests cannot result in inappropriate release of confidential data. The architecture is illustrated in Fig. 5.

The event processing backend is responsible for directly handling the data of the confidential datastore. The backend encapsulates data in events, which are associated with 'sticky' labels to enable tracking throughout the system. *Event Processing Units* (EPUs) generate, process and filter events in accordance with the system's functional requirements, and are responsible for labelling the events they produce. The *event dispatcher* acts as a broker to distribute events to those EPUs that are willing and able (by comparing labels and privileges) to process them. EPUs can generate *result* events, which along with their labels, are exported to a database in the web frontend. In this way, the flow of information from the confidential (main) database to web requests is indirect, and unidirectional. The *Event Processing Engine* manages the overall process by checking and tracking labels, and restricting access to the environment by managing the privileges assigned to EPUs.

The web frontend operates to serve requests by querying the local datastore that holds the data, and associated labels, as a result of backend processing. The taint checker uses these labels to effect confidentiality at the frontend, where a requesting client may only access a variable's contents if they hold the privileges allowing access to the associated label(s). This approach means that data flow policy is enforced in a manner transparent to web applications.

D. Library-Level IFC

Library-level IFC systems track explicit flow of information within a web application by extending a given language with IFC related features. These features include the ability to define "sandboxes" for containing labeled data, to associate labels with variables and define policies to be applied to labeled data.

The purpose of these systems is to prevent SQL injection, cross-site scripting and to protect users' sensitive information.

1) Resin: Resin is a runtime taint tracking system for PHP web applications, which improves server-side security and can be used to guarantee user-data privacy [58]. Resin provides mechanisms to help programmers implement IFC assertions. *Filter objects* define data flow boundaries that can be interposed at I/O interfaces or on a function call interface. *Sticky policies* can associate data with a policy that application programmers have defined themselves. The programmers rely on the Resin label tracking system to propagate those policies throughout the application (however the programmer must pay particular attention to implicit flow, see §IV-A).

Programmers are able to write filters and policies in their usual application language (PHP) which will likely ease the adoption of such techniques. Filters enable a programmer to, for example, prevent user passwords being leaked out of the application, and by using appropriate policy, programmers can ensure that certain data is automatically anonymised or sanitised when leaving the application.

2) PHP Aspis: The PHP Aspis tool [54] uses a simpler approach than Resin: Aspis marks all user generated data with a "taint" and propagates this taint across the application. Some critical methods are modified (such as *echo* or *print*) to present specific behaviour when faced with tainted data. This is done by performing code-to-code translation of the source code of executed software. Thus, PHP Aspis does not require a modified language runtime, and does not require that software be written to be aware of PHP Aspis. The tradeoff against Resin is a reduction in expressiveness and control from the point-of-view of the programmer.

3) Bello et al.: Bello et al. [57] use a Python taint library [99] that extends the Python language to support taint tracking without the need to modify the interpreter. The library specifically targets applications running on the Google App Engine (a PaaS cloud). Applications can specify policy that will be applied to tainted data, in a similar manner to Resin.

TABLE I
COMPARISON OF DIFFERENT CLOUD-RELEVANT FACETS OF EXISTING DIFC SYSTEMS

Implementation	Tracking Granularity	Interface	Distributed?	Re-engineering Effort	Label expressiveness
RIFLE [91]	Memory page	Hardware	No	Large	Confidentiality
Suh et al. [92]	Memory page	Hardware	No	Very Large	Taint
Flume [51]	Process	Linux-Like	No	Large	Integrity & Confidentiality
Laminar [96]	Thread, object	Linux-like	No	Medium	Integrity & Confidentiality
Asbestos [49], [93]	Process	Custom	No	Large	Integrity & Confidentiality
HiStar [50]	Process	Linux-Like	No	Large	Integrity & Confidentiality
DStar [94]	Message & Process	Linux-Like	Yes	Large	Integrity & Confidentiality
Aeolus [95]	Process	Custom	Yes	Large	Integrity & Confidentiality
DEFCON [98]	Message	Custom	Yes	Large	Integrity & Confidentiality
SafeWeb [59]	Message	Custom	Yes	Large	Integrity & Confidentiality
Resin [58]	Variable	Library	N/A	Medium	Programmer-defined
PHP Aspis [54]	Variable	Library	N/A	Low	Taint
Bello et al. [57]	Character level	Library	N/A	Low	Taint

