Technical Report

UCAM-CL-TR-893
ISSN 1476-2986

Number 893

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Towards practical information flow control and audit

Thomas F. J.-M. Pasquier

July 2016

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted January 2016 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Jesus College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

In recent years, pressure from the general public and from policy makers has been for more and better control over personal data in cloud computing environments. Regulations put responsibilities on cloud tenants to ensure that proper measures are effected by their cloud provider. But there is currently no satisfactory mechanism to achieve this, leaving tenants open to potentially costly lawsuits.

Decentralised Information Flow Control (IFC) at system level is a data-centric Mandatory Access Control scheme that guarantees non-interference across security contexts, based on lattices defined by secrecy and integrity properties. Every data flow is continuously monitored to guarantee the enforcement of decentrally specified policies. Applications running above IFC enforcement need not be trusted and can interact. IFC constraints can be used to ensure that proper workflows are followed, as defined by regulations or contracts. For example, to ensure that end users' personal data are anonymised before being disclosed to third parties.

Information captured during IFC enforcement allows a directed graph representing whole-system data exchange to be generated. The coupling of policy enforcement and audit-data capture allows system "noise" to be removed from audit data, and only information relevant to the policies in place to be recorded. It is possible to query these graphs to demonstrate that the system behaved according to regulation. For example, to demonstrate from run-time data that there is no path without anonymisation between an end-user and a third party.

Acknowledgements

First of all, I would like to thank Jean Bacon, my supervisor, who has provided tremendous advice and support during my time at the Computer Laboratory. Without her help I would not have been able to complete and I am forever grateful for this. I also appreciate the assistance and feedback given by Ken Moody.

I had the pleasure to work on the CloudSafetyNet grant and would like to thank the people I collaborated with through this project: Peter Pietzuch, Dan O'Keeffe, Divya Muthukumaran, Brian Shand; particular thanks are due to Jatinder Singh and David Eyers. I would also like to thank the people at the Microsoft Cloud Computing Research Centre: Jon Crowcroft, Christopher Millard, Ian Walden, Kuan Hon, Guido Noto La Diega and Dimitra Kamarinou. Thanks are also due to Julia Powles for her support on some of the legal aspects. I would also like to thank Olivier Hermant for his mentoring and help during my years in France and to this date; his confidence in my ability is the reason I applied to Cambridge.

I would also like to thank the friends I met while in Cambridge. I would like to thank the staff and pupils at the Cambridge Centre for Sixth Form Studies where I had pleasure teaching computing and programming.

My research was funded by the Engineering and Physical Sciences Research Council (EPSRC).

Finally, I would like to thank my parents, sister, and family for their support and encouragement during these years. A special thank-you to my aunt, who made sure I had the financial means to attend this institution and who passed away during my time here.

Publications

Bacon, J., Eyers, D., Pasquier, T., Singh, J., Papagiannis, I., and Pietzuch, P. (2014). Information Flow Control for Secure Cloud Computing. *IEEE Transactions on Network and System Management, SI Cloud Service Management*, 11(1):76–89.

Pasquier, T., Bacon, J., and Eyers, D. (2014a). FlowK: Information Flow Control for the Cloud. In *International Conference on Cloud Computing Technology and Science (CloudCom'14)*. IEEE.

Pasquier, T., Bacon, J., and Shand, B. (2014b). FlowR: Aspect Oriented Programming for Information Flow Control in Ruby. In *International Conference on Modularity*. ACM.

Pasquier, T., Bacon, J., Singh, J., and Eyers, D. (2016a). Data-Centric Access Control for Cloud Computing. In *Symposium on Access Control Models and Technologies*. ACM.

Pasquier, T. and Eyers, D. (2016). Information Flow Audit for Transparency and Compliance in the Handling of Personal Data. In *IC2E International Workshop on Legal and Technical Issues in Cloud Computing (CLaw'16)*. IEEE.

Pasquier, T. and Powles, J. (2015). Expressing and Enforcing Location Requirements in the Cloud using Information Flow Control. In *IC2E International Workshop on Legal and Technical Issues in Cloud Computing (CLaw'15)*. IEEE.

Pasquier, T., Shand, B., and Bacon, J. (2013). Information Flow Control for a Medical Web Portal. In *e-Society 2013*. IADIS.

Pasquier, T., Singh, J., and Bacon, J. (2015a). Clouds of Things need Information Flow Control with Hardware Roots of Trust. In *International Conference on Cloud Computing Technology and Science (CloudCom'15)*. IEEE.

Pasquier, T., Singh, J., and Bacon, J. (2015b). Information Flow Control for Strong Protection with Flexible Sharing in PaaS. In *IC2E, International Workshop on Future of PaaS*. IEEE.

Pasquier, T., Singh, J., Bacon, J., and Eyers, D. (2016b). Information Flow Audit for PaaS clouds. In *International Conference on Cloud Engineering (IC2E)*. IEEE.

Pasquier, T., Singh, J., Bacon, J., and Hermant, O. (2015c). Managing Big Data with Information Flow Control. In *International Conference on Cloud Computing (CLOUD)*. IEEE.

Pasquier, T., Singh, J., Eyers, D., and Bacon, J. (2015d). CamFlow: Managed Data-Sharing for Cloud Services. *IEEE Transactions on Cloud Computing*.

Singh, J., Bacon, J., Crowcroft, J., Madhavapeddy, A., Pasquier, T., Hon, W. K., and Millard, C. (2014). Regional Clouds: Technical Considerations. Technical Report UCAM-CL-TR-863, University of Cambridge.

Singh, J., Pasquier, T., and Bacon, J. (2015a). Securing Tags to Control Information Flows within the Internet of Things. In *International Conference on Recent Advances in Internet of Things (RIoT'15)*. IEEE.

Singh, J., Pasquier, T., Bacon, J., and Eyers, D. (2015b). Integrating Middleware with Information Flow Control. In *International Conference on Cloud Engineering (IC2E)*. IEEE.

Singh, J., Pasquier, T., Bacon, J., Ko, H., and Eyers, D. (2016). Twenty security considerations for cloud-supported Internet of Things. *IEEE Internet of Things Journal*.

Singh, J., Powles, J., Pasquier, T., and Bacon, J. (2015c). Data Flow Management and Compliance in Cloud Computing. *IEEE Cloud Computing Magazine*.

Contents

Chapter 1

Introduction

The research presented in this dissertation is the result of my work as a Research Assistant on the EPSRC grant EP/K011510/1 *CloudSafetyNet: End-to-end application security in the cloud*,[1] a collaboration between Imperial College London's Computing Department and the University of Cambridge Computer Laboratory; and my involvement in the Microsoft Cloud Computing Research Centre,[2] a collaboration between the Centre for Commercial Law Studies, Queen Mary University of London and the University of Cambridge Computer Laboratory.

Cloud computing is a widely adopted paradigm which sees the offloading of storage and computation from self-managed physical infrastructures to virtual infrastructures. A third party, the cloud provider, manages a shared physical infrastructure on top of which the virtual infrastructures are run. Sharing of infrastructure by multiple parties has reduced in a drastic fashion the cost of deploying new services. This made deployment of web services, even at large scale, affordable to small companies with minimal initial capital investment, which in turn has allowed the emergence of innovative web-based services at an unprecedented rate.

We can divide cloud computing commercial offerings into three main categories: 1) Infrastructure as a Service (IaaS), where tenants (the clients of a cloud provider) deploy virtual machines (VM) (e.g. Amazon EC2[3]); 2) Platform as a Service (PaaS), where tenants deploy applications (e.g. Heroku[4]) and 3) Software as a Service (SaaS), where tenants use applications entirely managed by the cloud provider (e.g. Gmail[5]). Fig. 1.1 outlines the extent of the software stack managed by the cloud provider.

Simultaneously, policy makers and the general public have seen an increased awareness of privacy and security concerns in cloud computing, notably fuelled by data misuse [Pa-

[1] `http://www.cl.cam.ac.uk/research/srg/opera/projects/csn/`
[2] `http://www.mccrc.eu/`
[3] `https://aws.amazon.com/ec2/`
[4] `https://www.heroku.com/`
[5] `https://www.google.fr/intx/fr/work/apps/business/products/gmail/`

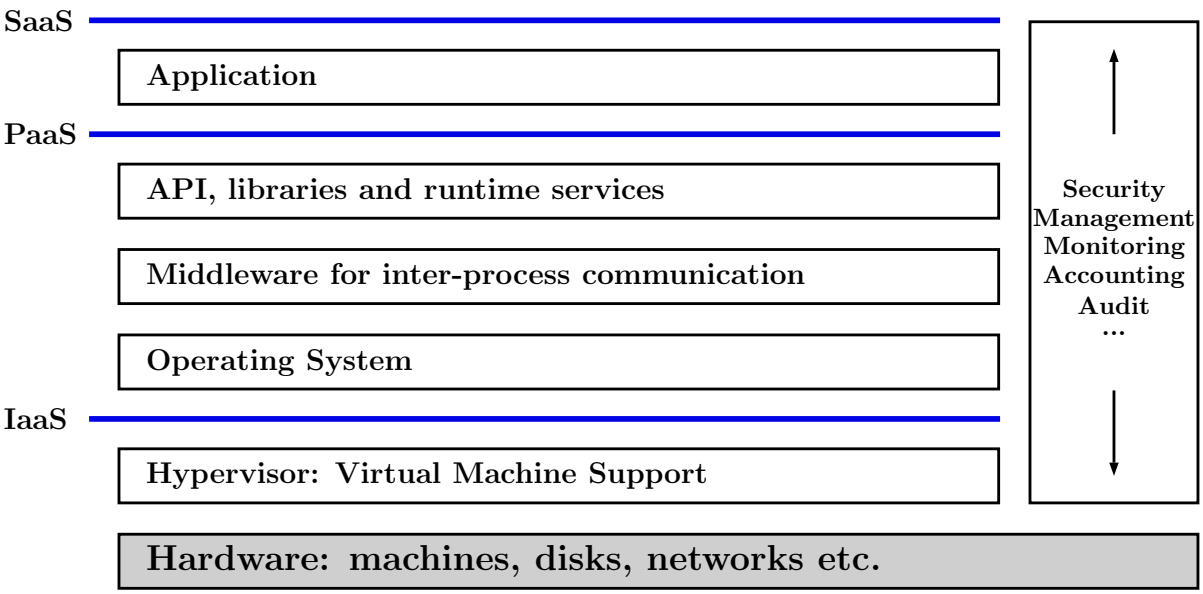

Figure 1.1: Classic cloud service provision architecture. Providers (offer and) manage services below the associated (blue) line, and tenants those above [Pasquier et al. 2014a].

pacharissi and Gibson 2011, Zhang et al. 2010] and more recently by the work of national agencies. These security concerns have led to the emergence of a large body of regulation [Millard 2013, Singh et al. 2015c], ranging from the requirement to obtain consent for a particular data usage, to restrictions on the physical location of storage and computation. These regulations have hindered the adoption of cloud computing in a number of regulated sectors [Bellamy 2013, El-Gazzar 2014] such as healthcare, finance, social services and education.

The reluctance to adopt cloud solutions is partly due to the relative inadequacy of currently available security mechanisms, but more importantly to the lack of technical means to ensure and demonstrate compliance with regulation. This leaves the regulated sectors in a difficult situation. Indeed, cloud benefits: e.g. relatively low cost, flexibility, scalability, low capital investment, focus on core business; need to be weighed against legal risk. A possible solution is single-tenant dedicated infrastructure. However, this negates some of the benefits of cloud computing and introduces a higher cost barrier. This barrier is slowing down innovation in regulated sectors such as healthcare, despite innovations in such sectors potentially benefiting society greatly (e.g. [Hillestad et al. 2005]).

It could be argued that it is necessary to lessen the regulation burden in order to see such innovations emerge. However, it seems unwise to compromise when privacy and basic civil rights are concerned; this appears to be the consensus that has emerged within the European Union [Charlesworth 2000]. Therefore, we believe that technical means must emerge to allow the cloud paradigm to be used , while providing guarantees of compliance with the requirements of laws and regulations. This dissertation presents CamFlow, a solution to enforce and demonstrate compliance with regulation through technical means.

1.1 Context: regulation as information flow constraints

Legal and regulatory considerations for data flows in the cloud revolve around four primary dimensions [Singh et al. 2015c]:

Contractual Obligations: The provision of cloud services establishes a number of contracts between the cloud provider and users, such as privacy policies or service level agreements (SLAs). There is a need for mechanisms to monitor the fulfilment of such contracts, in the same way that tools emerged to monitor Quality of Service (QoS) in communications' SLAs [Duncan and Whittington 2015]. Furthermore, an audit mechanism could help with establishing responsibility, when multiple parties are involved, in case of breach.

Data Protection: The data protection laws of many countries place tenants and providers under strict obligations and responsibilities over the management of personal data. The premise of these laws is that all the usages of information that allow individuals to be identified (directly or indirectly via inference), should be strictly controlled and audited. In theory, non-compliance can be met with severe penalties. In some jurisdictions such laws apply within vertical domains, such as in the US (e.g. medical data's HIPAA regulation,[6] commercial data's FTC Act[7] etc.), while in other jurisdictions, such as in the European Union, personal data fall under a more general schema: the Data Protection Directive 95/46/EC[8] and the new EU regulation currently under negotiation.[9]

Law Enforcement: Pressure is increasing for companies to report to their international clientele when access is demanded by government agencies. Over recent years, many governments have passed legislation to allow government agencies to access information in order to "protect national interest", sometimes through simple administrative procedures, such as the US PATRIOT Act,[10] or the French 2013 *Loi de programmation militaire*.[11] Since November 2015, the state of emergency in France has further decreased judicial oversight.

Regulatory and Common Law Protections: These apply in sensitive domains such as doctor-patient or lawyer-client relations. They may also apply in commercial domains, for example to protect trade secrets or intellectual property.

A lot of these regulatory or contractual requirements can be understood as constraints over the flow of data. For example, some regulation over medical data in the UK could

[6] http://www.hhs.gov/ocr/privacy/

[7] https://www.ftc.gov/site-information/privacy-policy

[8] http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML

[9] http://ec.europa.eu/justice/data-protection/

[10] http://www.justice.gov/archive/ll/highlights.htm

[11] http://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000028338825&categorieLien=id

simply translate into: "medical data must be encrypted before storage".[12] This in technical terms means that storage of medical data should be prevented unless the data has first gone through encryption. Similarly, EU regulation restricts EU companies' storage of personal data to a certain number of geo-locations. Again, in technical terms, this means restricting information flow based on data quality (i.e. personal data) and its origin (i.e. generated within the EU). Legal use-cases – taken from EU regulations or data protection agency recommendations – are explored in Chapter 7.

1.2 Enforcement: Information Flow Control

The early days of the Internet with its simple, client-server architecture are long gone. Most applications are composed of multiple services interacting through APIs (Application Programming Interface). Furthermore, applications may exchange information to deliver services to the end user. The different services and applications may fall under different management regimes and legislatures. However, most of this is unclear for most end users and problematic in certain cases. For example, EU personal data should be stored within the EU or in certain specified Safe Harbours,[13] therefore a complex chain, often undisclosed (or inexplicit), of third-parties may expose a company to legal pursuit as it failed to implement proper handling of personal data. What is needed is a security mechanism that applies across application and service boundaries. This mechanism needs to be data-centric in order to align with the body of regulation which tends to focus on data.

1.2.1 Scope of commonly deployed mechanisms

Access Control mechanisms currently in place in the cloud fail to meet the requirements of a highly regulated sector. These mechanisms only address access to data by principals, and do not implement any further control once the data has been accessed. Further, these mechanisms are point to point, principal-centric, application dependent and heterogeneous in their implementation. In practice, this means that as a piece of data flows through a complex multi-component system, it may fall under different access control regimes, with varying granularity (e.g. a front-end application authenticating individual users versus a back-end database authenticating entire applications for whole-table access).

Encryption mechanisms partially solve the issue of data safety while in persistent storage.

[12] Health & Social Care Information Centre – Information Governance – `http://systems.hscic.gov.uk/infogov`

[13] The European Union is currently reviewing the notion of Safe Harbour following the Maximilian Schrems versus Data Protection Commissioner case of 23/09/215. `http://curia.europa.eu/juris/document/document.jsf?text=&docid=157862&pageIndex=0&doclang=EN&mode=req&dir=&occ=first&part=1&cid=191188`.

Access to clear-text data is only possible for owners of the associated cryptographic key. But regarding the *processing* of data, until homomorphic encryption becomes practical [Naehrig et al. 2011], data need to be processed in clear text. Key holders who perform processing are then entirely entrusted with proper usage and non-disclosure of clear-text data.

Containment. Given the shared nature of the cloud, a key focus has been on *isolating* tenants (data and processing) in order to prevent information leakage. The goal of isolation is to segregate tenants, protecting their data and computation, and to limit a tenant's (direct) knowledge of others. A common approach involves containing tenants by allocating them their own *virtual machines* (VMs), each VM maintaining its own OS. More recently, *containers* [Soltesz et al. 2007] have enabled strong isolation of tenants over a shared OS. Though strong isolation of unrelated tenants is of clear importance, many applications and services will require data sharing across and outside isolation boundaries.

Augmenting existing mechanisms. While the above mechanisms contribute towards data security, they are insufficient to meet the complex requirements of today's software systems entirely. None of them can control the proper usage of data once "out of the hands" of the data owner, i.e. beyond their direct control. Each mechanism has its place, and we propose to complement them with a means to express and enforce data usage requirements throughout a multi-component system.

1.2.2 Information Flow Control to enforce policy-compliant data usage

Such an additional mechanism is **Information Flow Control** (IFC). IFC is a data-centric mandatory access control mechanism that guarantees non-interference across security contexts. IFC is continuously enforced at every information exchange. Some IFC systems have operated at the programming language level, in which case, IFC is enforced on assignments to variables. Our work enforces IFC at the Operating System (OS) level, in which case every system call is checked for IFC conformance, e.g. reading or writing a file. As discussed above, the body of regulation and law on cloud computing tends to be data-centric and is thus ideally complemented by IFC.

IFC was first introduced in [Denning 1976]. Entities in a system are associated with a *secrecy* and an *integrity* level, depending on the sensitivity of the information they handle. Over the secrecy dimension, the *no read up, no write down* [Bell and LaPadula 1973] principle is guaranteed, while over the integrity dimension, the *no read down, no write up* [Biba 1977] principle is guaranteed. For example, a top-secret entity can receive top-secret, secret and public information flow, while a public entity can only handle public information flow.

IFC was later expanded by Myers [Myers and Liskov 1998] as Decentralised Information

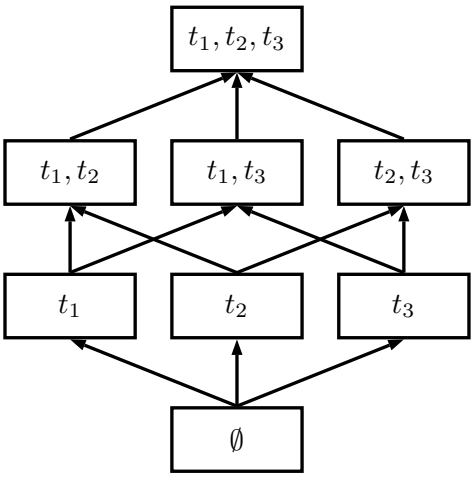

Figure 1.2: DIFC lattice determining allowed information flows.

Flow Control (DIFC). DIFC removes the need for a centrally/globally defined hierarchy and instead adopts a decentralised approach to managing labels. Labels are composed of tags, each representing a particular secrecy concern (e.g. medical-data) or a particular integrity concern (e.g. sanitised-data). The labels compose a lattice that defines allowed information flows, as illustrated in Fig. 1.2. Any entity in the system can define its own tags independently, in order to represent its own security concerns. We adopt this approach in this research, as applied in a variety of contexts from social media [Singh et al. 2009] to embedded-computing for BMW cars [Bouard et al. 2013].