However, their IFC only supports single-bit taint tracking, rather than more complex labels. One of the advances of this work over the two preceding systems is that it provides a library that modifies the interface with persistent storage to allow the persistence of taint information.

The work of Bello et al. is not the only project aiming to support persistence of taint information: DBTaint provides database interface libraries in several languages to propagate character-level taint into the database [56]; the work of Pasquier et al. [100] propagates more complex labels such as those described in Resin, but at the database row level.

E. Potential for IFC Provision within IaaS, PaaS and SaaS

Table I summarises the features of the above IFC implementations with cloud deployment in mind. The unit of isolation is implied by tracking granularity e.g. process, thread, object. Hardware-supported IFC is considered out of scope for our study, since it will only be feasible within real-world cloud deployment when integrated into commodity CPUs. It is not inconceivable that this will happen soon: hardware virtualisation support appeared very rapidly, and prototypes such as the Capability Hardware Enhanced RISC Instructions platform (CHERI) [101] have demonstrated incremental integration of hardware-supported isolation.

We explore how the different schemes' IFC designs relate to the three main types of cloud provider:

IaaS: The software running on VMs provided by IaaS hosts can usually ignore their own virtualisation. Since network connectivity is provided between VMs, any fine-grained distributed IFC implementation, such as HiStar, can be run by a tenant over IaaS infrastructure agnostic to its hosting. However in this case, the cloud host must be fully trusted.

More interesting cases would involve the cloud provider participating in the IFC, and the client VMs exposing labels. Given that the security of the virtualisation hypervisor is critical, it is unlikely that IaaS providers would modify it directly, but IFC at the network level might allow the cloud provider to give some form of guarantee on the flow exchanged between different instances from the same tenant or between instances belonging to different tenants. The IFC software would still need to be a distributed variant. This would be a logical extension of the network controls provided by IaaS operators, e.g., the Security Groups provided within services such as Amazon EC2. Currently, these only effect firewall configuration, but they indicate an appropriate place at which to manage inter-VM IFC.

Most distributed IFC systems would require a significant reengineering effort to use. This would be at odds with IaaS allowing clients to offload computing to IaaS VMs with a view to minimising their management effort. While IaaS provides many opportunities for IFC use, it does not present one clear route for such usage.

PaaS: We believe this is the most promising cloud architecture type in which to integrate IFC support. Applications typically need to be modified, at least in part, to use PaaS APIs, so their developers can consider exposing labelling semantics when doing this redevelopment. PaaS APIs usually provide services to facilitate storage and communication, and thus the PaaS provider is well positioned to intercept and manage requests that would need IFC checks performed on them.

Process, thread and message-level IFC tracking granularity are likely to work effectively. For example, Laminar provides an IFC-enabled OS that supports process isolation, and a modified Java virtual machine that guarantees thread isolation runs above the OS. It therefore become possible for a cloud provider to allow tenants to run Java applications in an IFC-aware environment. The PaaS provider can manipulate the JVM container to effect elasticity based on tenants' needs.

The components within PaaS designs—such as the *cartridges* that execute on RedHat OpenShift *gears*—are likely to match parts of an application that should run with different privileges. For example, the number of gears in a typical web application is likely to correlate with the number of components in the Asbestos web server illustrated in Fig. 4.

The PaaS provider will need to isolate tenants from each other. However the IFC mechanisms that the provider uses to achieve this isolation can themselves be provided for tenants to use to isolate their own clients' concerns. This would facilitate collaboration between multiple tenants or their clients with controlled information disclosure.