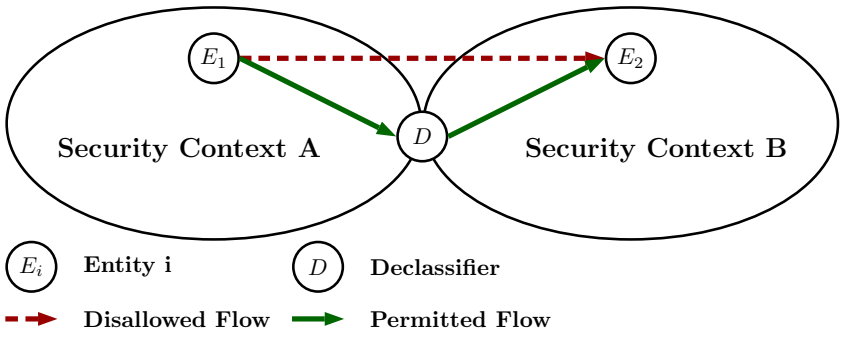

Figure 1.3: Transferring data across security contexts through a Declassifier (or Endorser, not shown).

It has been demonstrated that IFC helps in reducing the Trusted Computing Base (TCB) [Krohn et al. 2007, Myers and Liskov 2000, Vandebogart et al. 2007, Zeldovich et al. 2006]. When IFC is in place, there is no need to trust applications running above the level of IFC enforcement for proper data usage to be guaranteed [Kumar and Shyamasundar 2014]. For example, in the case of enforcement at the OS level, applications running on the OS need not be trusted [Krohn et al. 2007]. A tamper-proof mechanism enforcing

IFC policy (be it the compiler, the kernel etc.) controls every data exchange between entities, guaranteeing that data do not leave their designated security contexts. Privileges to transfer data across security contexts are limited to a well-defined number of trusted entities called declassifiers and endorsers, as illustrated in Fig. 1.3. The TCB is limited to the enforcement mechanism and to a lesser extent, to the declassifier/endorser, to transfer data from and to well-defined security contexts.

In this dissertation the assumption is that the cloud provider that manages the underlying infrastructure is the provider of this enforcement mechanism. The cloud provider is bound through regulation or contract to implement proper security measures [Millard 2013]. It is also in the best *economic* interest of cloud service providers to provide security guarantees; if they are not able to meet some standard, their customers will move to competitors. Of great significance is that a large new market could be opened up for regulated sector organisations, if security mechanisms were provided that allow cloud providers to demonstrate compliance with regulation. To achieve this, the regulated sector needs to trust providers to enforce and demonstrate compliance with contractual duties. In practice, the widespread adoption of cloud services is a strong indication that clients are willing to trust cloud service providers. For general usage, we believe that trust in the few large cloud providers is more justified than in the multitude of applications that are offered above cloud services.

1.2.3 Application level policy

IFC can be seen as a mechanism for enforcing policy; the labels associated with entities represent application policies. IFC is a simple, low-level mechanism. At a higher level, sticky policies have been proposed for a similar purpose, to achieve end-to-end control over data.

The *sticky policies* approach was introduced by Casassa-Mont et al. [Casassa-Mont et al. 2003, Pearson and Casassa-Mont 2011] from HP Laboratories, UK, in the EnCore project [Casassa-Mont et al. 2012]. Other projects have continued to work on this approach [Bandhakavi et al. 2006, Chadwick and Lievens 2008, Garcia et al. 2014]. Here, data are encrypted along with a list of policies to be enforced on that data. In order to obtain the decryption key from a Trusted Authority (TA), a party must agree to enforce the policies associated with the data. This agreement may be considered as part of forming a contractual link between the data owner and the service provider.

The policies that can be encoded are much more complex than the simple secrecy and integrity constraints of IFC. However, due to their complexity, they are only enforced at application and/or organisation boundaries, as otherwise the enforcement cost would be prohibitive. IFC on the other hand, as we demonstrate throughout this dissertation, can be enforced continuously at a reasonable cost. The implementation of complex policy

through IFC can be achieved through the assignment of proper secrecy and integrity labels to entities, in particular declassifiers and endorsers.

The sticky policy approach provides no means to ensure the proper usage of data once decrypted. A malicious service could be sued and black-listed by the TA, but only if and when a breach of agreement is detected. The system builds upon the trust established between the data owner, the TAs and services that manipulate the data. The approach proposed in this dissertation builds only upon the trust between the data owner and the cloud provider. The end user manages and expresses a desired policy, and the cloud provider is responsible for IFC enforcement. Services and applications running on top of the cloud provider platform need not be trusted. We believe this to be a great improvement to the overall trustworthiness of the system.

1.3 Audit: tracking information flow

We have seen how some regulations can be expressed as constraints over the flow of information within a system, and how such constraints can be enforced by augmenting access control, encryption and containment with IFC. However, we believe that this is not yet sufficient to meet the needs of regulated sectors. There is a need for transparency when demonstrating compliance, which can be achieved by capturing and recording information flows. The record of such flows may allow us to demonstrate consistent application of regulations and policies, and understand the chain of events and the causes leading to a data leakage. In this way, we may better understand the behaviour of the system.

Current application-centric logging mechanisms fail to provide such transparency. Aggregation mechanisms have been provided to process logs from several applications and several layers of the software stack. Such logs tend to focus on a specific aspect – for example, relating to a web server or a database – rather than providing a general overview of the system behaviour. However, none can give satisfactory results in a cloud computing context, as the logged data are heterogeneous and often fail to capture information on specific items of data [Ko et al. 2011]. In order to demonstrate compliance with data regulation, it would appear that a data-centric approach to logging is required. We believe that such a mechanism could emerge from provenance research.

1.3.1 Provenance systems

Data provenance (sometimes called *lineage*), can be understood as a means to describe *where, when, how* and *by whom* data was generated or manipulated. Provenance data are generally used for: verification of data quality, generation of replication recipes, attribution of ownership, understanding of context, determination of resource usage, and analysis of

error in data generation [Simmhan et al. 2005]. Some of these uses of provenance can be directly associated with legal requirements.

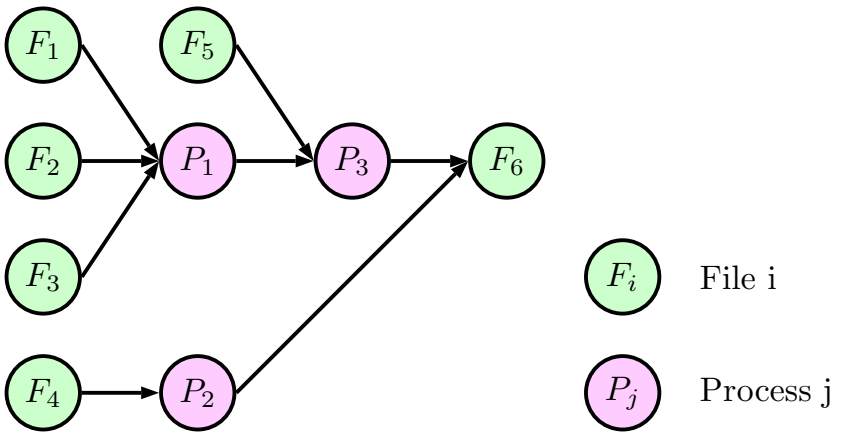

Figure 1.4: Representing the relationship between information flows and entities as a directed graph.

Provenance metadata is intended to represent how the data contained in a given entity was generated and how it relates to other entities in the system. This metadata is generally represented and analysed as a directed graph, as illustrated in Fig. 1.4.

System provenance was traditionally a focus of databases or data storage systems. For example, in [Muniswamy-Reddy et al. 2006] the authors record interactions between files and processes in the Linux OS. However, this fails to capture a certain number of important interactions in a complex real-life system. In order to understand how information is manipulated or generated, it is not sufficient only to monitor the interaction between a process and persistent data, but also requires the additional collection of data that captures the interactions of processes, inter-process communication mechanisms and the OS kernel via system calls. This has led to the emergence of whole-system provenance [Pohly et al. 2012].

1.3.2 From Provenance to Information Flow Audit

Whole-system provenance [Bates et al. 2015b, Pohly et al. 2012] and IFC kernel-level enforcement, as presented in this dissertation, use the same underlying mechanism to track exchanges of information between processes and other kernel objects. Furthermore, key concepts, vocabulary and understanding of the system are extremely similar. However, no link has been clearly established between IFC and provenance data capture until recently. In [Akoush et al. 2014] the authors used provenance data to verify *a posteriori* the respect of IFC policies. In Chapter 5 and [Pasquier et al. 2016b], we present the capture of provenance data during IFC enforcement.

During the enforcement of IFC, the data logged about the system behaviour allows a provenance graph to be built. As the enforcement mechanism and the audit mechanism are one, the data captured can be tailored to focus on and meet the requirements of the policy being enforced. One of the main problems with provenance is the large storage overhead introduced by provenance data [Bates et al. 2015a]. By focusing the information logged to that relevant to and selected for some purpose, we can significantly reduce this storage overhead.

Information Flow Audit allows us to verify compliance by performing queries over the audit graph. A simple example would be a query verifying that there exists no path from a medical database to a research database without passing through an entity that anonymises the data. In the case of data leakage, studying the paths from the data source may enable how data leaked to be understood, and responsibilities attributed accordingly. We believe that Information Flow Audit can help in demonstrating compliance, assigning responsibility, helping investigation and generally improving transparency. Throughout the dissertation we refer to the combination of Information Flow Control and Audit as IFC&A. The implementation mechanism for capturing audit data is discussed in Chapter 5 and its application to compliance with regulation is discussed in Chapter 7.

1.4 Scope of the work: PaaS and SaaS clouds

This section defines the scope of this dissertation. There are many levels at which Information Flow Control and Audit could be implemented in the cloud and this dissertation focuses on container-based PaaS. Other implementation levels are discussed, but they do not constitute the core of the work. Further, the threat model is discussed, limiting the claim made on the guarantees provided by this work.

1.4.1 Container-based PaaS

One of the goals of CloudSafetyNet is to explore intra- and inter-tenant IFC. Our earlier work [Pasquier et al. 2014b] (see Appendix A) presented how IFC could be added to cloud-deployed web applications written in the Ruby programming language, with no modification to the base code. All the specification of IFC constraints was handled in a self-contained file, clearly separated from the rest of the application. Aspect Oriented Programming (AOP) [Elrad et al. 2001] was used as the mechanism to insert IFC checks.

We applied the technique to build a portal for brain cancer patients to access their medical data. IFC was used to guarantee segregation of data over two dimensions 1) patients (i.e. Bob cannot see Alice's data); 2) medical versus personal data (i.e. to prevent medical and personal data from being correlated). Although it worked, it could not operate

beyond the application border and required a large amount of the software stack to be trusted.

We wanted a mechanism to facilitate inter-tenant as well as intra-tenant IFC. This means that the mechanism should apply beyond the application border and across the whole system. Furthermore, we should ideally minimise how much of the software stack needs to be trusted. Looking back at Fig. 1.1, two choices seemed available: implementing IFC within the hypervisor or the operating system. We observed that container-based solutions [Bernstein 2014, Tucker and Comay 2004] were gaining traction in PaaS offerings [Dua et al. 2014, Strauss 2013].[14]

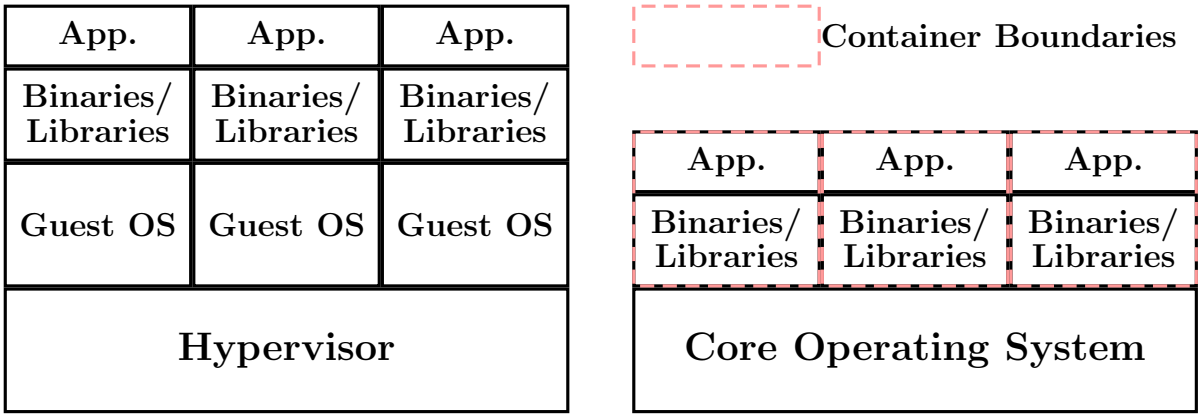

Figure 1.5: Traditional virtualization and paravirtualization require a full OS image for each instance, while containers can share a single OS and, optionally, other binary and library resources [Strauss 2013].

OS virtualisation was long assumed to be the only way to provide the necessary isolation for server applications. However, advances in container technology (notably in Linux OS-level isolation) have been disproving this assumption. Perceived as effective deployment solutions, products such as Docker[15] containers have seen increasing interest.

We therefore decided to implement the mechanism within the OS kernel to support container-based PaaS solutions. This 1) aligns with current trends, 2) pushes the enforcement as low as possible to reduce the amount of trust placed in the software stack, and 3) provides the easiest environment to develop a prototype. In consequence, this dissertation presents an OS implementation in Chapter 3 and a prototype container-based PaaS platform in Chapter 6. We discuss enforcement at other levels (e.g. hypervisor) or for other service delivery models (i.e. IaaS and PaaS), in Chapter 9.

Interaction across machines is handled through a modified version of the SBUS Messaging Middleware (MW) [Singh and Bacon 2014] (see Chapter 4), referred to as CamFlow-MW in this dissertation and our recent publications. Labelled communications are mediated by

[14] We discuss the rise of the Unikernel [Madhavapeddy and Scott 2014] in Chapter 9.
[15] `https://www.docker.com/`

the MW that ensures that IFC constraints are enforced on inter-machine message passing.

As shown in Fig. 1.1, our work can easily extend upwards to SaaS provisioning. In order to extend downwards to IaaS provisioning, two approaches can be envisaged: 1) implementing IFC within the hypervisor and enforcing IFC between virtual machines (VM); 2) using vTPM [Berger et al. 2006] and remote attestation [Kil et al. 2009] to verify the presence of OS-level IFC enforcement on interaction with other VMs. We discuss these possibilities further, as well as support for multi-cloud interactions and architectures where cloud services are part of wide-scale distributed systems, such as in IoT, in the concluding Chapter 9. CamFlow has been designed with such expansion in mind.

1.4.2 Limitations of the work

When building systems like CamFlow it is important to discuss and determine the threat model. This comes down to defining how the system may be attacked, by whom and for what purpose. The more severe the threat model, the more complex the system to protect against it and the more impact it may have on performance or programming techniques. Therefore a balance between risk and the required guarantees must be achieved.

CamFlow's enforcement is centred around data protection and providing transparency on data flows. Further, as mentioned, IFC is not seen as replacing but *complementing* other techniques. Therefore, traditional security techniques should be in place – these assumptions are stated for our PaaS proof-of-concept in Chapter 6. We believe that CamFlow helps in protecting against buggy, non-malicious code, virulent bugs – allowing unintentional execution of hostile code e.g. buffer overflow or SQL injection – and even hostile code that tries to "naively" transfer data outside of the specified security context. How this is achieved is discussed in more detail in Chapter 3. However, CamFlow does not provide guarantees against hostile code using *covert* channels – e.g. using CPU load [Okamura and Oyama 2010], timing channels [Cabuk et al. 2004] or cache based mechanisms [Percival 2005]. Such covert channels have a bandwidth which is an order of magnitude smaller than current overt channels. Therefore, CamFlow provides an improvement over the status quo; we prefer to see CamFlow as a data management mechanism rather than as a security mechanism.

This being said, IFC potentially assists in containing the effects of attacks; IFA (Information Flow Audit, see Chapter 5) logs attempted but disallowed flows and thus contributes to forensics. However, these security aspects have not been a focus of this work.

1.5 Challenges

IFC&A offers much potential for the cloud as a combination of a data management and an audit mechanism. However, a certain number of challenges need to be taken into account when building our solution.

Challenge 1: Extending data control beyond application boundaries. Security concerns hinder the adoption of cloud computing services, especially for sensitive personal data. The problem arises because data is not only shared between the data owner and an application, but generally between services and applications managed by (potentially several) third parties. Data analytics may be carried out on personal data without the owner's consent and be used for diverse purposes, such as targeted advertising. In general, there is concern that data may pass beyond the control of its owner and be used for unforeseen purposes. There is a need for a mechanism to express the data owner's usage requirements and to constrain application usage of data.

Challenge 2: Building complex policy from simple primitives. Our proposed approach must allow complex policies to be built, such as *medical data stored in database X must have received proper consent and be anonymised* [Singh et al. 2015c]. We need to demonstrate that such policies can be expressed as the composition of IFC labels and declassifiers/endorsers.

Challenge 3: Providing transparency. The mechanism proposed should provide means to audit and gain insight into the system behaviour. There should be a means for tenants and regulatory authorities to assess and verify proper usage of data as it flows through the system. Such an audit mechanism should allow demonstration that proper measures in regard to regulation are in place. We argue that being able to technically implement the above policies is not sufficient, it must also be demonstrated that they are in place.

Challenge 4: Performance overheads of additional security mechanisms. We argued that the low cost of the cloud computing paradigm has allowed an unprecedented rate of innovative services. The proposed mechanism should not present such a performance overhead as to increase the cost of adopting cloud computing unnecessarily.

Challenge 5: Flexibility and customisation. The proposed approach should be flexible enough to allow customisation to different usage. Cloud providers and/or tenants should not be constrained to a particular design, but rather our approach should provide enough flexibility to function with already existing technology and applications.

1.6 Dissertation outline and contributions

The thesis of this work is that it is desirable and possible to provide Information Flow Control and Audit at the OS level within cloud service provision in order to enforce and demonstrate compliance with high level policies such as regulations or contractual duties. This dissertation presents the formal basis of the thesis and its practical evaluation. In each chapter we highlight the original contribution and reference the relevant papers if published. The implementation underlying this work has been made available.[16]

Chapter 2: presents the CamFlow IFC model. Our novel contributions are extending IFC with separation of duty constraints [Pasquier et al. 2014a] and parametrisation of tags [Pasquier et al. 2015c]. We also discuss why implicit declassification /endorsement, present in some past IFC implementations, should be avoided [Pasquier et al. 2014a].

Chapter 3: describes the implementation of IFC at the kernel level through the Linux Security Module mandatory access control framework [Pasquier et al. 2015d]. We paid particular care to build an easily maintainable and self-contained mechanism, preserving compatibility with existing applications.

Chapter 4: presents a messaging middleware, providing the expected features of such software in a cloud environment, and enabling enforcement of IFC end-to-end across machines [Pasquier et al. 2015a, Singh et al. 2015a;b].

Chapter 5: presents how IFC can generate provenance-like data to provide audit logs. These logs are presented as directed graphs that can be used by a wide range of tools. To our knowledge, this is the first practical audit system for IFC. Some of the ideas presented in this chapter have been accepted for publication [Pasquier et al. 2016b].

Chapter 6: presents our PaaS prototype that uses the solutions presented in preceding chapters [Pasquier et al. 2014a; 2015a;c].

Chapter 7: presents use cases of the application of Information Flow Control and Audit to comply with regulation requirements and Data Protection Agencies [Pasquier and Eyers 2016, Pasquier and Powles 2015, Singh et al. 2014a; 2015c].

Chapter 8: presents the related work, focusing in particular on Information Flow Control and Provenance.

Chapter 9: concludes this dissertation and describes areas for future research.

[16] `http://camflow.org/`

Chapter 2

Information Flow Control models

In this chapter, we start by introducing the IFC model in §2.2 that will be used throughout the dissertation and an extension in §2.3. This chapter is based on the following published work [Bacon et al. 2014, Pasquier et al. 2014a;b; 2015c]. A further comparison with other IFC models is provided in Chapter 8.

2.1 Background

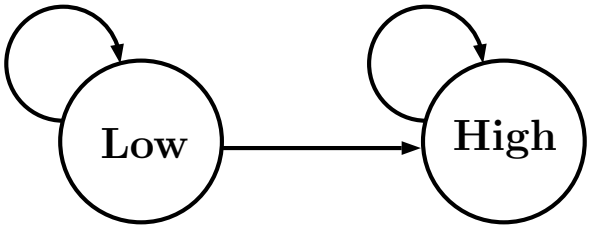

Figure 2.1: Simple non-interference. Arrows represent allowed flows.

IFC goes beyond access control, as it not only restricts a program's access to data, but also the propagation of data [Mantel and Sands 2004]. Its purpose is to guarantee non-interference between *security contexts*. The simplest case is when there are two kinds of data; high sensitivity and low sensitivity. In such a scenario, the flow of information is only allowed from low to high, as shown in Fig. 2.1. That is, a low-sensitivity process will not learn information about a highly sensitive process. In the more general case, partial ordering of security contexts can be achieved, generally as a lattice [Denning 1976]. The partial ordering defines where data can legitimately flow. Partial ordering within our model is discussed in §2.2.1.

However, practical systems may need some information to flow from high to low. Such a transfer of information should only be allowed if the data goes through a controlled

downgrading process. Here, we assume some policy must exist and be enforced by the downgrading process [Roscoe and Goldsmith 1999]. Downgrading within our model is discussed in §2.2.3.

The formal demonstration of the security of IFC when applied to operating systems has been presented in [Krohn 2008]. Our model derives from this work, with some further restriction of programming patterns, forcing explicit rather than implicit downgrading by the programmer (§2.2.3), and restricting the lattice that can be built (§2.2.5) to enforce Chinese Wall-like policies. We further refine tag definitions by defining and supporting parametrisation that modifies the partial order relationships (§2.3). The rest of this chapter describes this model and the modification we made to it.

IFC augments authorisation by enforcing dynamically that only permitted flows of information can occur, end-to-end, across applications. *Entities* to which IFC constraints are applied include cloud web applications [Pasquier et al. 2014a], a web worker instance [Akoush et al. 2014], a file, a database entry [Schultz and Liskov 2013], etc. IFC is applied continuously, typically on every system call for an IFC-enabled OS. IFC policy should therefore be as simple as possible, to allow verification, human understanding and to minimise runtime overhead. Our implementation of the model at the OS level is discussed in Chapter 3.

2.2 CamFlow IFC model

In this section we discuss the IFC model that will be used in the dissertation.

Description	Notation	Rule
Permissible Data Flow	$A \rightarrow B$	(2.1)
Creation Flow	$A \Rightarrow B$	(2.2)
Security Context Change	$A \rightsquigarrow A'$	(2.3)
Privilege Delegation	$A \overset{t^{\pm}_{X}}{\hookrightarrow} B$	(2.4) and (2.5)

Table 2.1: Types of flow

2.2.1 Enforcing safe flows via labels

A label comprises a set of tags, where each tag represents a particular security concern for a category of data. These concerns cover for example, sensitivity, purpose, quality, state, authority etc. In our IFC model two labels are associated with every entity A: a *secrecy label* $S(A)$ and an *integrity label* $I(A)$. The current state of these two labels (sets of tags) is the *security context* of an entity.

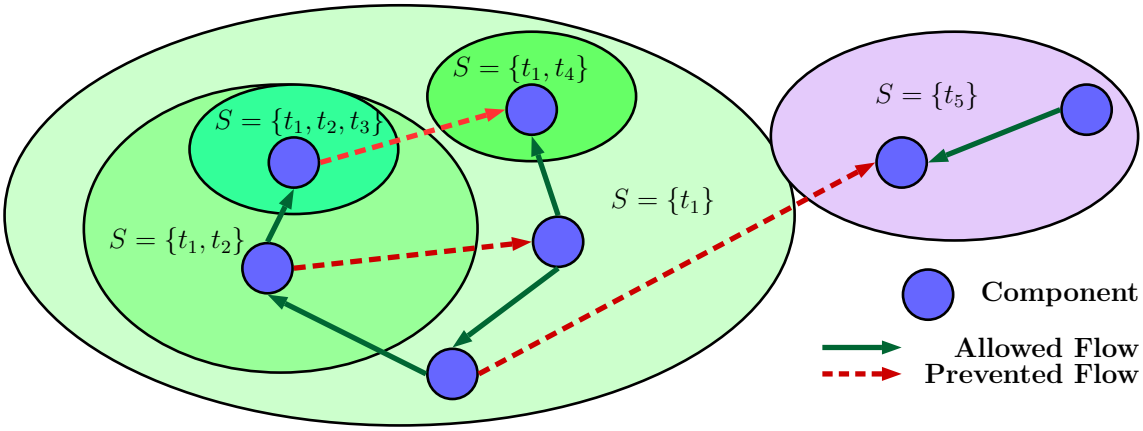

Figure 2.2: End to end secrecy enforcement.

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

$$A \to B, \text{ iff } S(A) \precsim S(B) \wedge I(B) \precsim I(A) \tag{2.1}$$

where $\precsim$ is any preorder (here it is mere inclusion $\subseteq$, this is refined for our two-component tag model in §2.3). These checks are simple to understand and apply, involving only matching of the tags at the flow endpoints. A non-permissible flow from A to B is denoted $A \nrightarrow B$.

Consider the *read* and *write* functions of [Bell and LaPadula 1973] [Biba 1977]. In the IFC world *read* is the equivalent of an incoming flow and *write* is the equivalent of an outgoing flow. The subrule concerning secrecy labels ensures that an entity only passes information to an entity that is allowed to receive it, thus enforcing "no read up, no write down". The subrule concerning integrity labels enforces quality of data during reading down and writing up. It is therefore possible to represent traditional security requirements as IFC constraints, although we use labels to represent more general security contexts, e.g. integrity can also indicate authority, such as to send an actuation command to a vehicle or home automation device, and allows an entity to express constraints on the data it is willing to manipulate (e.g. a data store only accepting medical data if encrypted).

Fig. 2.2 describes the end-to-end behaviour of data flow with respect to the secrecy dimension. Data produced in a certain *security context* can only flow within the same context or into a more restricted sub-context. In practice, this can be used to ensure that data produced by a component cannot be used for another purpose than the one originally defined (e.g. medical data is used within the medical security context).

Fig. 2.3, illustrates the end-to-end behaviour of data flow with respect to the integrity

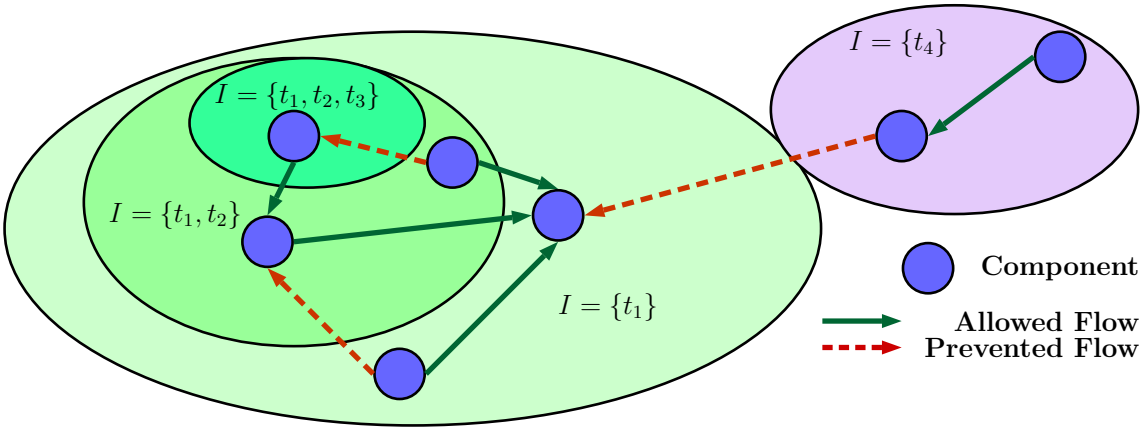

Figure 2.3: End to end integrity enforcement.

dimension. Data can only flow within a security context or towards a context with lower integrity. In practice, this can be used to enforce that an actuation request came through a trusted chain. Further, that any data and any previous commands contributing to the actuation request are trusted.

Example – secrecy: Suppose a hospital patient Alice, on being discharged to her home, is issued with a heart monitor. Data from this device is stored in her home and also flows to a process in the hospital's system, which carries out an analysis of her condition. Because Alice's health data is personal, the heart monitor and data are labelled with $S = \{alice, medical\}$. In order to receive this data, the hospital process's S label must also include the tags *alice* and *medical*.

Example – integrity: The hospital process is only allowed to receive data from a hospital-issued device and to achieve this is labelled $I = \{hospital\text{-}dev\}$. In order to send data to the hospital process, Bob's device and data must also be labelled $I = \{hospital\text{-}dev\}$. Suppose Bob's heart monitor is capable of remote actuation, e.g. to change the sampling rate if analysis detects a possible health problem. The device must only accept actuation commands from authorised sources, e.g. also labelled $I = \{hospital\text{-}dev\}$.

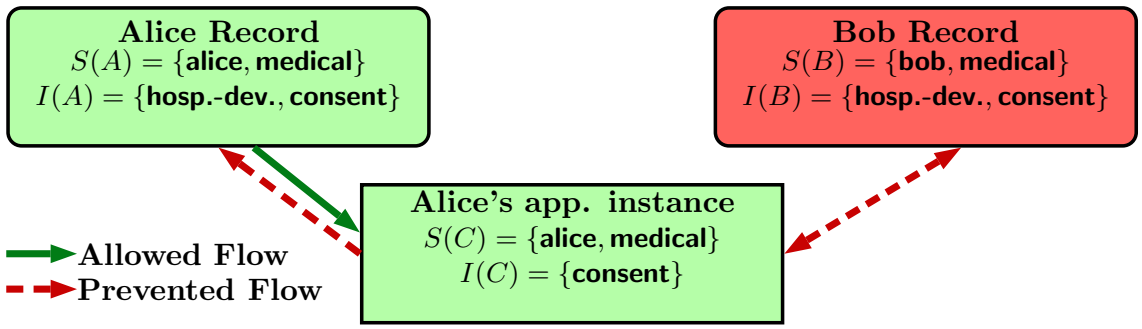

Figure 2.4: Flow constraint illustration.

Fig. 2.4 illustrates a combination of these two examples. An application is running on behalf of Alice to collect and process her medical data provided by her device. In addition, the application requires that consent has been verified. We further expand on medical use cases in Chapter 7.

2.2.2 Creation of an entity

We define $A \Rightarrow B$ as the operation of the entity A creating the entity B. We have the following rule for creation:

$$\text{if } A \Rightarrow B, \text{ then } S(B) := S(A) \text{ and } I(B) := I(A) \tag{2.2}$$

That is, the created entity inherits the labels of its creator. Examples are entities such as files or messages, or creating a process in a Unix-style OS by fork.

2.2.3 Privileges for managing tags and labels

It is possible to distinguish between two kinds of entity within our model.

- *Active entities* that perform some computation (i.e. processes) have a mutable security context. They may need to change their security context to meet some system requirement. Privilege management to achieve this is discussed below.
- *Passive entities* (e.g. file, socket etc.) simply contain information and have an immutable security context.

Active entities can have privileges that allow them to modify their labels. An entity has two sets of privileges for removing tags from its secrecy and integrity labels (P_S^- for S and P_I^- for I), and two sets for adding tags to these labels (P_S^+ for S and P_I^+ for I). That is, for an entity A to remove the tag $t_s \in S(A)$, it is necessary that $t_s \in P_S^-(A)$, similarly to add the tag t_i to the label $I(A)$ it is necessary that $t_i \in P_I^+(A)$.

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

$$\begin{array}{rcl} X(A) & := & X(A) \cup \{t\} \text{ if } t \in P_X^+(A) \quad \text{or} \\ X(A) & := & X(A) \setminus \{t\} \text{ if } t \in P_X^-(A) \end{array} \tag{2.3}$$

For example, in order to receive information from an entity B, an entity A will need to set its labels (if it has the privilege) such that the flow constraints expressed by the tags associated with B are respected; i.e. such that the flow $B \to A$ respects the safe flow subrules in rule (2.1). We propose the following notation: for a process and its labels, $A[S, I] \rightsquigarrow A[S', I']$ is the modification of the process labels following rule (2.3).

Only privileged processes can change their security context. Most processes need not be aware of IFC and an application manager can set up an appropriate IFC security context for the different processes that compose a given application. See Chapter 6 for an example of such application design.

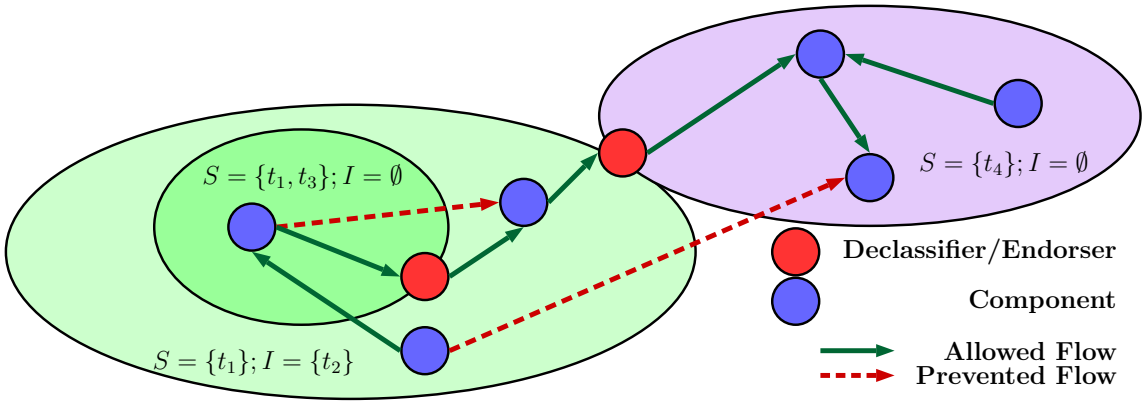

Figure 2.5: Declassification and endorsement.

In Figs. 2.2 and 2.3, we have shown how flows are always restricted to monotonically increasing secrecy constraints and decreasing integrity constraints. However, to build practical systems, information will likely have to flow between domains, thus breaking IFC constraints. Fig. 2.5 presents declassifiers and endorsers, the entities in the system that are trusted to perform such tasks. These entities are allocated privileges that allow them to change their security context in order to transfer data from one context to another. Generally, endorsers/declassifiers perform some operation on the data (e.g. analysis, transformation etc.) or verify authorisation and change security context if authorisation is successful.

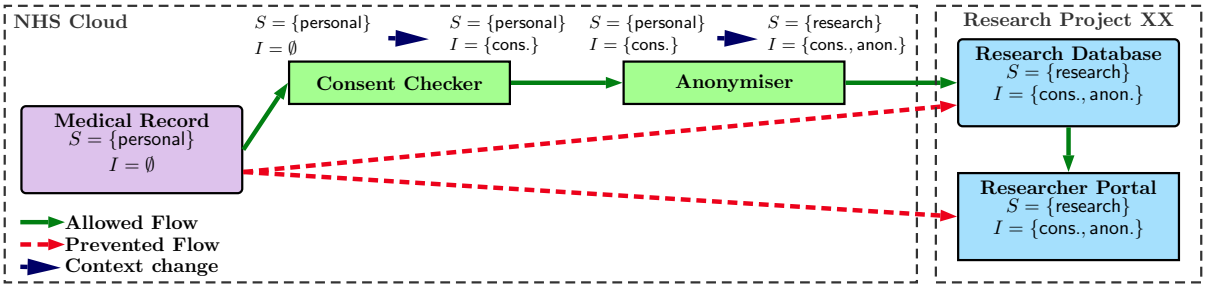

Figure 2.6: Medical data declassified and endorsed for research purposes.

Example – declassification: A medical record system is held in a private cloud. Research datasets may be created from these records, but only from records where the patients have given consent. Also, only anonymised data may leave the private protected environment. We assume a health service approved anonymisation procedure. Fig. 2.6 shows the anonymiser inputting data tagged as personal and declassifying the data by

outputting data with secrecy tag research. The underlying policies and legal issues are further discussed in Chapter 7.

Example – endorsement: Endorsement usually involves adding a tag to an I label. For example, a process might receive data from the network, carry out a verification process then output the data with tag *valid_data*. Such a process may be involved in data format conversion if non-standard data came from a remote source. A similar endorsement process can be used for many kinds of input data such as PHP scripts, downloaded software, indeed, any input amenable to a validation process.

In our hospital example, the patient may have received some treatment in another hospital or clinic and have a treatment record there, where the data format may differ. A process is charged with checking the patient's identity and verifying the data, including reformatting. The data is then output with tag *valid_data* and can be safely processed within the hospital domain.

Integrity tags may need to be removed after an anonymisation process, as the quality of data may have been degraded by the process. For example, if detailed information is removed, the data may no longer be proper to use as the source of an actuation command.

Previous work [Krohn et al. 2007, Porter et al. 2014] allows implicit declassification and endorsement. That is, if an active entity has the privilege to declassify/endorse and the privilege to return to its original state (i.e. for declassification/endorsement over t the entity has privilege t^- and t^+), the declassification/endorsement may occur implicitly without the need for the entity to make the label changes. We believe that this could in practice lead to *unintentional* data disclosure. Suppose an entity has the privilege to declassify top-secret information. The requirement for explicit label change makes it unlikely that the entity will send such data accidentally to an unintended recipient at a lower privilege level. Our model has stronger constraints that require endorsement and declassification operations to be expressed *explicitly* in the code.

2.2.4 Creation and privileges

On creation, labels are automatically inherited by a created entity from its creator (rule (2.2)), but privileges are not. If the child is to be given privileges over its labels, they must be passed explicitly. We denote the flow generated by an entity A giving selected privileges $t_X^\pm$ to an entity B as $A \overset{t_X^\pm}{\hookrightarrow} B$ (for example, allowing t to be removed from S, would be denoted $A \overset{t_S^-}{\hookrightarrow} B$). In order for a process to delegate a privilege to another process it must own this privilege itself. That is,

$$A \overset{t_X^\pm}{\hookrightarrow} B \text{ only if } t \in P_X^\pm(A) \tag{2.4}$$

2.2.5 Conflict-of-Interest (CoI)

A policy maker may need to specify a CoI (or Separation of Duty) between principals and/or roles [Brewer and Nash 1989, Sandhu 1990]. A CoI may arise when a principal could give professional advice to a number of competing companies. Separation of data access may be enforced by a Chinese Wall policy [Brewer and Nash 1989].

We define a set C of tags that represents some specified conflicting interests. In order for the configuration of an entity A to be valid with respect to C, rule (2.5) must be respected:

$$\left|\left(S(A) \cup I(A) \cup P_S^+(A) \cup P_I^+(A) \cup P_S^-(A) \cup P_I^-(A)\right) \cap C\right| \leq 1 \tag{2.5}$$

That is, an entity is non-conflicting in this context if the set of its potential tags (past, present and future) contains at most one element from the set of tags within the related CoI group. In detail, by potential tags we mean the tags in its current S and I labels and those tags that it has the privilege to add to $S(A)$ (i.e. $P_S^+(A)$) and to $I(A)$ (i.e. $P_I^+(A)$) or that it may have removed from $S(A)$ (i.e. $P_S^-(A)$) and from $I(A)$ (i.e. $P_I^-(A)$). Concretely, this means that no security context domain can exist with conflicting tags, and that data cannot be declassified/endorsed between conflicting security context domains. CoI rules need to be checked every time a privilege is granted.