The PaaS provider needs to see the components that make up an application, and understand how they interact. This type of specification may help to reduce the effort required to integrate security into an application, as the security concerns may be able to be woven into existing code by a specialist. This is seen in the IFC schemes implemented using Aspect

Oriented Programming (AOP) weavers, such as AspectJ for Java or Aquarium for Ruby.

SaaS: Clients are often unaware of how SaaS providers' software works, and do not depend on knowing this. Thus, SaaS providers can use any IFC system that meets their needs, including systems such as Flume OS that would require significant reengineering effort to deploy.

An interesting emerging avenue for IFC relates to clients' interactions with the SaaS provider (this also holds, to a lesser degree, with providers of the other cloud types). Much of our discussion assumes that the cloud provider is trusted not to leak information. However, increasingly, services are expected to interconnect. Just as technologies such as OpenID and OAuth have been developed to effect distributed authentication and authorisation, SaaS applications may promote development of external representations of IFC labelling, that facilitates the interoperation of IFC-enriched cloud services.

VI. Conclusions and Future Work

We believe that DIFC is most appropriately integrated into a PaaS cloud model—which can be tested by augmenting existing open source implementations such as VMware CloudFoundry[9] and Red Hat OpenShift.[10]

We have discussed how DIFC has been used to protect user data integrity and secrecy. In order to apply these techniques to a cloud environment a number of challenges need to be overcome. These include: selecting the most appropriate DIFC model; policy specification, translation, and enforcement; audit logging to demonstrate compliance with legislation and for digital forensics. DIFC should not impose an unacceptable performance overhead and it is important that application developers using cloud-provided IFC are aware of the trust assumptions inherent in the IFC provision. We plan to address these challenges in our future work.

Security concerns are a major disincentive for use of the cloud, particularly for companies responsible for sensitive data. We believe that augmenting existing approaches to cloud security with DIFC is a promising way forward.

[9]http://www.cloudfoundry.org/

[10]https://github.com/openshift/

References

[1] D. Denning, *Cryptography and Data Security*. Addison-Wesley Longman, 1982.

[2] Biba, "Integrity considerations for secure computer systems," MITRE Co., technical report ESD-TR 76-372, 1977.

[3] R. Wu, G.-J. Ahn, H. Hu, and M. Singhal, "Information flow control in cloud computing," in *CollaborateCom*, 2010.

[4] H. Hacigümüş, B. Iyer, *et al.*, "Executing SQL over encrypted data in the database-service-provider model," in *Proc. 2002 ACM SIGMOD*, pp. 216–227.

[5] J. Bacon, D. Evans, *et al.*, "Big ideas paper: enforcing end-to-end application security in the cloud," in *2010 ACM/IFIP Middleware*.

[6] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011. Available: http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf

[7] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[8] P. Barham, B. Dragovic, *et al.*, "Xen and the art of virtualization," in *2003 ACM SOSP*.

[9] T. Ristenpart, E. Tromer, *et al.*, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proc. 2009 ACM CCS*, pp. 199–212.

[10] Y. Zhang, A. Juels, *et al.*, "Cross-VM side channels and their use to extract private keys," in *Proc. 2012 ACM CCS*, pp. 305–316.

[11] A. Ganjali and D. Lie, "Auditing cloud management using information flow tracking," in *Proc. 2012 ACM STC*, pp. 79–84.

[12] D. Leinenbach and T. Santen, "Verifying the Microsoft Hyper-V hypervisor with VCC," in *FM 2009: Formal Methods*. Springer LNCS 5850, 2009, pp. 806–809.

[13] M. Dalton, C. Kozyrakis, and N. Zeldovich, "Nemesis: preventing authentication & access control vulnerabilities in web applications," in *2009 USENIX Security Symposium*.

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

[15] A. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM TOSEM*, vol. 9, no. 4, pp. 410–442, 2000.