Example – Conflict-of-Interest: A CoI can arise when data relating to competing companies is available in a system. In a medical context, this might involve results of analyses of the usage and effects of drugs from competing pharmaceutical companies. The companies might agree to analysis only if their data is guaranteed to be isolated, i.e. not leaked to competitors, and this can be demonstrated.

The hospital may be participating in drug trials and want to ensure that information does not leak between trials: suppose a conflict is::
$C = \{Pfizer, GSK, Roche, ...\}$
and some data (e.g. files) are labelled $PfizerData[S = \{Pfizer\}, I = \emptyset]$ and $RocheData[S = \{Roche\}, I = \emptyset]$. The CoI described ensures that it is not possible for a single entity (e.g. an application instance) to contain both *RocheData* and *PfizerData* either simultaneously or sequentially, i.e. enforcing that Roche-owned data and Pfizer-owned data are processed in isolation.

2.3 Parametrisable IFC model

This section looks at means to increase the expressiveness of the tag model presented in §2.2. The revised model is inspired by work on parametrisable role-based access control (RBAC). The benefit of the revised model is motivated through an exemplar use case

inspired by a medical research application.

2.3.1 Parametrisation in RBAC system

RBAC is a mature, widely used access control scheme with a large literature and existing standards [Ferraiolo et al. 2001, Sandhu et al. 1996].[17] Role definitions in RBAC tend to be functional in their scope, being application- or organisation-specific. Administrative roles are also included to capture the need to manage RBAC itself.

In work on RBAC [Bacon et al. 2002, Giuri and Iglio 1997, Lupu and Sloman 1997], parametrised roles were found to provide elegant expression of policy and avoid explosion in the number of roles required. For example, certain company software such as "payroll" may need to access all employees' data whereas each employee can access only their own data record. To achieve this, a company either creates a role per employee e.g. *employee_smith* or parametrises a single role, for example *employee*(*smith*) etc. The payroll software can then access *employee*(*), where * indicates all employees. We argue that the IFC label model needs similar refinement in order to provide system-wide enforcement of such aspects of application policy, thus following the Principle of Least Privilege (PoLP).

2.3.2 Motivation

A major aim of our IFC model is to provide a simple, human-understandable expression of policy that leads to clear, verifiable and efficient computation. Parametrisation should further simplify the management and expression of policies. In particular, for large-scale data processing, policy is expressed more concisely using the parametrised tag model and is easier to maintain. Further, the computation cost is comparable for small-scale processing and is reduced in a number of scenarios for large-scale processing.

We propose to decompose a tag t into a pair $\langle c, s\rangle$ with c the concern of type $\mathscr{C}$ and s a specifier of type $\mathscr{S}$. For example, the pair $\langle medical, bob\rangle$ represents Bob's medical data. A statistical analysis over a set of patients' medical data could be represented as $\langle medical, statistical\_analysis\rangle$ and anonymised medical records as $\langle medical, anonymised\rangle$.

A major requirement is to be able to specify *all* data records of a certain kind without enumerating all possible tags, as required by current models. Therefore for any concern c and specifier s we establish the subtyping relation in Fig. 2.7.

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

[17] `http://csrc.nist.gov/groups/SNS/rbac/`

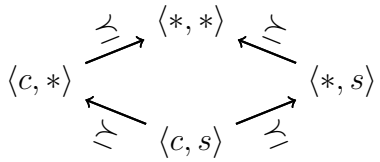

Figure 2.7: Subtyping relations.

Our model is designed for a distributed system or cloud platform where there is likely to be a large amount of user data stored with persistent labels in files, databases, key-value stores, etc. In a company context, data records may relate to individual employees; in a public health context, data may represent the medical records of patients; in an educational context, data may relate to students, staff etc. Specifying and enforcing access to all, some specified subgroup or only one data record of a given type is a universal requirement, discussed in the literature on policy.

Suppose a principal is allowed access to a subset of records, e.g. doctors may be able to access only the records of the patients they are currently treating. Temporally separated processes are likely, i.e. to deal with one patient's records at a time. Each time, current authorisation policy is enforced and is translated into labels to ensure correct behaviour at runtime (see an implementation example of such behaviour in Chapter 6). Note that as a doctor's group of patients under treatment changes, a lookup of current patients at the authorisation point in the application will ensure that labels are created only for current patients, selected at runtime from the entire database.

A use case arises from the need of a certain application to perform computations on all health records of a given type, whereas another application is authorised only to access records on behalf of a single individual. In the healthcare domain, statistical analysis of the medical records of patients is needed for various purposes including public health, environmental concerns, clinical practice etc. We are concerned with the privacy and confidentiality of medical data and will therefore discuss the construction of the IFC secrecy label for a statistical analysis program.

In the standard atomic tag model a tag represents a single security concern. To express the idea of Bob's medical data, we would use two tags *bob_data* and *medical_data*. The entity carrying out the statistical analysis of the medical data would then need to have not only the *medical_data* tag, but also a tag corresponding to every patient's data, for rule (2.1) to be satisfied. This makes the use of IFC infeasible for such purposes:

- Enumerating "all" tags would be prone to error as the database state changes, with records being added and removed.
- This entity would be over-privileged by the PoLP, being able to receive any data labelled only with the tag *bob_data* although our intention was for it only to be

concerned with *medical_data*. It would also be privileged to declassify, see §2.2.3, over all the patients' personal tags and trusted not to leak information about any patient.

2.3.3 Changes to flow constraints

To adapt rule (2.1) from §2.2.1 for the flow $A \to B$, we need only redefine the $\precsim$ binary relation between sets of tags X and Y as follows:

$$X \precsim Y \text{ iff } \forall t \in X \; \exists t' \in Y : t \preceq t' \tag{2.6}$$

Together with rule (2.1), this entails that a flow $A \to B$ is allowed if and only if for all secrecy tags of A there exists a supertype in the secrecy tags of B and that for all integrity tags of B there exists a supertype in the integrity tags of A.

Example – secrecy: The examples illustrating the atomic tag model given in §2.2 relate to hospital patients being monitored in their homes. We saw that the process receiving Bob's heart rate data, labelled with $S = \{medical, bob\}$, also needed to have the tags *medical* and *bob* in its S label.

For research purposes, it may be useful to amalgamate medical data from a group of patients to generate statistical data. Such an amalgamation process would need a tag for every such patient: $(S = \{medical, alice, bob, charlie, etc....\})$ and the (large) set of patients' tags would need to be kept consistent with the current set of home-monitored patients. Instead, the process is labelled $S = \{\langle medical, * \rangle\}$. This expresses the intended policy concisely and accurately, without the need to maintain the current set of patients' tags.

Example – integrity: Consider a home control system. An entity A labelled $I(A) = \{\langle actuator, * \rangle\}$ is able to send data (e.g. an actuation request or data used in the building of such a request) to an entity B labelled $I(B) = \{\langle actuator, alarm \rangle\}$ or an entity C labelled $I(C) = \{\langle actuator, light \rangle\}$. The entity A could represent the central domotic control system with B and C being actuators in the house. In a patient monitoring context the controller might be sending actuation commands to a variety of patient monitoring devices, e.g. to start, stop or change the monitoring intervals or thresholds. Since actuation affects the physical world, including people's health and safety, it is important that the authority of the whole actuation command chain is established, which is achieved by the flow rules concerning integrity labels.

2.3.4 Changes to privileges

Rule (2.3) becomes (where X is S or I):

$$\begin{array}{rcl} X(A) & := & X(A) \cup \{t\} \text{ if } \exists t' \in P_X^+(A) : t \preceq t' \quad \text{or} \\ X(A) & := & X(A) \setminus \{t\} \text{ if } \exists t' \in P_X^-(A) : t \preceq t' \end{array} \tag{2.7}$$

We also add special privileges noted $\langle c, \Delta\rangle$, $\langle \Delta, s\rangle$ and $\langle \Delta, \Delta\rangle$ that allow removal only of the tags $\langle c, *\rangle$, $\langle *, s\rangle$ and $\langle *, *\rangle$ respectively.
The privilege delegation rule (2.4), becomes:

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

Example – declassification: A process A, with the privilege $P_S^-(A) = \{\langle medical, \Delta\rangle\}$ and the label $S(A) = \{\langle medical, *\rangle, \langle medical, anonymised\rangle\}$ is able to declassify to $S(A) = \{\langle medical, anonymised\rangle\}$, but does not have the privilege to remove $\langle medical, anonymised\rangle$. The use of Δ privileges therefore allows the trust placed in a certain entity to be precise, and is particularly useful when specifying declassifier privileges. Without it, we would have had only $P_S^-(A) = \{\langle medical, *\rangle\}$ and no guarantee that the process would not declassify to $S(A) = \emptyset$ (also removing $\langle medical, anonymised\rangle$), thus allowing universal access to the anonymised data, rather than to medical research processes with the tag $\langle medical, anonymised\rangle$.

2.3.5 Changes to Conflict-of-Interest

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

$$\begin{array}{rcl@{\qquad}rcl} \pi_1 : \mathscr{C} \times \mathscr{S} & \to & \mathscr{C} & \pi_2 : \mathscr{C} \times \mathscr{S} & \to & \mathscr{S} \\ \pi_1(\langle c, s\rangle) & = & c & \pi_2(\langle c, s\rangle) & = & s \end{array} \tag{2.8}$$

We extend these operations to sets, such that:

$$\begin{array}{rcl@{\qquad}rcl} \pi_1 & : & \wp(\mathscr{C} \times \mathscr{S}) \to \wp(\mathscr{C}) & \pi_2 & : & \wp(\mathscr{C} \times \mathscr{S}) \to \wp(\mathscr{S}) \\ \pi_1(T) & = & \{\pi_1(t) \mid t \in T\} & \pi_2(T) & = & \{\pi_2(t) \mid t \in T\} \\ & = & \{c \mid \langle c, s\rangle \in T\} & & = & \{s \mid \langle c, s\rangle \in T\} \end{array} \tag{2.9}$$

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

$$SU(A) = S(A) \cup I(A) \cup P_S^+(A) \cup P_S^-(A) \cup P_I^+(A) \cup P_I^-(A) \tag{2.10}$$

A conflict of interest is denoted $P_{CoI}(f, C)$ where f is π_1, π_2 or id and C is a set of conflicting tags. Rule 2.5 in §2.2.5 becomes:

$$\forall P_{CoI}(f, C), |f(SU(A)) \cap C| \leq 1 \tag{2.11}$$

Here we note:

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

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

Example – conflict of interest: Consider the example used in §2.2.5 on isolating application instances that access drug trial data for different companies. If a new company was to use the application, a new tag would need to be added to the CoI group. For a rule applying to a more rapidly changing set this could prove problematic.

Using the two-component model, we have application instances labelled for example as:

$$[S = \{\langle drug, Roche\rangle\}, I = \emptyset] OR [S = \{\langle drug, Pfizer\rangle\}, I = \emptyset]$$

and the CoI policy is expressed as:

$$P_{CoI}(id, \{\langle drug, *\rangle\})$$

This is simple to read and understand (i.e. an application instance can manipulate information for only one specifier of concern *drug*) and this policy will not change over time as companies come and go.

2.3.6 Compatibility with atomic tags

Some tags function adequately as atomic tags, e.g. $\langle$*network-input*$\rangle$ may be included in an integrity label to indicate that input data should not be trusted. The tags $\langle$*EU-data*$\rangle$ and $\langle$*US-data*$\rangle$ can be used to enforce the geographical location of stored data to allow laws and regulations to be enforced. Such tags do not need to be two-component tags but can conveniently be expressed as such, e.g. $\langle input, network\rangle$ or $\langle location, EU\rangle$. Policy may sometimes be conveniently expressed for such tags using $*$, but if $*$ is never used, our two-component tag model degrades gracefully to what is effectively a conventional atomic tag model, since both components must match for data to flow, as for two separate

tags. Backwards compatibility with policies defined for atomic tags could be achieved, e.g. by a convention that the single tag should become a specifier of a null concern in a two-component tag.

2.4 Summary

This chapter presented the IFC model that has been used throughout this research. Although our basic IFC model follows standard practice for decentralised IFC since [Myers and Liskov 1997], first applied to OS in [Krohn et al. 2007], we have improved on it as follows: 1) enforcement of explicit label change on declassification and endorsement (§2.2.3); 2) addition of a mechanism to specify conflict of interest to enforce Chinese Wall-like policies (§2.2.5); and 3) the parameterisable tag presented in §2.3.[18]

Chapter 7 shows examples of tag naming to enforce policy and demonstrate compliance with law and regulation. Past work had mostly focused on secrecy tags in their examples of IFC application. We discuss how integrity tags can be used, beyond the classic example of sanitising input data, to allow services and applications to specify the expected properties of data in order to deal with regulations and place limits on the responsibility they are willing to take.

A major research question is "which aspects of access control policies need to be embodied in IFC tags for continuous, runtime, cross-application enforcement?". It is possible to design tags that capture every aspect of e.g. parametrised RBAC with environmental constraints [Bacon et al. 2002]. However, for IFC to be deployed in practice it needs to be simple to understand and evaluate, and to impose minimal overheads due to management and enforcement. Further work is needed on IFC label design for specific (cross-application) use cases, but our experience to date is that few tags are needed to capture legal/regulatory and compliance requirements at runtime (see Chapter 7) and many checks need only be done through declassifiers and endorsers. In this chapter we have shown that tags can be generalised to capture parametrisation, motivating this through a specific style of application that is central to cloud service provision, i.e. large-scale data analytics (see §2.3).

[18] Future work on n-elements may expand expressiveness further.

Chapter 3

Practical Information Flow Control for Linux

This chapter presents CamFlow-LSM, an implementation for enforcement of IFC within the Linux operating system via a Linux Security Module (LSM). A first prototype (FlowK) [Pasquier et al. 2014a] demonstrated the feasibility of implementing IFC without requiring modification of the existing system call API. However, despite successfully demonstrating this claim, FlowK had serious shortcomings [Pasquier et al. 2015a]. Indeed, the system call interposition method adopted was shown to have serious issues in terms of security [Garfinkel 2003, Watson 2007] and performance [Pasquier et al. 2014a]. This chapter is based on and extends the following published work: [Pasquier et al. 2014a; 2015a;d]. The work remains in progress and the current implementation is available online.[19]

3.1 Linux Security Module

The Linux Security Module (LSM) framework [Wright et al. 2003] is a general framework that allows a variety of different access control models to be implemented within the Linux OS. SELinux [Smalley et al. 2001] and AppArmor [Bauer 2006] are examples of two well-known mandatory access control (MAC) implementations as LSMs. LSMs have also been used beyond access control e.g. to implement Provenance within Linux [Pohly et al. 2012]. The purpose of the LSM framework is to allow the addition of further restrictions on top of the Linux default discretionary access control (DAC) mechanism.

The LSM framework extends kernel objects with a security field and provides security hooks, called when an access to an object is attempted, as illustrated in Fig. 3.1. The security field associated with kernel objects is used to store security metadata, used when making access decisions. The security hooks can be divided over two dimensions:

[19] `http://camflow.org/`

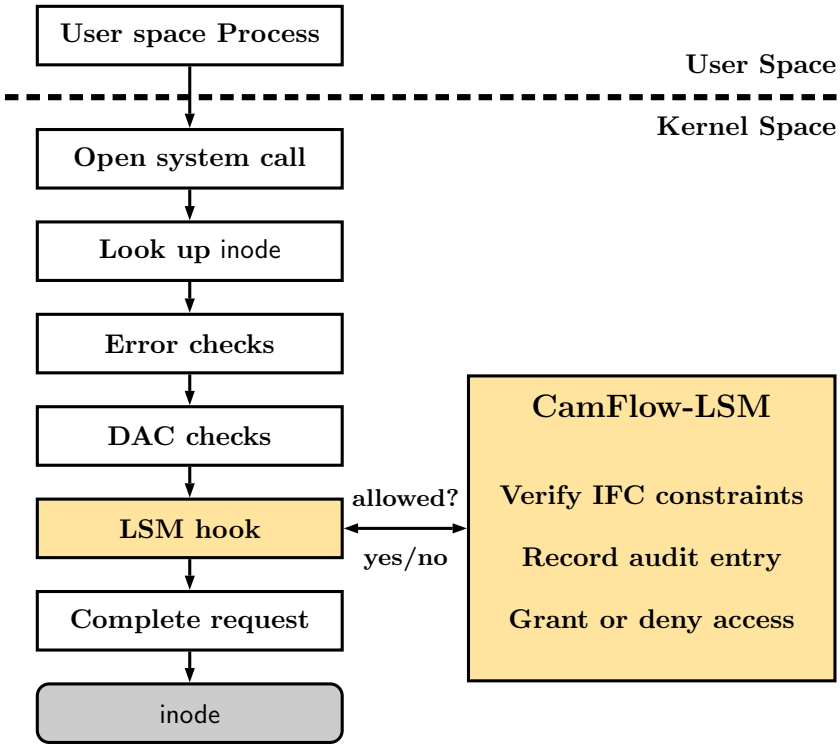

Figure 3.1: Linux Security Module hooks example on the open system call.

1. they can be grouped as logical sets corresponding to kernel objects (e.g. task, sock, file etc.) and some miscellaneous hooks for system operations;

2. they can can be grouped into two broad categories: hooks to manage the security field (e.g. alloc_security, free_security etc.),[20] and hooks that perform access control (e.g. inode_permission).

3.2 Design philosophy

Our kernel module, *CamFlow-LSM*, is implemented as a Linux Security Module. Although our work is Linux-specific, a similar approach could be used on any system providing LSM-like security hooks. Unlike other DIFC OS implementations [Krohn et al. 2007, Porter et al. 2014] our kernel patch is self-contained, strictly limited to the security module, does not modify any existing system calls and follows LSM implementation best practice. In contrast, for example, Laminar [Porter et al. 2014], mainly designed to support an IFC-enabled Java VM, modifies several hundred lines of code across the kernel in addition to the LSM itself, notably adding extra parameters to existing system calls. This large kernel modification renders Laminar hard to maintain, and represents non-trivial engineering effort to port to a new kernel version. In comparison, updating our LSM from kernel

[20] It also includes hooks to set information in the security field (e.g. inode_init_security).

version 3.17.8 to 4.1.5 required only a few lines of code to be changed, related to an unavoidable need to conform to kernel API modifications.

In this work, we assume that the rest of the kernel can be trusted and does not interfere with the IFC enforcement mechanism. LSM system hooks have been statically and dynamically verified [Edwards et al. 2002, Ganapathy et al. 2005, Jaeger et al. 2004], and our implementation inherits from LSM the formal assurance of IFC's correct placement on the path to any controlled kernel object. Therefore, we make the reasonable assumption that any operation to a controlled kernel object is intercepted.

We set ourself two design objectives: 1) to separate policy enforcement and policy management; 2) to minimise the amount of code required to run in kernel space. This led to an architecture where the LSM is solely focused on enforcement, while management aspects run in user space. By placing management aspects in user space, we allow tenants or cloud providers to customise these aspects. Kernel enforcement is discussed in §3.3, how applications interact with the kernel module to manipulate their security context in §3.4, and the API to build user space services in §3.5.

3.3 Kernel level IFC enforcement

Any interactions between a process and some other object in the kernel are intercepted by the LSM framework. The LSM framework calls CamFlow security hooks in order to: 1) allocate the proper security context to a newly created entity; 2) handle the security context on destruction of a kernel object; 3) make security contexts persist as extended attributes; 4) verify that IFC constraints are respected. CamFlow-LSM implements continuous label checking, as definied in IX [McIlroy and Reeds 1992], that is, not assuming that labels are static (since a process's labels can and will change during the course of an execution), and therefore verifying at each interaction that the labels on both sides accord.

Listing 3.1 gives an example of such security hooks where CamFlow-LSM verifies IFC constraints when a process attempts to connect to a socket. In the rest of this section we further discuss various aspects of the enforcement mechanism.

Applications running on SELinux [Smalley et al. 2001] or AppArmor [Bauer 2006] need not be aware of the MAC policy being enforced. Similarly, we see no reason to force applications running on an IFC system to be aware of IFC. This implementation choice is important; cloud providers can incorporate IFC without requiring changes in the software deployed by tenants. Alternatively, policy may be declared by applications through a pseudo-filesystem (as is typical for LSMs) abstracted by a user space library and enforced transparently by the IFC mechanism.

As discussed in §2.2.3:

- passive entities are entities in the system that do not perform computation tasks and

```
static int camflow_socket_connect
  (struct socket *sock, struct sockaddr *address, int addrlen){

  struct sock *sk = sock->sk;
  /* if local socket, verify IFC constraints */
  if(sk->sk_family==AF_UNIX || sk->sk_family==AF_LOCAL){
    return camflow_inode_permission
      (SOCK_INODE(sock), MAY_READ & MAY_WRITE);
  }else{ // only an unlabelled process socket can use other sockets
    /* retrieve current process security context */
    context_t* tsec = current_security();
    /* only unlabelled processes can use external sockets */
    if(!context_labelled(tsec)){
      return 0;
    }else{
      // labelled processes are not allowed to use external sockets
      return -EPERM;
    }
  }
}
```

Listing 3.1: Security hook for the connect socket system call.

cannot therefore modify their security context (e.g. file, pipe, shared memory etc.);

- active entities are entities that perform computation and can therefore modify their security context. In an OS, this corresponds to processes.

Passive and active entities are associated with secrecy and integrity labels. In addition, active entities are associated with privileges, determining how they can modify their security context. This section gives details of some enforcement mechanisms; other system calls/mechanisms are omitted as they are uninteresting and/or straightforward.

3.3.1 Tag and label representation

Tags and their corresponding privileges are presented by opaque 64 bit integers. They are generated from a monotonically increasing counter, and made opaque through the Blowfish block-cipher encryption algorithm [Schneier 1994]. Encryption prevents processes from making inferences based on the counter value. Listing 3.2 presents the code to create a new tag.

3.3.2 Fork and exec

Forking processes. In line with our requirements to avoid the modification of existing system calls and in order to avoid unintended data leaks by the programmer (see §2.2.2), any newly created process inherits the security context of its parent. In order for a child

```
int create_tag(tag_t* out ){
  tag_t in=0;
  int err=0;

  // atomic increment and persist counter, 0 if failed
  in = update_counter();
  if(!in){
    // make tag opaque
    crypto_cipher_encrypt_one(tfm, (u8*)&out, (u8*)&in);
  }else{
    // likely persistence failed, should retry
    err = -EAGAIN;
  }
  return err;
}
```

Listing 3.2: Simplified tag creation code.

to run in a different security context than its parents, one of the following patterns must be followed:

1. the parent sets itself to the desired security context and forks the child, potentially then returning to its original security context;
2. the parent forks a child and passes the privileges for the child to set its security context;
3. the parent uses a specific API to set the security context of the next forked child, if privileges allow (further details are given in §3.4).

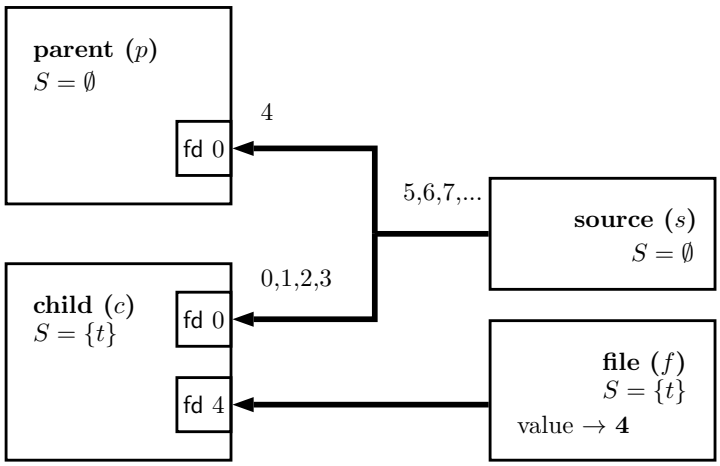

Figure 3.2: Leaking data through a shared file descriptor.

Preventing data leakage through a shared file descriptor. Fig. 3.2 describes a means for data to leak from child to parent across security contexts [Krohn 2008]. The

code corresponding to the figure is presented in the appendix Listing B.1. A parent (p) such that $S(p) = \emptyset$ and a child (c) such that $S(c) = \{ t\}$ share a file descriptor to a source file (s) such that $S(s) = \emptyset$. The source file contains a sequence of integers $\{0, 1, 2, 3, ...\}$. The child opens a file $S(f) = \{t\}$ and reads an integer value 4 from this file. It then reads that many (4) integers from the source. By reading 4 entries in the file, the child modifies the offset of the shared file descriptor. When the parent reads from the source file, it reads at the current offset[21] of the file descriptor which corresponds to the value 4. Although, the parent is unable to read the file f directly, the child is able to leak certain data to the parent across security contexts.

The default behaviour is for a child to inherit its parent's file descriptors. Such a data leak is prevented by freezing the security context of a process with shared file descriptors. That is, all shared file descriptors need to be closed before a process changes its security context. Alternatively, the parent needs to prevent their inheritance through, for example, fcntl(fd, F_SETFD, FD_CLOEXEC).

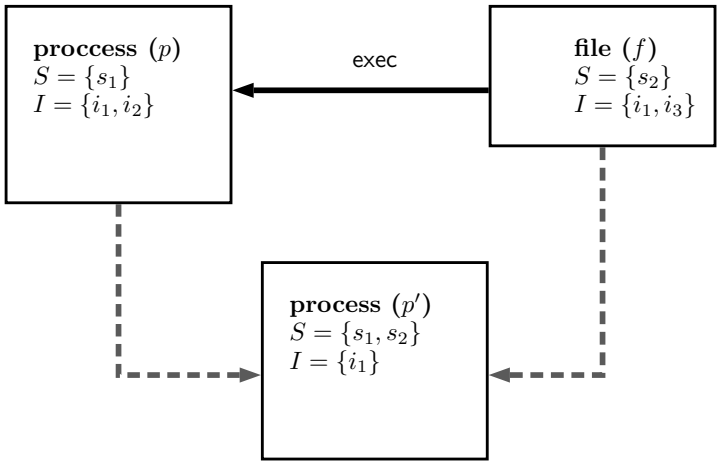

Figure 3.3: Setting security context through exec.

The exec command. In order to facilitate the setting up of an application in a particular security context, a process calling exec (or similar) inherits the security context of the executed file. Given a process p and a file f, p becomes p' after execution of f, such that $S(p') = S(p) \cup S(f)$ and $I(p') = I(p) \cap I(f)$. This is illustrated in Fig. 3.3. This operation is prevented if the resulting security context would violate Conflict of Interest constraints (see §2.2.5).

[21] This feature is likely to be part of the kernel to facilitate collaboration on file manipulation over tasks (threads/processes). Indeed from the fork manual page: *the child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description [...] as the corresponding file descriptor in the parent. This means that the two descriptors share open file status flags,* ***current file offset****, and signal-driven I/O attributes [...].*

3.3.3 Files

As in previous work [Krohn et al. 2007, Zeldovich et al. 2008], we conservatively assume that write implies read in some operations. Indeed, a writer can easily, for example, learn information about the size of a file. Every operation on inodes (which includes files and directories) is continuously verified, therefore a process that changed its security context may not be able to read from or write to an open file. The content and other attributes of a file (such as for example its size) are read/write constrained by the file label. On creation, files and directories inherit the security context of their creator.

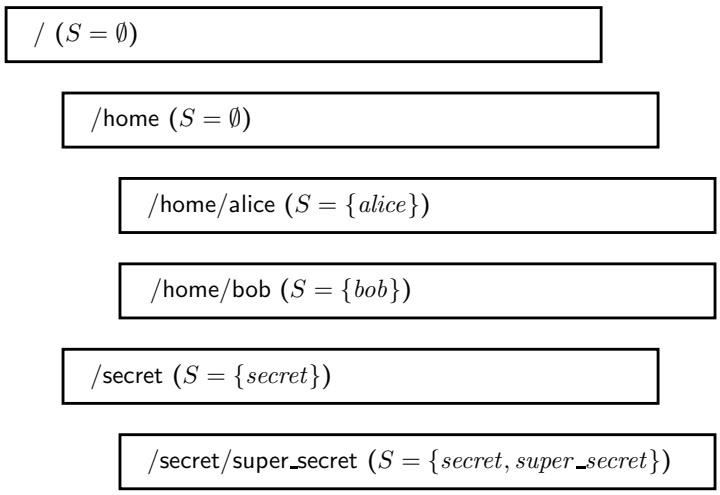

Figure 3.4: Secrecy in a directory structure.

Files and directories. A directory contains directories or files with ever-increasing secrecy constraints (the root directory / has $S = \emptyset$) and ever decreasing integrity (the root directory / has $I = *$ a special case representing all integrity tags). This means that a directory can only contain directories or files for which access is more restricted. In other words, given a parent directory P and a child directory/file C, $S(P) \precsim S(C)$ and $I(C) \precsim I(P)$, so information should be able to flow $P \rightarrow C$ (see §2.2.1). The corresponding security hook code for creating new directories is illustrated in Listing 3.3. Indeed, a process needs to be able to open a directory to be able to create a new entry in it. This is illustrated in Fig. 3.4, where directories down the file structure have an increasing number of secrecy tags.

The mmap command. The security context of a process is frozen until munmap is called for all mapped files. As discussed previously we conservatively assume write implies read. The IFC constraints applied (read or read/write) to map a file depend on the protection applied. When mapped as MAP_PRIVATE this is considered as read-only, even if the protection allows write, as changes are not applied to the underlying file or memory of other processes.

```
static int camflow_inode_mkdir(struct inode *dir,
  struct dentry *dentry, umode_t mode)
{
  struct inode* child_inode = d_backing_inode(dentry);
  // retrieve parent directory and child directory security context
  context_t *child_ctx = child_inode->security, *parent_ctx = dir->
    i_security;
  // check if the child can be created in the parent
  int rv = can_flow(parent_ctx, child_ctx);
  return rv;
}
```

Listing 3.3: Security hook for mkdir.

Making labels persistent. CamFlow-LSM uses the standard LSM approach of protected extended attributes to store the serialised secrecy and integrity labels of a file. These extended attributes can only be read or written from kernel space and are not accessible to user space processes.

3.3.4 IPC

Pipes. As files, pipes and sockets have inodes, on every operation the security context of a pipe is checked against the security context of the process reading or writing the pipe. The security context of a pipe is inherited from its creator and cannot be altered. We preserve the reliability (in the Unix sense) of the pipe, i.e., data written to the pipe are not dropped if the process at the other end is not able to read (because its security context changed). We believe this to be acceptable, in order to preserve the reliability of pipes. It would be possible to modify the implementation to silently drop data.

Further, in order to prevent illegal information flows, a pipe does not deliver an end-of-file (EOF) notification when the writer exits or closes the pipe if the writer cannot write to the pipe at the time of exit. As in most OS DIFCs [Krohn et al. 2007, Porter et al. 2014], CamFlow delivers EOF notifications only if the notification constitutes a legal flow. Again, there should be a timeout on pipe to avoid deadlock of a process waiting on an EOF notification.

Unix/Local Socket. Sockets are generally not accessible to labelled processes, as illustrated in Listing 3.1. Unix/Local sockets function in a fashion similar to pipes or files.

Shared memory. As for mmap mapped files, shared memory objects freeze the security context of a process until the process has unmapped all shared memory. See §3.3.3.

Messages. When a new message queue is created it inherits the labels of the creating process. When an existing message queue is opened, IFC constraints are checked, depending on the read/write mode (again we conservatively assume write implies read in this situation). Labels are checked continuously against the message queue labels on any other subsequent

system calls (e.g. mq_send, mq_receive etc.).

3.3.5 Privilege management

Active entities can create tags and are automatically allocated privileges over newly created tags. Privileges over tags can be shared with other principals: processes, users or groups. Privileges owned by groups are inherited by the users belonging to these groups. Standard entities cannot use privileges owned by users or groups. Those privileges are used by trusted entities which manage and set up the environment on behalf of a user.

Privileges are allocated by the kernel and owned by the creating process (any process can create tags and the associated privileges in a decentralised fashion). Privileges can be passed to other processes, users or groups. A process can add or remove a tag from its label if it owns the appropriate privilege, if the current user owns the privilege or if the current group owns the privilege. How tags are shared and managed must be considered with care when designing an application and the system must be administered accordingly.

Users and groups are those of a Unix system. In a cloud context they could respectively match applications (processes are application instances) and tenants. This mechanism allows the sharing of privileges to be managed easily and therefore data between applications and tenants.

3.4 User space API and library

Interaction between a user-space process and the LSM is achieved through a pseudo file system (like /proc or /sys), in a similar fashion to SELinux. This pseudo file system is then abstracted through a user-space library; Listing 3.4 gives an overview. User-space applications should generally not interact directly with the pseudo file system, but use the library-provided API instead.

The main core API is mostly focused on tag creation, manipulation and the associated privileges. Further management functionality is provided, but is related to user-space helpers. These are special services that provide functionality outside of the kernel to facilitate application development. They are discussed in the next section.

3.5 User space helper services

User space helper services (ushers[22]) are used to perform tasks in user space that are not directly related to the enforcement of IFC constraints. We implemented prototype

[22]An usher is an official in a court of law that ensures secure transaction of documents and escorts participants to the courtroom.

```c
/* Create a new tag, grant privileges to calling process and associated
   user. */
tag_t create_secrecy(void);

/* Copy current process secrecy label into the provided buffer. */
int get_secrecy_label(tag_t* buffer, size_t size);

/* Add secrecy tag to current process security context if privileges allow.
    */
int add_secrecy(tag_t tag);

/* Remove secrecy from current process security context if privileges allow
   . */
int remove_secrecy(tag_t tag);

/* Pass secrecy privilege for adding tag to the process associated to the
   pid. */
int pass_secrecy_p(pid_t pid, tag_t tag);

/* Pass secrecy privilege for removing tag to the process associated to the
    pid. */
int pass_secrecy_n(pid_t pid, tag_t tag);

/* Allow definition of the security context in which the next child should
   be. It fails if privileges would not allow. */
int set_child_in(tag_t* secrecy, size_t s_size, tag_t* integrity, size_t
   s_size);
```

Listing 3.4: CamFlow-LSM user space API (only secrecy API shown, corresponding functions exist for integrity).

ushers, providing the minimum functionality needed. However, other ushers respecting the expected behaviour and presenting the expected interface to the kernel can be built. This design decision supports a more modular implementation, allowing developers to meet the requirements of a particular environment. The ushers can be implemented to fit with existing services on the platform or to provide specific functionality as required.

Fig. 3.5 represents the three types of usher in our prototype (implementation details are given later in this section; practical examples are given in Chapters 4, 5 and 6):

1. Each application can have several attached *bridge*-ushers that allow the application to interact with applications running outside of the IFC enforcement or implementing IFC at a different level such as, for example, IFDB [Schultz and Liskov 2013]. The role of bridge-ushers is to work as trusted intermediaries ensuring that IFC constraints remain in place.

2. A manager-usher is shared by the whole machine. Its role is to allow the persistence of security contexts across executions, make the tag counter persist (see §3.3.1) across executions and save metadata attached to a tag (e.g. its representation for inter-machine communication as discussed in Chapter 4).

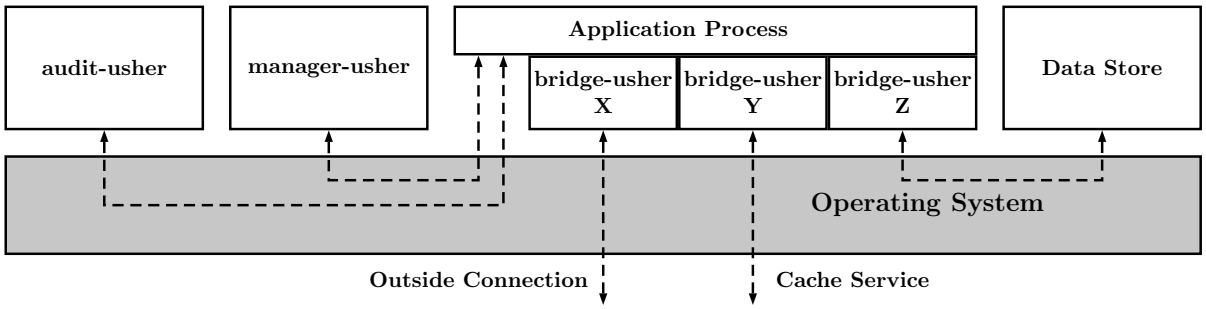

Figure 3.5: Ushers and an application process.

3. The audit-usher records or processes the information on data flow capture by CamFlow-LSM.

```
/* Copy the secrecy label of the process associated with pid into the
   buffer. */
int get_remote_secrecy_label(pid_t pid, tag_t* buffer, size_t size);

/* Copy the secrecy privileges for adding tag of the process associated
   with pid into the buffer. */
int get_remote_secrecy_p(pid_t pid, tag_t* buffer, size_t size);

/* Copy the secrecy privileges for adding tag of the process associated
   with pid into the buffer. */
int get_remote_secrecy_n(pid_t pid, tag_t* buffer, size_t size);

/* Create a netlink socket connecting to CamFlow-LSM. */
int camflow_create_socket(int unit);

/* Send data to CamFlow-LSM.  */
ssize_t camflow_send(int sockfd, const void* buff, size_t len, int flags);

/* Receive data from CamFlow-LSM. */
ssize_t camflow_recv(int sockfd, void *buf, size_t len, int flags);

/* Receive data from CamFlow-LSM. */
int camflow_get_context(pid_t, tag_t* secrecy, tag_t* integrity);
```

Listing 3.5: General Usher API (only the secrecy API is shown, corresponding functions exist for integrity).

The interaction between the LSM module and the ushers is achieved through the use of a Netlink socket [Dhandapani and Sundaresan 1999, He 2005]. Netlink sockets are used to transfer information between kernel space and user space processes by providing kernel/user space bidirectional communication links. A Netlink socket consists of a standard socket interface for user processes (using sendmsg and recvmsg) and an internal kernel API. Listing 3.5 describes an extra layer of abstraction for user space processes that format messages for communication with CamFlow-LSM. The standard Linux socket interface can

```
/* request a bridge-usher to be attached to this process */
chan = camflow_attach_bridge("my_bridge");

/* create a netlink socket connecting to the kernel level implementation of
    the bridge-usher */
sckfd = camflow_create_socket(chan);

/* send data to the bridge-usher */
camflow_send(sckfd, HELLO, strlen(HELLO), 0);

/* receive data from the bridge-usher */
camflow_recv(sckfd, buffer, 256, 0);
printf("received: %s\n", buffer);
```

Listing 3.6: Simplified bridge client code.

also be used directly, provided the applications follow the expected format. Communication between applications and ushers is mediated by CamFlow-LSM and is similarly based on a Netlink socket interface. Usher processes can retrieve the security context of another process in order to perform IFC or policy checks.

3.5.1 Bridge-usher

The purpose of these ushers is to *bridge* between applications running within our IFC-enforcing OS and systems enforcing IFC at a different level. For example, a bridge-usher can be used to allow applications to interact with data management systems enforcing IFC constraints (e.g. IFDB [Schultz and Liskov 2013]). We give an example of a bridge-usher allowing interaction with an IFC-enforcing key-value store in Chapter 6.

Another example is to allow communication between machines enforcing IFC, through the use of messaging middleware that enforces IFC at this level (as presented in Chapter 4). Indeed, only processes P such that $S(P) = \emptyset$ (i.e. not subject to security constraints) are allowed to directly connect to or receive messages from outside connections (e.g. through a socket), see Listing 3.1. In order to connect to the outside world, a process must either: 1) be able to declassify to $S = \emptyset$; or 2) communicate through a bridge-usher.

```
/* Spawn and attach the bridge-usher process to the process calling this
    function, the name corresponds to the name specified in the system
    configuration. */
int camflow_attach_bridge(char* name);

/* Called by the bridge-usher process as start-up, finalise the association
    of the bridge-usher and its attached process. */
int camflow_register_bridge(void);
```

Listing 3.7: Bridge-usher API.