[16] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[17] C. S. Alliance, "Security guidance for critical areas of focus in cloud computing," 2011.

[18] J. A. Hall and S. L. Liedtka, "The Sarbanes-Oxley Act: implications for large-scale IT outsourcing," *CACM*, vol. 50, no. 3, pp. 95–100, 2007.

[19] R. T. Mercuri, "The HIPAA-potamus in health care data security," *CACM*, vol. 47, no. 7, pp. 25–28, 2004.

[20] V. J. Winkler, *Securing the Cloud: Cloud Computer Security Techniques and Tactics*. Elsevier, 2011.

[21] CNIL, "Summary of responses to the public consultation on Cloud computing," 2012.

[22] "EU: General data protection regulation," 2012.

[23] A. G. Araiza, "Electronic discovery in the cloud," *Duke Law & Tech. Rev.*, 2011.

[24] S. Biggs and S. Vidalis, "Cloud computing: the impact on digital forensic investigations," in *Proc. 2009 ICITST*, pp. 1–6.

[25] M. Armbrust, A. Fox, *et al.*, "Above the clouds: a Berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

[26] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. 1982 IEEE SOSP*, pp. 11–20.

[27] S. Bleikertz, A. Kurmus, *et al.*, "Secure cloud maintenance: protecting workloads against insider attacks," in *Proc. 2012 ACM ASIACCS*, pp. 83–84.

[28] D. Suciu, "SQL on an encrypted database: technical perspective," *CACM*, vol. 55, no. 9, pp. 102–102, 2012.

[29] D. Liu and S. Wang, "Query encrypted databases practically," in *Proc. 2012 ACM CCS*, pp. 1049–1051.

[30] C. Cadar, D. Dunbar, and E. Dawson, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *2008 USENIX OSDI*.

[31] C. Cadar, V. Ganesh, *et al.*, "EXE: automatically generating inputs of death," *ACM TISSEC*, vol. 12, no. 2, pp. 1–38, 2008.

[32] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *2006 IEEE SOSP*.

[33] R. J. Feiertag, "A technique for proving specifications are multilevel secure," SRI International, CSL, Tech. Rep. CSL-109, 1980.

[34] J. McHugh and D. I. Good, "An information flow tool for Gypsy," in *Proc. 1985 IEEE SOSP*, pp. 46–48.

[35] J. McHugh, "An information flow tool for Gypsy," in *Proc. 2001 ACM ACSAC*, pp. 191–201.

[36] P. Saxena, D. Akhawe, *et al.*, "A symbolic execution framework for JavaScript," in *2010 IEEE SOSP*.

[37] V. Simonet, "FlowCaml in a Nutshell," in *2003 APPSEM-II*.

[38] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, 2003.

[39] O. Tripp, M. Pistoia, *et al.*, "TAJ: effective taint analysis of web applications," in *2009 ACM PLDI*.

[40] S. Bandhakavi, S. King, *et al.*, "VEX: vetting browser extensions for security vulnerabilities," in *2010 Usenix Security Symposium*.

[41] Y.-W. Huang, F. Yu, *et al.*, "Securing web application code by static analysis and runtime protection," in *2004 ACM WWW*.

[42] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2007.

[43] P. Orbaek and J. Palsberg, "Trust in the λ-calculus," *J. Funct. Program.*, vol. 7, no. 6, pp. 557–591, 1997.

[44] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comput. Secur.*, vol. 4, no. 2-3, pp. 167–187, 1996.

[45] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *1997 ACM SOSP*.

[46] S. Zdancewic and A. C. Myers, "Secure information flow and CPS," in *ESOP*. Springer, 2001, pp. 46–61.

[47] D. Volpano and G. Smith, "A type-based approach to program security," in *TAPSOFT*. Springer, 1997.

[48] A. C. Myers, "JFlow: practical mostly-static information flow control," in *1999 ACM POPL*.