The bridge-usher framework associates with a constrained process an usher process that

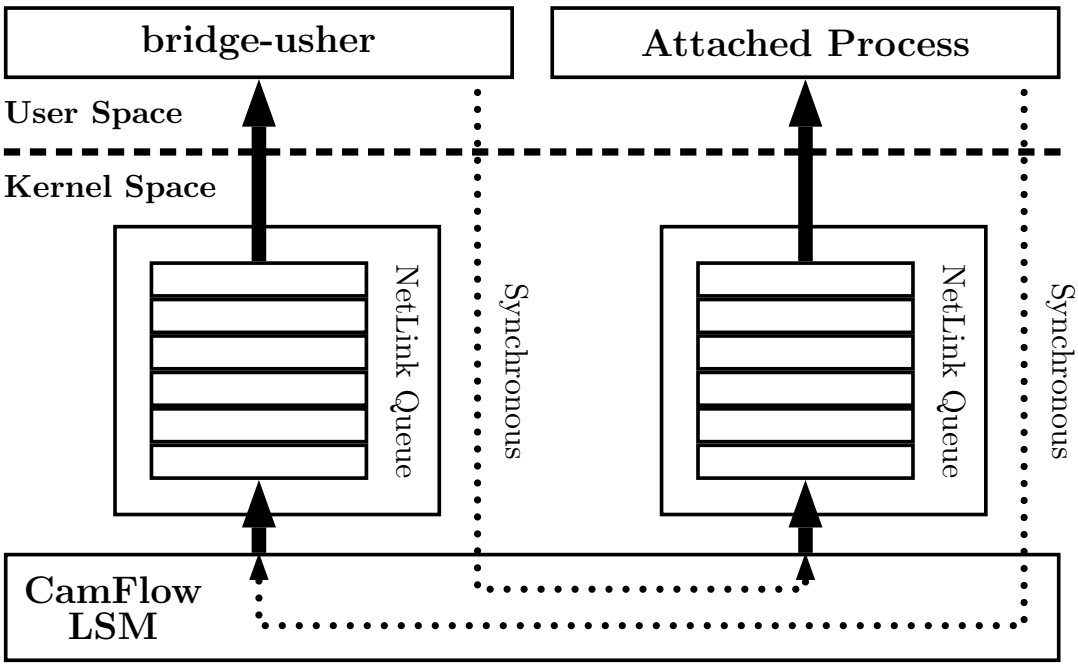

Figure 3.6: General architecture of a bridge-usher and its attached process.

can perform operations outside of the IFC constraints applied to the process. Interaction between a process and its bridge-usher is done through a socket interface.

Data sent through the socket is forwarded by the LSM from the constrained process to its associated usher (and vice-versa). The LSM also logs the message being exchanged through the audit-usher implementation described in Chapter 5.

```
/* register with LSM the newly spawned bridge-usher */
camflow_register_bridge();

/* create a netlink socket connecting to the kernel level implementation of
    the bridge-usher */
sckfd = camflow_create_socket(NETLINK_CAMFLOW_BRIDGE);
do{
  /* receive data from the bridge client */
  camflow_recv(sckfd, buffer, 256, 0);
  printf("received: %s\n", buffer);

  /* send data to the bridge client */
  camflow_send(sckfd, buffer, strlen(buffer), 0);
}while(1);
```

Listing 3.8: A simplified bridge-usher that prints received messages to standard output and echoes them back.

As seen in Listings 3.6 and 3.8, basic implementation of client and bridge-usher code is extremely simple. A first step is to attach/register the usher (for client and usher respectively) then use primitives that provide similar semantics and interface as a standard

socket.

3.5.2 Manager-usher

The manager-usher fulfils three main functions:

1. making the tag counter persist across OS executions on behalf of the kernel;
2. storing write-once, read-many information on tags at creation, such as a global name understood across machines;
3. storing the security context of an application across multiple executions.

```
/* Register the manager-usher callback. */
int register_manager(manager_op_t* op);

/* Stop the manager-usher. */
int stop_manager(void);

/* Callback implemented by the manager-usher to persist the current tag
   counter. */
int push_tag_counter(tag_t counter)

/* Callback implemented by the manager-usher to retrieve the current tag
   counter. */
int pull_tag_counter(tag_t counter);

/* Callback implemented by the manager-usher to persist the metadata
   associated with a tag ID. */
int push_tag_metadata(tag_t tag, char* metadata, size_t size);

/* Callback implemented by the manager-usher to return the metadata
   associated with a tag ID. The kernel has verified that the calling
   process own the proper tags or associated privileges. */
int pull_tag_metadata(tag_t tag, char* metadata, size_t size);

/* Function called by the manager-usher to restore the security context of
   a given process, normally following authentication or token verification
   . The kernel has verified that the calling process security context is
   empty. */
int restore_context(pid_t pid, tag_t* secrecy, size_t s_size, tag_t*
   integrity, size_t i_size);

/* Callback implemented by the manager-usher providing the file descriptor
   to a socket connecting to a process wishing to restore/save its security
    context. */
```

```
void context_rqst(int sckfd);
```

Listing 3.9: Manager-usher API.

CamFlow-LSM communicates through a Netlink socket with the manager-usher in order to make its tag counter persist and retrieve the current value at boot time (the manager-usher is started during the boot process). The manager-usher is also used to make users and group privileges persist across OS execution.

The manager-usher can associate user-space-relevant metadata with a particular tag (e.g. a global name or a display name). In order to do so, the process must have created the relevant tag and must have $S = \emptyset$ to avoid using tag metadata as a side channel. A process having a tag or the corresponding privileges can request to read the metadata associated with this tag. Other ushers can also retrieve metadata associated with a tag in order to perform their task.

The kernel provides a socket-based connection when a process wants to restore a security context. This connection can be used for authentication or token verification when an application process wants to restore a previously stored security context.

```
/* callback to store tag counter */
int push_tag_counter(tag_t counter);
/* callback to retrieve tag counter */
int pull_tag_counter(tag_t* counter);
/* callback to store tag metadata */
int push_tag_metadata(tag_t tag, char* metadata, size_t size);
/* callback to retrieve tag metadata */
int pull_tag_metadata(tag_t tag, char* metadata, size_t size);
/* callback for token verification/authentication and to save context*/
void context_rqst(int sckfd){
  /* ... */
  read(sckfd, ...);
  /* ... */
  write(sckfd, ...)
  /* ... */
  restore_context(...);
  /* ... */
  close(sckfd);
}

struct manager_op op = {
  .push_tag_counter = push_tag_counter,
  .pull_tag_counter = pull_tag_counter,
  .push_tag_metadata = push_tag_metadata,
  .pull_tag_metadata = pull_tag_metadata,
  .context_rqst=restore_rqst
};
```

```
28
29 int main(void){
30   register_manager(&op); // register callbacks
31   /* do something */
32   stop_manager(); // stop audit recording
33   return 0;
34 }
```

Listing 3.10: Basic manager-usher code skeleton.

The basic implementation of a manager-usher using the API presented in Listing 3.9 is given in Listing 3.10. The manager-usher must implement the different callbacks required to provide functionality, while the underlying implementation is left to developer choice.

```
1 /* Callback implemented by the manager-usher to persist the metadata
     associated with a tag ID. */
2 int save_tag_metadata(tag_t tag, char* metadata);
3
4 /* Callback implemented by the manager-usher to return the metadata
     associated with a tag ID. The kernel has verified that the calling
     process owns the proper tag or associated privileges. */
5 int retrieve_tag_metadata(tag_t tag, char* metadata);
6
7 /* Return a socket connecting to the manager-usher in order to restore or
     save a security context. */
8 int get_context_mngr_sck();
```

Listing 3.11: Application side manager-usher API.

Listing 3.11 presents the application-facing API to interact with the manager-usher. The application is provided with functions to save and retrieve metadata associated with a tag it created. A function allows the application to retrieve a socket connected to the usher-manager. This socket allows for the saving and restoration of the security context.[23]

Discussion: The intent of the manager-usher is to provide a customisable framework for the tags and security context persistence. The approach discussed in this section may prove overly complicated and depart too much from standard Linux management. The author is currently exploring an alternative using the Linux kernel key retention service [Edge 2006].[24] This service is designed to store and manage credential information such as authentication tokens, cryptographic keys and cross-domain user mappings, for use by the kernel and file systems. Such credential information can be managed by processes with the appropriate privileges. Future versions of CamFlow will replace the manager-usher presented here, by an implementation relying on the more standard service discussed above.[25]

[23] Details are left to the developer's choice.

[24] `https://www.kernel.org/doc/Documentation/security/keys.txt`

[25] Progress can be followed at `https://github.com/CamFlow/camflow-dev`.

3.5.3 Audit-usher

The enforcement of IFC can naturally be used to generate audit of data-flow in the system. The particulars of audit collection are discussed in Chapter 5.

3.6 Evaluation

We evaluate CamFlow-LSM over two dimensions: the ease of development and the overhead introduced over standard Linux. Further evaluation is provided in Chapter 5 and 6 on other specific aspects. Chapters 7 looks at how complex policy can be enforced from the simple enforcement mechanism described in this chapter.

3.6.1 Programmability

Programmability in MAC systems – and IFC in particular – is often perceived as difficult. One of the main difficulties when building IFC is the phenomenon of label creep [Sabelfeld and Myers 2003], where the number of constraints increases to a point where no application in the system is able to function. Systems such as HiStar [Zeldovich et al. 2006] and Asbestos [Vandebogart et al. 2007] limited label creep through the decentralisation of privileges. However, as clean-slate OS implementations they required either to re-implement the whole software stack or at least to port existing applications. Flume [Krohn et al. 2007] limited such issues by augmenting standard Linux with IFC, while retaining the standard system call API for non-constrained processes. However, constrained processes were faced with a non-standard system call API and re-implementation was required in order for constrained applications to function correctly. Further, this required patches to glibc and ld.so to function.

CamFlow-LSM takes the approach of preserving the standard API for all processes at the cost of slightly more effort when manipulating security contexts. This allows unmodified applications to run within IFC contexts transparently and without any engineering effort. Modification is only required for applications that need to manipulate their security context. As is argued in Chapter 7, security context manipulation should be well-separated from the main application logic.

Another issue is the experience of developers programming within the CamFlow environment. The difficulty arises from the very purpose of CamFlow, preventing flows of information outside of a well defined security context. This means that applications are very hard to monitor as they often cannot easily be instrumented. A solution is to allow debugging through print and to route stdout to a trusted file for which IFC rules are relaxed – such issues and workarounds were first discussed in [Krohn 2008].

Error reporting is another issue. Indeed, error reporting could be a source of data leaks

in itself, so in best-case scenarios the programmer is faced with a generic "permission denied". In other scenarios – such as in pipe programming as discussed in §3.3.4 – the programmer is faced with the consequence of context mismatch. However, gaining understanding of what went wrong can be achieved through the audit mechanism, described later in Chapter 5.

Finally, containing the whole enforcement mechanism to a single LSM, rather than a wide-ranging kernel patch, allows the code to be updated easily across kernel versions. This avoids the issue met by Flume that *"important maintenance tasks remain [...] since even minor kernel revisions can break the patch"* [Krohn 2008]. Further, a lot of code is kept in user space through ushers, and can easily be maintained and customised to meet particular environment requirements.

3.6.2 Micro-benchmark

We tested the CamFlow-LSM module on Linux Kernel version 3.17.8 (01/2015) from the Fedora distribution.[26] The tests are run on a machine with an Intel 2.2Ghz i7 CPU and 6GiB RAM.

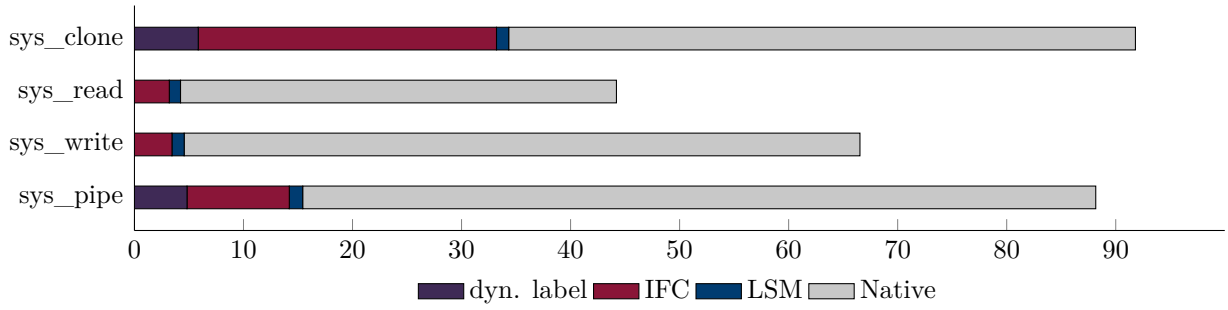

Figure 3.7: Overhead introduced into the OS by CamFlow LSM (x-axis time in μs).

Measurements are done using the Linux tool ftrace [Bird 2009] to provide a microbenchmark. Two processes read from and write to a pipe respectively. Each has 20 tags in its security label, substantially more than we have seen a need for in current use cases. We measure the overhead induced by: creating a new process (sys_clone), creating a new pipe (sys_pipe), writing to the pipe (sys_write) and reading from the pipe (sys_read). The results are given in Fig. 3.7.

We can distinguish two types of induced overhead: verifying an IFC constraint (sys_read, sys_write) and allocating labels (sys_clone, sys_pipe). The sys_clone overhead is roughly twice that of sys_pipe as memory is allocated dynamically for the active entity's labels and privileges. Recall that passive entities have no privileges. Overhead measurements

[26]It is not feasible to provide a comparison with the Laminar implementation [Porter et al. 2014], that is closest in technical terms to our work, as the implementation available `https://github.com/ut-osa/laminar` is for an obsolete kernel version 2.6.22 (07/2007).

for other system calls/data structures are essentially identical as they rely on the same underlying enforcement mechanism, and are not included.

In some previous work [Krohn et al. 2007, Pasquier et al. 2014a], IFC was introduced into OS kernels by interposition techniques for which overheads were multipliers. The CamFlow-LSM overhead is a few percent, see Fig. 5.5. We provide a build option that further improves performance by limiting labels and privileges to a maximum fixed size (by default, label size can increase dynamically to meet application requirements). This reduces the overhead of the system calls that create new entities (the dynamic label component in Fig. 3.7). However, for most applications, the overhead is imperceptible and lost in system noise. It is hard to measure without using kernel tools, as the variation between two executions may be greater than the overhead.

3.7 Summary

This chapter presented the work done on CamFlow-LSM, implementing the IFC model presented in §2.2. While implementation for the Linux kernel [Krohn et al. 2007] and as a LSM [Porter et al. 2014, Roy et al. 2009] have been proposed in the past, the implementation presented here presents distinct advantages: 1) conscious effort was made not to alter the system call interface and not to require re-engineering of IFC-constrained applications that do not manipulate their security context; 2) to provide a simple and audited socket interface for applications and trusted components to interact; 3) to be self-contained, we avoid any modifications to the kernel as the IFC enforcement is strictly contained within the LSM; 4) to provide performance in line with or better than that reported for comparable implementations on micro benchmarks.

In order to realise the vision presented in the introduction, it is necessary for us now to explore how IFC enforcement can be extended beyond a single machine.

Chapter 4

Enforcing Information Flow Control in a distributed system

The continuous protection of data is achieved through application of the IFC model described in Chapter 2. CamFlow enforcement is implemented at two levels: 1) at the OS level, for within-machine enforcement and 2) at a message-passing level, for cross-machine enforcement. This section presents a design proposal for a messaging-middleware, working alongside our OS level enforcement presented in Chapter 3. The proposed middleware is currently being implemented as a bridge-usher, using the feature presented in §3.5.1.

The presented design builds upon SBUS [Ingram 2009, Singh et al. 2014b] a messaging middleware that supports strongly-typed messages; a range of interaction paradigms, including request-reply, broadcast, and stream-based; flexible resource discovery mechanisms; and security including access controls and encrypted communication. It also provides dynamic reconfiguration capabilities. The implementation runs on Unix-like OSs, including Linux [Ingram 2009] and later extended to OSX, Android and iOS [Singh et al. 2014b].

My conceptual design, CamFlow-MW uses and extends SBUS to enable IFC enforcement across machines. From here, we use CamFlow-MW as subsuming SBUS functionality. We only introduce middleware concepts as relevant to the IFC discussion; see [Singh and Bacon 2014, Singh et al. 2014b; 2015b] for SBUS specifics. This chapter is based on and extends the following published work [Pasquier et al. 2015d, Singh et al. 2015a;b]. At the time of submission, this aspect of the work remains in development and evaluation is left to future work. This is further discussed in Chapter 9.

4.1 Middleware Overview

CamFlow-MW is a particular example of implementation of the bridge-usher described in §3.5.1. A separate instance of CamFlow-MW is associated with each process. CamFlow-MW's role is to manage communication on behalf of the process and ensure the enforcement

of IFC policies. Each process can create endpoints that can be understood as typed ports. CamFlow-MW supports client/server (RPC, conversation) or stream-based interaction (one shot, push stream, pull stream). The stream-based interaction allows a *producer* to connect to multiple *consumers* effectively enabling message multicast.

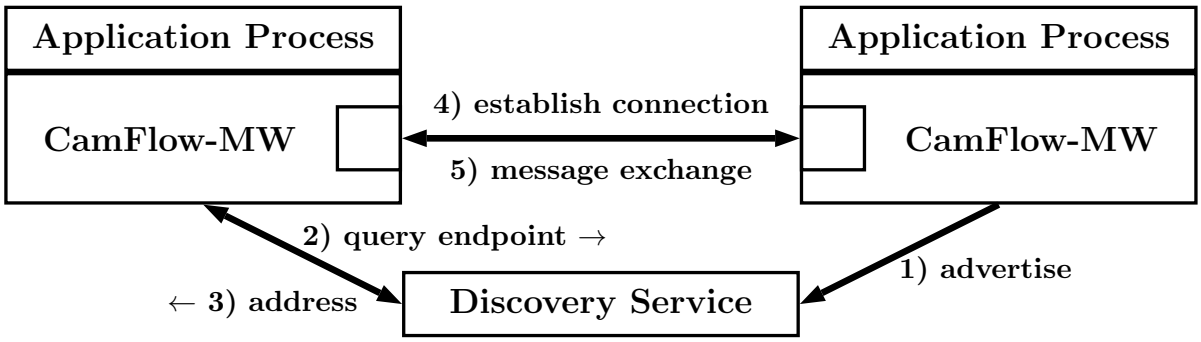

Figure 4.1: CamFlow-MW overview

CamFlow-MW adopts a decentralised peer-to-peer architecture. It provides *endpoints* to applications, which can be understood as typed communication ports. These endpoints and the application metadata are advertised to the discovery service. When a client/consumer wants to find a server/producer it queries the discovery service and is returned the corresponding address. The discovery service maintains access control policy to determine the applications that are allowed to register and query. Further, restrictions can be applied by the advertising process to determine who is able to access its advertisement. A connection is established (in §4.2 we describe the authorisation protocol) between the two endpoints and data can be exchanged.

The middleware encapsulates data in strongly typed messages. The message type mechanism derives from LITMUS [Ingram 2009] that allows expressive description of message types. Each endpoint is associated with a message type (client/server interactions involve two types, one for query and one for response) and connection only occurs between compatible endpoints.

There are potentially a large number of discovery services. They can be federated and replicate information, or alternatively they may only operate within a specific scope, e.g. a particular tenant or application. The discovery query works over two dimensions: the application metadata (e.g. owner, specific application, its class, author etc.), and the endpoints' metadata describing the message type.

The middleware can handle migration or shutdown of remote applications by transparently, from the application point of view, re-establishing the connection with another endpoint matching the query criteria.

4.2 Enforcing Information Flow Control

Connection establishment in addition to verifying the compatibility of endpoints requires the enforcement of security policy. There are three main steps in the secure establishment of a connection:

1. authentication and secure channel establishment;

2. access control policy enforcement;

3. information flow control policy enforcement.

4.2.1 Authentication and secure transmission

In order to protect data and metadata from eavesdropping, the TLS protocol [Dierks and Allen 1999] is used. Both parties exchange certificates to ensure mutual authentication. Here we name these certificates Identity Certificates (IDCs, Public Key Certificates that tie a *subject*'s identity to a private/public key pair) to identify an application instance and associate it with its owner.

4.2.2 Access control enforcement

CamFlow-MW maintains an *access control list* (ACL) for each endpoint associated with an application process. The ACL is generated from the process's configuration, and can be modified at run time by the application process or a trusted remote management system (third party reconfiguration is further discussed in [Singh et al. 2014b]).

The ACL is expressed over endpoint *type* and instance *identity certificate*. The certificates used for access control are those authenticated via TLS in the earlier phase. Further, as discussed in [Pasquier et al. 2015b], the access control can be optionally extended to verify the integrity of the remote platform via remote attestation (RA) [Kil et al. 2009]. RA leverages hardware, such as a Trusted Platform Module (TPM) [Morris 2011], to verify that the configuration and the software stack on the remote machine has not been altered. In our case we are interested in the integrity of the kernel and the IFC enforcement mechanism described in Chapter 3, and the integrity of the MW.

RA and TPM are available in a wide range of platforms from cloud containers/VMs [Berger et al. 2006; 2015] to embedded systems [LeMay et al. 2012] or smartphones [Nauman et al. 2010]. This opens the door for the potential expansion of our approach beyond cloud computing environments. This is discussed further in §9.3.

4.2.3 Information Flow Control enforcement

The enforcement of IFC occurs after the access control, once the MW decided that the remote party can be trusted.[27] CamFlow-MW at both parties' ends retrieve the labels of the application they are attached to. These labels need to be translated from their local kernel representation (i.e. an opaque bit string, as described in §3.3.1) to a global representation that can be remotely verified. Once more, we use standard techniques, in particular X.509 attribute certificates [Farrell and Housley 2002]. This is further discussed in §4.3.

When the labels of the application process are modified, the MW is notified and the IFC constraints of every active connection are re-evaluated. In case a connection no longer accords with IFC constraints, the connection is closed by the MW. Depending on the connection type and the configuration, a discovery query can be sent and connection established to a similar endpoint that accords with IFC policy, transparently from the application process.

Past OS-level IFC implementations [Krohn et al. 2007, Zeldovich et al. 2006] required trusted processes to interface between IFC-constrained application processes and legacy applications such as databases. The MW, as described in [Singh and Bacon 2014], can provide a standardised interface that behaves as a *proxy* between applications and the database, translating different database query types into message types and endpoints.

4.3 Representing tags across machines

We link IDCs with X.509 Attribute Certificates to provide global representation of tags that can be verified. An X.509 Attribute Certificate (AC) certifies some attribute of the certificate's *holder* (the identity to which the AC is tied) [Chadwick and Otenko 2003, Farrell and Housley 2002, Park and Sandhu 2000].

We define *Tag Certificates* (TCs) as ACs that encode tags, i.e. secrecy and integrity attributes, and are bound to a particular identity. Thus, a TC specifies a set of tags and who may use them. A binding (which defines the holder) may be to the owner's identity, e.g. so the tags apply to that person's applications; to another's identity, e.g. another user to allow a third party data access; or to a particular device. There are several approaches for binding ACs (in our case, TCs) to IDCs [Park and Sandhu 2000].

4.3.1 Monolithic signature

A single authority manages both identity and tags. The tags and the IDC are tightly coupled, comprising a single *monolithic block*. Adding or revoking tags requires a new

[27] AC policies are met and optionally the integrity of the remote environment has been verified.

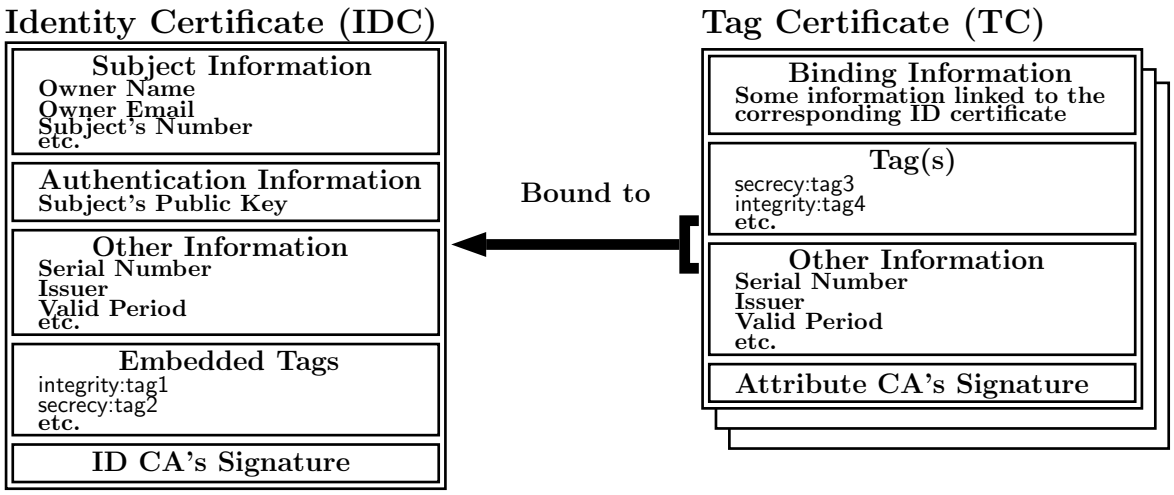

Figure 4.2: An Identity Certificate with a Tag Certificate as Attribute Certificate.

certificate to be issued. This approach is suitable where tags represent an intrinsic characteristic of the principal that is unlikely to change, e.g. a medical sensor generating data tagged as secrecy:medical.

4.3.2 Autonomic signatures

The binding information in the TC directly refers to the IDC certificate's *subject*. The fact that IDCs and TCs are independent adds flexibility to certificate management (revocation, and new assignments of tags to subjects are facilitated). In IDCs, subject information is generally composed of several fields, which together, are guaranteed to uniquely identify the subject, according to PKI policy, as discussed in the IETF specification [Farrell and Housley 2002]. A TC is bound to its related IDC by including in it guaranteed unique information fields about the application's owner. We also assume that *some owner-unique* field(s) may be shared by all the IDC owner's applications, so the same TC can be shared by all applications belonging to that owner. This approach greatly simplifies TC management when a large number of applications is associated with a single owner, for example if Bob wants to tag all his applications with the same tag secrecy:bob-home.

4.3.3 Chained signatures

The TCs are bound to IDCs, using the signature of the IDCs. This creates a strong link between the TC and the IDC, in that a TC only has one IDC, so that revoking the IDC implicitly revokes any TCs. However, it differs from the monolithic approach in that TCs can be independently created/revoked without affecting the IDC.

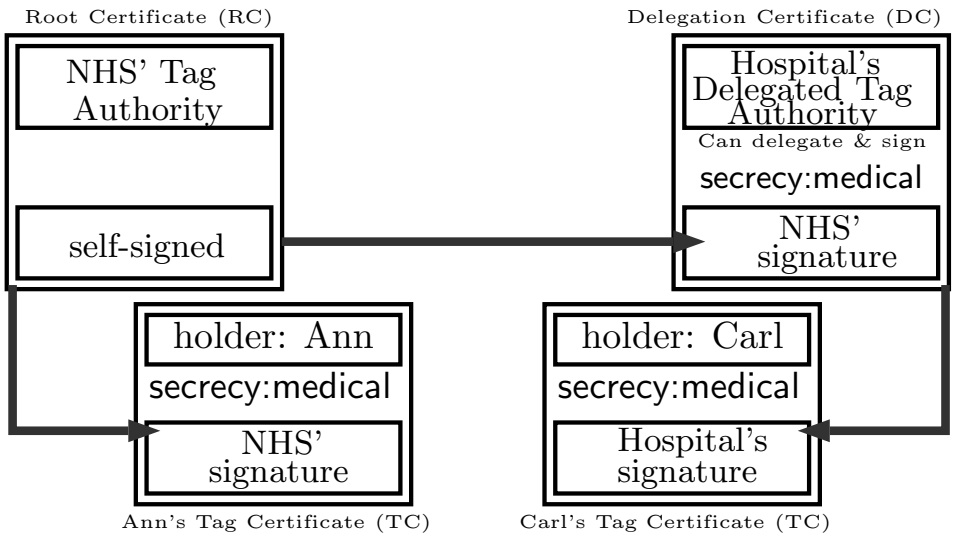

Figure 4.3: Two (simplified) Tag Certificates representing the same tag (same name, valid signature chain with the same root)

4.3.4 Tag ownership and delegation

Conceptually, we define a tag t as a $\{\text{name}, \text{tag-owner}\}$ pair. If a tag $t = t'$, then name = name$'$ and tag-owner = tag-owner$'$. The tag-owner may or may not be the *holder* of the TC. For example, Bob can issue a TC for Ann's application allowing it to receive some of Bob's information flows; though Ann is the holder, Bob remains the tag-owner.

In the proposed scheme, two tags are said to have the same tag-owner if the signing authority's certificate chains have the same root. This means that it is possible to *delegate* the creation of new TCs by building a valid authority certificate chain. This is illustrated in Fig. 4.3 where the TCs held by Ann and Carl have the same root, with tag-owner *National Health Service* (NHS). A hospital is trusted and authorised to issue secrecy:medical TCs for its employees on behalf of the NHS. The Root Certificate is an NHS *self-issued* certificate. The Delegation Certificate (DC), issued to the hospital, is a certificate signed by the Root Authority (the TC's original issuer), that specifies for which tags management is delegated, see Fig. 4.3.

Verifying the TCs' certification chains, in addition to the standard verification of signatures, consists of verifying that every certificate in the chain has been approved to generate the tag or delegate further as appropriate.

Delegation allows TCs to be created for, and transmitted to, authorised parties, e.g. to allow them to receive data. For example, sensor data may be routinely streamed to cloud services for processing and storage. It facilitates the management of tags by trusted third parties where appropriate.

Importantly, because TC creation/revocation does not require trusted CAs, it paves the way for decentralised policy specification and management. This gives control to

individual applications, to have the authority to create and define tags to meet their particular security requirements and to pass them to others without the involvement of a central trusted third party.

4.4 Summary

In this chapter we presented CamFlow-MW. The middleware was previously developed by the Opera Research Group at the University of Cambridge as part of the PAL[28] and TIME-EACM[29] research projects under the name SBUS. In this research we extended SBUS to comply with Information Flow Control (presented in §4.2 and §4.3), and design decisions made for the bridge-usher interface (§3.5.1) were made with the CamFlow-MW implementation in mind. This is, to our knowledge, one of the first efforts to implement DIFC within a distributed messaging middleware.

After describing how local enforcement of IFC is possible on a Linux machine via the implementation of a specific Linux Security Module, and how it can be extended beyond machine boundaries via a dedicated messaging middleware, we next look at how audit data can be generated during IFC enforcement.

[28] `http://palproject.org.uk/`
[29] `https://www.cl.cam.ac.uk/research/srg/opera/projects/time-eacm/`

Chapter 5

Practical Information Flow Audit

IFC enforces policy-compliant use of data, by controlling its exchange between components of a system over the dimensions of *secrecy* and *integrity* [Kumar and Shyamasundar 2014]. IFC complements existing security mechanisms through end-to-end, data-bound security policy.

This chapter discusses, with a particular focus on OS integration, how IFC can be extended to collect audit records that can be used to demonstrate compliance with data handling requirements, through *Information Flow Audit* (IFA). Compliance concerns can be internally or externally imposed on an organisation, company, industry or product, perhaps emerging from contractual obligations (including SLAs), legal regulation, internal policy or industry standards. In cloud services, managing obligations and demonstrating compliance requires the means for monitoring and understanding the circumstances in which data moves between the components comprising the cloud infrastructure.

Provenance systems [Carata et al. 2014, Chapman et al. 2012] concern audit; they assist in understanding the lifecycle of data: *how was it created? when? by whom? how was it manipulated?* As both provenance and IFC concern the flow of information between entities, IFC enforcement is a natural source of provenance-like data. The advantage of IFC with IFA compared with general provenance metadata collection is that in IFC, audit data is a by-product of enforcement, whereby IFC audits only selected (labelled) entities. Further, as the audit data of IFA is intrinsically linked to the control mechanism (IFC), it readily assists policy management, including the identification of errors in policy specification. This creates a feedback loop that allows the alignment over time of policy specification with regulations and laws. This chapter is based on and extends the following published work [Pasquier et al. 2015d; 2016b]. The work remains in progress and the current implementation is available.[30] Illustrative use cases are discussed in Chapter 7. The work remains in progress and the current implementation is available online.[31]

[30] `http://camflow.org/`
[31] `http://camflow.org/`

5.1 Information Flow Audit

Traditional cloud logging systems are mostly based on and composed of legacy and/or service-specific logging systems (OS, web-server, database etc.) These are difficult to interpret system-wide, as they tend to log only those events relevant to the particular system component. As such, it is argued that cloud logging systems should be redesigned to be information-centric (rather than system-centric), thus accounting for the movement of information [Ko et al. 2011]. Further, it has been argued that forensic investigation requires the collection of data that captures the actions of processes, IPC mechanisms and the kernel [Pohly et al. 2012].

IFC complements existing security mechanisms by providing guarantees about policy-compliant data usage. Our aim is to augment IFC with audit that makes visible how the data flows through the system and is used. This allows tenants to effectively demonstrate that proper mechanisms are in place and that all data goes through those mechanisms. If information has been shared when it should not have been, or this is claimed by some party, we aim to provide forensic data to understand how/whether it happened.

5.1.1 Provenance systems

Provenance systems concern audit, associating with each data object metadata describing the transformation involved in generating this data. They typically concern some aspects of: data quality, replication recipes, ownership attribution, context understanding and audit [Simmhan et al. 2005]. Provenance systems generally present the relationship between data objects and transformations (processes) as a directed graph leading to and from the data objects being audited. Such graphs capture when, why, by whom and how this data object was created and/or used and their processing allows such behaviour to be understood. In this work on IFA we use the graphs and processing tools that have been developed for establishing data provenance.

5.1.2 From provenance to Information Flow Audit

IFC constrains the flow of information in a system, being enforced as system components interact. As such, information-centric logging is naturally provided by recording information flow decisions, metadata on the entities involved in the flow and any metadata associated with the decisions. This includes details of data exchanges (e.g. reading from a pipe or file, sending a message), process management operations (such as creating a new process and setting up its security context), and security operations such as declassification or endorsement. This covers the four types of flow described in Chapter 2.

The information generated by IFC enforcement can allow the generation of a provenance-like directed graph, answering the questions: *how, when, where and by whom* a piece of

Node		Edge	
Attribute	**Description**	**Attribute**	**Description**
Entity ID	Unique local identifier of the entity.	Event ID	The ID of the event (e.g. a number, timestamp, ..)
Machine ID	Unique identifier for the machine on which the node was recorded.	Machine ID	Unique identifier for the machine on which the edge (flow) was recorded.
Type	Type of node: e.g. process, FIFO, socket, file etc.	Type	Type of flow: data flow, privilege, creation, security context change.
Name	A name for the node (e.g. filename, executable name etc.).	Sender ID	The ID of the entity from which the data is flowing.
User ID	The user ID of the principal owning the entity.	Receiver ID	The ID of the entity to which the data is flowing.
IFC Labels	Secrecy and integrity label of the entity (and privileges for processes).	Allowed	If the flow was allowed or not.
Additional metadata	Node type specific or user space application-specified attributes.	Additional metadata	Edge type specific metadata (e.g. system call name).

Table 5.1: The attributes of audit nodes (entities) and edges (flows).

information was manipulated. This allows understanding of how a particular piece of information moved through the system infrastructure, across various components and services. Importantly, the tight coupling between the enforcement and audit mechanism facilitates understanding and verifying system behaviour and control policy.

As described in Table 5.1, audit entries can be divided into two main categories: *flows* (i.e. **edges** of the graph) and information about *entities* (i.e. information describing the **nodes** of the graph). A node corresponds to [entity, security context], with a change in security context represented by a *security context change* flow towards a new node. Fig. 5.1 gives an example of how flow of information in the system can be represented. An edge entry is relatively simple: it describes the sender and receiver of a data flow, the type of flow, whether or not it was allowed, and an event identifier for allowing dependencies to be determined. Node entries contain metadata describing the node: its type (e.g. file, socket, process etc.), its ID etc. and again, an event ID so that dependencies can be determined.

5.1.3 Example: discovering data disclosure paths

As discussed, IFA can be represented in a directed graph. The graph can be analysed to 1) trace information flows within, across and between system components; and 2) to examine which components are attempting to violate IFC constraints.

For example, the IFA graph can be used to identify the origin of a data leak. Suppose that an information leak is suspected between different security context domains $[S, I]$ and $[S', I']$. Determining whether such a leak can occur is equivalent to discovering whether there is a path in the graph between the two contexts. If the leak occurred, there must be a path between some entity E such that $S(E) = S \wedge I(E) = I$ and another entity F such that $S(F) = S' \wedge I(F) = I'$.

The existence of such a path demonstrates that a leak is possible. To investigate whether a leak occurred it is essential to consider the event identifier associated with the edges comprising the path. We denote by e_l, the last incoming edge to the entity under investigation with labels $[S', I']$; only edges such that $e < e_l$ should be considered. When applied to all nodes along a path, this rule ensures strictly monotonically increasing event identifiers from the first node to the last.

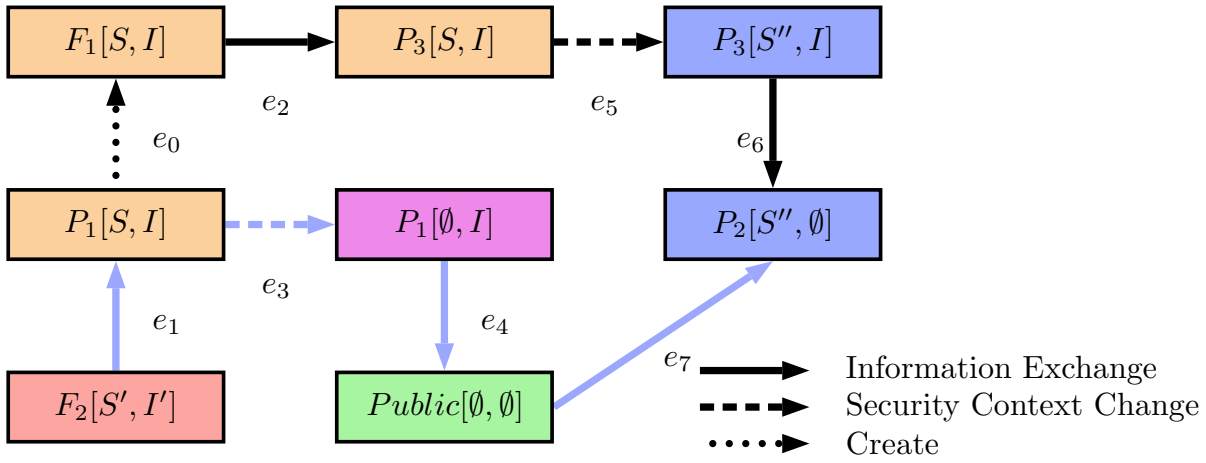

Figure 5.1: Simplified audit graph from IFC OS execution (we omit metadata for readability). Blue/pale arrows show the path to disclosure.