[49] P. Efstathopoulos, M. Krohn, *et al.*, "Labels and event processes in the Asbestos operating system," in *Proc. 2005 ACM SOSP*, pp. 17–30.

[50] N. Zeldovich, Boyd-Wickizer, *et al.*, "Making information flow explicit in HiStar," in *Proc. 2006 Usenix OSDI*, pp. 19–19.

[51] M. Krohn, A. Yip, *et al.*, "Information flow control for standard OS abstractions," in *2007 ACM SOSP*.

[52] S. Chong, K. Vikram, and A. Myers, "SIF: enforcing confidentiality and integrity in web applications," in *2007 Usenix Security Symposium*.

[53] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE SOSP*.

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

[55] E. Chin and D. Wagner, "Efficient character-level taint tracking for Java," in *Proc. 2009 ACM SWS*, pp. 3–12.

[56] B. Davis and H. Chen, "DBTaint: cross-application information flow tracking via databases," in *2010 USENIX WebApps*.

[57] L. Bello and A. Russo, "Towards a taint mode for cloud computing web applications," in *Proc. 2012 ACM PLAS*, pp. 7:1–7:12.

[58] A. Yip, X. Wang, *et al.*, "Improving application security with data flow assertions," in *Proc. 2009 ACM SOSP*, pp. 291–304.

[59] P. Hosek, M. Migliavacca, *et al.*, "SafeWeb: a middleware for securing Ruby-based web applications," in *2011 ACM/IFIP Middleware*.

[60] W. Enck, P. Gilbert, *et al.*, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 2010 Usenix OSDI*, pp. 1–6.

[61] V. P. Kemerlis, G. Portokalidis, *et al.*, "libdft: practical dynamic data flow tracking for commodity systems," in *Proc. 2012 ACM SIGPLAN/SIGOPS8: VEE*, pp. 121–132.

[62] E. Bosman, A. Slowinska, and H. Bos, "Minemu: the world's fastest taint tracker," in *RAID*. Springer, 2011, pp. 1–20.

[63] F. Qin, C. Wang, *et al.*, "Lift: a low-overhead practical information flow tracking system for detecting security attacks," in *Proc. 2006 IEEE/ACM MICRO-39*, pp. 135–148.

[64] U. Bayer, P. M. Comparetti, *et al.*, "Scalable, behavior-based malware clustering," in *NDSS*. Internet Society, 2009.

[65] U. Bayer, A. Moser, *et al.*, "Dynamic analysis of malicious code," *J. Computer Virology*, vol. 2, pp. 67–77, 2006.

[66] H. Yin, D. Song, *et al.*, "Panorama: capturing system-wide information flow for malware detection and analysis," in *2007 ACM CCS*.

[67] K. Singh, S. Bhola, and W. Lee, "xBook: redesigning privacy control in social networking platforms," in *2009 USENIX Security Symposium*.

[68] M. Krohn, P. Efstathopoulos, *et al.*, "Make least privilege a right (not a privilege)," in *2005 USENIX HotOS*.

[69] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks," in *2006 ACM EuroSys*.

[70] F. Bellard, "QEMU, a fast and portable dynamic translator," in *2005 USENIX ATC*.

[71] J. M. McCune, B. Parno, *et al.*, "Flicker: an execution infrastructure for TCB minimization," in *2008 ACM EuroSys*.

[72] B. Parno, J. McCune, *et al.*, "CLAMP: practical prevention of large-scale data leaks," in *2009 IEEE Symp. on Sec. & Priv.*.

[73] B. D. Payne, R. Sailer, *et al.*, "A layered approach to simplified access control in virtualized systems," *ACM SIGOPS OSR*, vol. 41, no. 4, pp. 12–19, 2007.

[74] I. Papagiannis, M. Migliavacca, *et al.*, "Enforcing user privacy in web applications using Erlang," in *2010 IEEE SOSP, W2SP Workshop*.

[75] A. Nguyen-Tuong, S. Guarnieri, *et al.*, "Automatically hardening web applications using precise tainting," in *2005 IFIP Intnl. Information Security Conference*.

[76] P. Vogt, F. Nentwich, *et al.*, "Cross site scripting prevention with dynamic data tainting and static analysis," in *2007 Network and Distributed System Security Symposium*.

[77] J. Burket, P. Mutchler, *et al.*, "GuardRails: a data-centric web application security framework," in *2011 USENIX WebApps*.

[78] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *CACM*, vol. 20, no. 7, pp. 504–513, 1977.

[79] T. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *2009 ACM PLAS*.

[80] T. Austin and C. Flanagan, "Permissive dynamic information flow analysis," in *2010 ACM PLAS*.

[81] M. Gyung, S. McCamant, *et al.*, "DTA++: dynamic taint analysis with targeted control-flow propagation," in *Network and Distributed System Security Symposium*. Internet Society, 2011.

[82] US Department of Defense, "Trusted Computer System Evaluation Criteria (Orange Book)," 1983.

[83] A. Sabelfeld and A. Russo, "From dynamic to static and back: riding the roller coaster of information-flow control research," in *Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2009.

[84] A. Russo, Alejandro, and Sabelfeld, "Dynamic vs static flow-sensitive security analysis," in *2010 IEEE CSFS*.

[85] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing- and termination-sensitive secure information flow: exploring a new aproach," in *2011 IEEE SOSP*.

[86] B. W. Lampson, "A note on the confinement problem," *CACM*, vol. 16, no. 10, pp. 613–615, 1973.

[87] S. B. Lipner, "A comment on the confinement problem," *ACM SIGOPS OSR.*, vol. 9, no. 5, pp. 192–196, 1975.

[88] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *1997 IEEE CSFW-10*.

[89] I. Gray, J.W., "Probabilistic interference," in *1990 IEEE SOSP*.

[90] P. Syverson and I. Gray, J.W., "The epistemic representation of information flow security in probabilistic systems," in *Proc. 195 IEEE CSFW-8*, pp. 152–166.

[91] N. Vachharajani, M. Bridges, *et al.*, "RIFLE: an architectural framework for user-centric information-flow security," in *2004 ACM/IEEE MICRO-37*, pp. 243–254.

[92] G. E. Suh, J. W. Lee, *et al.*, "Secure program execution via dynamic information flow tracking," *ACM SIGOPS OSR*, vol. 38, no. 5, pp. 85–96, 2004.

[93] S. Vandebogart, P. Efstathopoulos, *et al.*, "Labels and event processes in the Asbestos Operating System," *ACM TOCS*, vol. 25, no. 4, 2007.

[94] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control," in *2008 USENIX NSDI*, pp. 293–308.

[95] W. Cheng, D. R. K. Ports, *et al.*, "Abstractions for usable information flow control in Aeolus," in *2012 USENIX ATC*.

[96] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: practical fine-grained decentralized information flow control," *SIGPLAN Not.*, vol. 44, no. 6, pp. 63–74, 2009.

[97] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: security and privacy for MapReduce," in *2010 Usenix NSDI*.

[98] M. Migliavacca, I. Papagiannis, *et al.*, "Information flow guarantees in multi-domain distributed applications," in *2010 ACM/IFIP Middleware*.

[99] J. Conti and A. Russo, "A taint mode for Python via a library," in *Inf. Sec. Tech. for Apps.* Springer LNCS 7127, 2012, pp. 210–222.

[100] T. F. J.-M. Pasquier, B. Shand, and J. M. Bacon, "Information flow control for a medical webportal," in *e-Society*. IADIS, 2013.

[101] R. N. M. Watson, P. G. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps, M. Roe, and H. Saidi, "CHERI: a research platform deconflating hardware virtualization and protection," in *Proc. 2012 Runtime Environments, Systems, Layering and Virtualized Environments*.