Fig. 5.1 shows in pale blue a possible data disclosure path between $[S', I']$ and $[S'', \emptyset]$ security context domains. We can see from the order of the event identifiers e_0 and e_1 that the data disclosure could not have occurred through file F_1 and process P_3, but occurred through P_1's security context change. P_1 wrote into the public security context domain (represented by a single node as flows are not tracked within this domain) and P_2 read from it at e_7. We present in §5.3.1 how this analysis can be done in practice with our prototype implementation.

5.1.4 Combining IFC and Provenance

One of the problems of provenance systems is the extremely large amount of data being collected, often rendering the approach impractical. In whole-system provenance, data is collected at the granularity of system calls [Bates et al. 2015b, Pohly et al. 2012] (as in kernel-level IFC systems) but IFA records provenance-like metadata only on IFC labelled

entities. "Public" entities are not audited as, from an IFC perspective, the information is not sensitive and therefore can flow freely. IFC essentially aims at labelling sensitive data, which is the data we need to keep track of in practice. IFC audit can be seen as attaching policy metadata to sensitive entities, thus applying a policy filter to select the entities to be audited. In other words, our approach involves a tight coupling between the enforcement and provenance mechanism, which allows a large reduction in the amount of data collected. We argue that much provenance data is excess "background noise" generated by the system, so uninteresting and unrelated to the sensitive data that we aim to protect.

The granularity at which provenance is tracked via IFC audit depends on the IFC enforcement mechanisms employed; CamFlow entails OS object-level and message-level enforcement. Enforcing IFC in a database, for example, would require a specific database implementation, such as IFDB [Schultz and Liskov 2013], where IFC would be enforced at a finer granularity than at the kernel-object level. Different levels of IFC enforcement can be made to interact gracefully, as in [Porter et al. 2014]. As IFC mechanisms are made to interoperate, an API should be provided for (internal) IFA to complement system-wide audit data. Similarly, the metadata collected will vary, according to the IFC enforcement mechanism(s), the applications involved, and higher-level provenance requirements [Simmhan et al. 2005].

This follows the distinction made by Braun et al. [Braun et al. 2006] between observed and disclosed provenance. *Observed provenance* is where the system derives provenance information from events that it can observe, in our case system calls. *Disclosed provenance* is a system where provenance information is provided by applications or users. Our implementation mostly belongs to the observed provenance category, but allows applications to annotate themselves in the graph and bridge-usher to provide provenance information about their internal behaviour or the system they connect to, thus providing disclosed provenance features. This is further discussed in §5.2.2.

5.2 Implementation

The kernel records data flows between kernel objects and the metadata on those objects. These audit entries are then read by an audit-usher. The usher's role is to translate the raw and binary data provided by the kernel into human/machine-readable log data.

A system developer wanting to implement a custom audit-usher needs to implement the callbacks illustrated in Listing 5.1. The underlying concerns (access to log data, threading etc.) are handled transparently by CamFlow-LSM. The current implementation relies on relayfs [Zanussi et al. 2003].

Relayfs provides per-CPU kernel buffers that can be efficiently written from kernel

```c
/* callback to handle edge */
void log_edge(edge_t* e);
/* callback to handle label node metadata */
void log_label(meta_label_t* l);
/* callback to handle string metadata */
void log_str(meta_str_t* s);
/* callback to handle node */
void log_node(node_t* n);
/* callback for filer function */
bool filter(byte_t* raw);

struct audit_op op = {
  .log_edge = log_edge,
  .log_label = log_label,
  .log_str = log_str,
  .log_node = log_node,
  .filter = filter // set to NULL if no filter
};

int main(void){
  register_audit(&op, 4); // register callbacks and number of worker
    threads
  /* do whatever */
  stop_audit(); // stop audit recording
  return 0;
}
```

Listing 5.1: CamFlow audit-usher API.

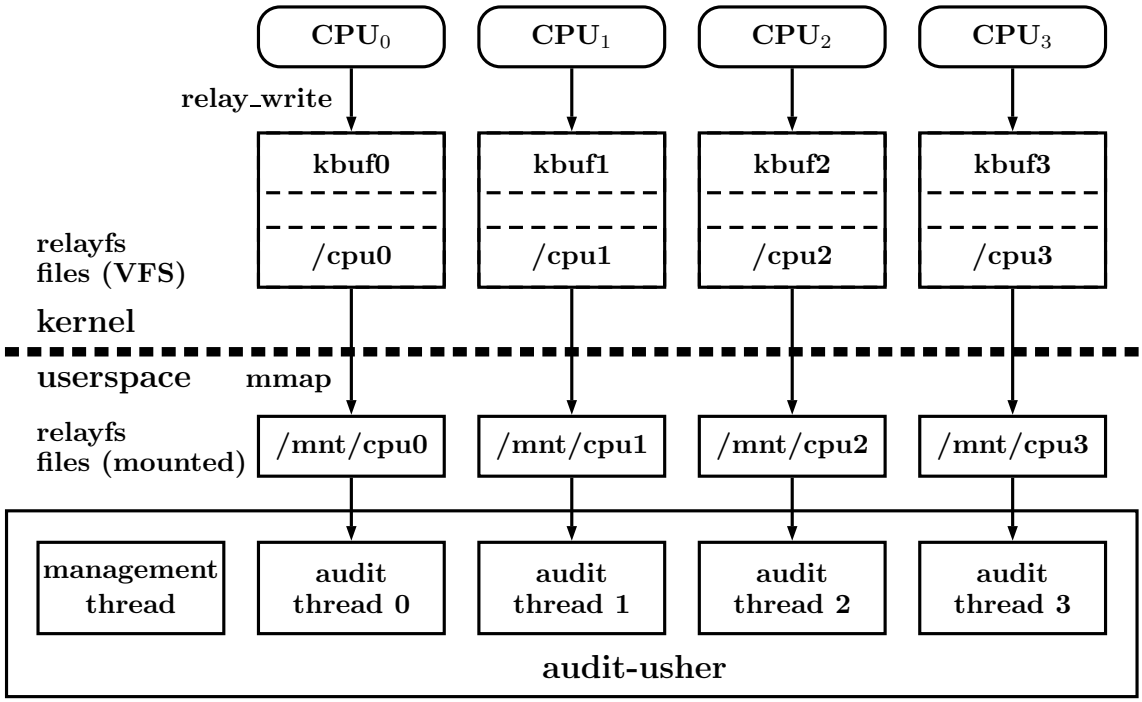

Figure 5.2: Audit-usher using relayfs.

code and read from user space. These buffers are represented as files that the audit-usher can mmap and read efficiently from user-space. Relayfs has been designed to provide the simplest possible mechanism to read and log large amounts of data by relaying them from kernel to user-space. The architecture is illustrated in Fig.5.2.

Customisation of the audit collection allows implementers to tailor the mechanism to their needs. For example, one may want to format the data in accordance with the *Open Provenance Model* [Moreau et al. 2011], feed the data to a graph database, use a graph processing framework to perform real-time event detection, etc. Our proposed implementation does not constrain developers into a particular usage pattern, and while dealing transparently with the underlying mechanisms, allows them to focus on the aspects relevant to them. We describe an example implementation to display an IFA graph through a web interface or to feed the information to a graph database in §5.3.1.

5.2.1 System objects

Processes are the only active entities within the Linux OS (see §2.2.3). Each process is associated with IFC labels and privileges at creation and assigned a unique ID within the current boot (boot and machine are also allocated unique IDs). A process and its memory are treated as a black box. `Fork` generates a create flow from the current process to the forked process. `Exec` creates a data flow from the file being executed to the calling process.

Files, pipes, sockets etc. fall under the *inode* category within the kernel. They are passive entities and their security context is immutable. Creation, reading and writing from those entities are protected by IFC policies and flows are recorded.

Files need to be identified as they persist across boots. A file inode ID is unique within its file system, and a file system is generally associated with a unique identifier at creation. We generate unique identifiers for kernel-internal pseudo-filesystems. The combination of inode ID and filesystem ID allows files to be identified uniquely within our audit logs. Sockets and pipes can be identified in the same fashion as, from the kernel perspective, they are inodes that belong to pseudo-filesystems.

Messages in message queues are handled individually and each message represents a unique node in the audit graph. As they have no kernel source of identifier, their IDs are generated by CamFlow-LSM.

Finally, a file mapped to an address space or shared memory does not provide fine-grained read/write semantics from the audit perspective. We can only enforce and record flows when mapping is established. However, to prevent such a mechanism leaking data across security contexts, once memory has been mapped (in read/write/both mode), the security context of the associated process is frozen (see Chapter 3). Indeed, we conservatively assume that any data accessed by a process mapped to this shared memory flows to other mapped processes. Again, the underlying mechanism relies on filesystems or

pseudo-filesystems, which can be used to uniquely identify nodes within the audit graph.

5.2.2 Bridging with other layers of enforcement

IFC can be implemented across different layers of the software stack, for example, within applications [Porter et al. 2014] or within database systems [Schultz and Liskov 2013]. The CamFlow framework provides a mechanism to bridge from an IFC-constrained process to a trusted process enforcing IFC at a different layer of abstraction (our messaging middleware is such a process and is described in more detail in §5.2.3). A system developer can implement such a mechanism to build more complex systems.

CamFlow associates a bridge-usher process with a constrained process and allows communication through a standard socket interface. This bridge-usher process can perform operations outside the IFC constraints applied to the constrained process. Data sent through the kernel socket is forwarded by CamFlow-LSM from the constrained process to its associated bridge-usher (and *vice versa*). Such messages are recorded and associated with a unique identifier by the CamFlow-LSM module, i.e. logging the flow of information between the bridge-usher and its attached process. As there is "layered" IFC, it is possible to provide layered audit data [Muniswamy-Reddy et al. 2009]. An API allows a bridge-usher to generate an audit subgraph of its internal behaviour and allows incoming or outgoing messages' nodes to be connected to this subgraph. This complements the system *observed* provenance, by *disclosed* provenance from applications [Braun et al. 2006], which provides richer semantic knowledge and allows a better understanding of the system. For example, in the case of a database, this could be providing details of information flow in relation to database objects.

Integration with a system natively supporting IFC is trivial, as IFA is a simple by-product of IFC enforcement. One possible approach to integrate IFC&A with a legacy solution is to use *aspects* for instrumentation. Aspects are used in [Mace et al. 2015] to track data flows in MapReduce/HDFS. We used Aspect Oriented Programming (AOP) to incorporate IFC into web applications [Pasquier et al. 2014b], see Appendix A. Our solution provides the API to insert the data collected at that level into the whole system graph, but instrumentation of legacy applications per se is beyond the scope of this dissertation. This is further discussed in Chapter 9.

5.2.3 Audit across machines

Only processes P such that $S(P) = \emptyset$ (i.e. not subject to security constraints) are allowed to directly connect to or receive messages from outside connections (e.g. through a socket). In order to connect directly to the outside world, a process must either: 1) be able to declassify to $S = \emptyset$; or 2) communicate through a bridge-usher.

An implementation of a bridge-usher is within our messaging middlware CamFlow-MW discussed in Chapter 4. As IFC is enforced within the middleware, the decisions made and their associated data flows can be recorded. This is done through the mechanism described in §5.2.2. The disclosed provenance information generated by the middleware complements the observed provenance graph generated at the OS level. Further, in a distributed system, the disclosed provenance information constitutes the connection between the OS level graphs corresponding to each machine.[32]

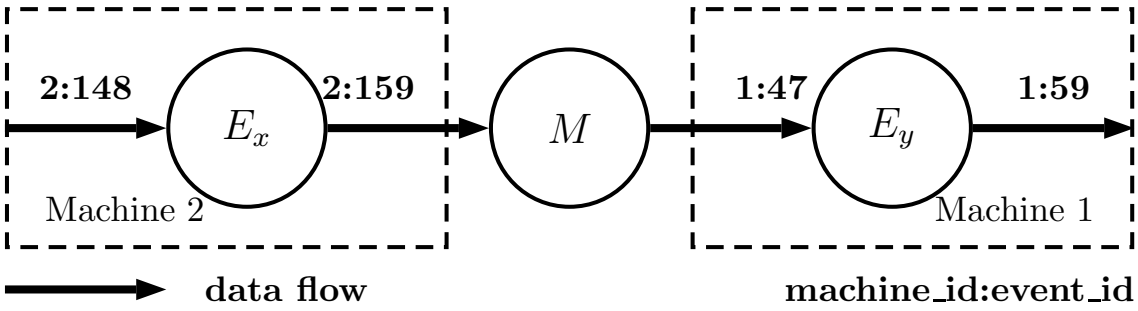

Figure 5.3: Communication through an inter-machine message. Partial order along the path: $2:159 \precsim 1:47$.

In §5.1.3, we discussed how the ordering of event identifiers is used to understand the succession of events. Once the system is distributed one may be tempted to introduce a complex synchronisation scheme to maintain this ordering. However, we argue that it is not necessary and should probably be avoided for the sake of simplicity and runtime performance. Indeed, the flow of sensitive data is allowed across machines only through limited IFC-aware communication channels with well-understood semantics. This creates a partial order of events across machines which is sufficient to order events along *any* given path. For example, as shown in Fig. 5.3, in the case of cross-machine message passing, all writes at the message-sending entity/node happened before any read on the destination machine.

5.3 Evaluation

In this section, we evaluate our solution in terms of practicality and performance. We first look at using our audit framework for data flow visualisation and analysis. In the second part, we run a micro-benchmark, and a macro-benchmark where we compare the performance with whole-system provenance solutions.

[32]We only consider labelled entities and assume, as stated in the introduction to this section, that inter-machine communication only takes place through the middleware. An alternative approach could consider network packet labelling, although this is not yet supported when stacking LSMs [Edge 2015]. This is left for future work and is not part of this dissertation.

5.3.1 Using the framework for Information Flow Audit

In order to evaluate the usability of CamFlow IFA, we now show that the collected data can provide useful insights and can easily be integrated with existing tools. We demonstrate the feasibility of our approach through two simple *audit-usher* prototypes that connect to open-source graph visualisation tools and graph databases respectively. We selected the open-source Cytoscape tool [Smoot et al. 2011] for visualisation and Neo4J[33] for the graph database as they have previously been used in a provenance context [Chen and Plale 2015, Chen et al. 2012, Tylissanakis and Cotronis 2009, Woodman et al. 2011]. The code base is small and relies on off-the-shelf libraries and tools.

Visualising Data Flow: In order to evaluate the feasibility of our approach we built a small tool that reads raw data from the kernel and formats it to generate an audit file. The audit-usher application is very simple, comprising fewer than 100 lines of C code. These log files are then parsed by a Ruby script that builds a graph description in JSON that can be visualised through the Cytoscape tool.

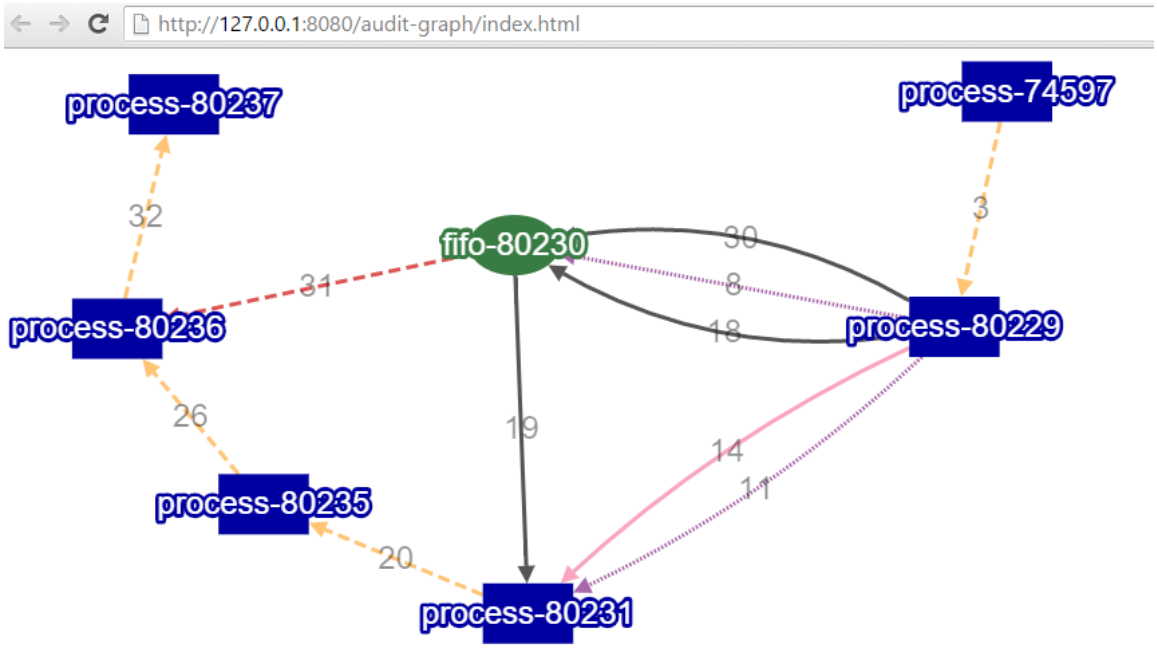

Figure 5.4: Example screen-shot of a small audit sub-graph. Edges' key: orange/dashed—security context change; purple/dotted—creation flow; pink/light—privilege passing; black/plain—allowed data-flow; red/dashed—disallowed data-flow.

Fig. 5.4 presents a sub-graph generated from IFA logs. Nodes are labelled with the tuple {type-id}, where the id uniquely identifies the pair {object,security-context}. Hovering over the nodes displays additional metadata. The following events are represented in the graph: edge 8 shows a parent process process-80229 creating a pipe fifo-80230 and, down

[33] `http://neo4j.com/`

```
// Find all paths on machine "1234" from medical to public
MATCH p =(:Entity {machineid:1234, secrecy: "medical" })
      -[:FLOW*]->
      (:Entity {machineid:1234, secrecy: "public" })
// Restrict to path with monotonically increasing flow event ids
WITH p, range(0,length(p)-2) AS idx, relationships(p) AS rs
WHERE ALL (i IN idx
           WHERE (rs[i]).eventid <(rs[i+1]).eventid)
RETURN p;
```

Listing 5.2: Query (simplified) to find all paths from medical to public.

edge 11, a child process process-80231. The parent passes privileges (edge 14) to the child and writes to the pipe (edge 18), then the child reads from it (19). The child changes its security context (20, 26) and its process ID. Finally, the parent writes to the pipe (30), but the child process-80236 can no longer read from it (31) due to incompatible security contexts.

Analysing the Audit Graph: Our second experiment with the IFA aspect of our framework consisted of pushing data into a graph database in order to perform query and analysis of system behaviour. The implementation of the audit-usher has a small code footprint and easily allows the well-established Neo4J graph database to be used.

Listing 5.2 presents a single-machine query using the Cypher query language.[34] This solves in practice the example presented in §5.1.3. The query searches for all paths between a node in the medical domain to a node in the public domain on machine 1234. The results are a collection of nodes and edges representing the paths between the nodes. These paths can be used to generate subgraphs that represent the transfer of information between the two security contexts' domains. The query presented here, for simplicity, does not deal with node-specific semantics (e.g. shared memory) and is restricted to a single machine (see the discussion in §5.2.3). Such considerations can be either 1) encoded within a more complex query by extending the where clause to deal with entity-specific semantics, or 2) managed through Neo4J's traversal API.[35]

Compliance with regulations can be demonstrated through queries over the graph. For example, the plain English policies:

- *"European personal data sent to the US must be anonymised"* (further discussed in §7.2) is equivalent to writing a query that verifies that there is no path between EU and US labelled data without an encryption process.
- *"Medical data stored in database X must have received proper consent and be anonymised"* (further discussed in §7.3) can be expressed as a query verifying

[34] http://neo4j.com/developer/cypher-query-language/
[35] http://neo4j.com/docs/stable/tutorial-traversal-java-api.html

that there is no path between *medical* labelled data and the database, without a consent and anonymiser process;

In practice, further human input may be required to investigate data leakage or compliance with regulation. This is reasonable given that flow policy will be specified by users, another advantage of coupling enforcement and provenance. The subgraphs generated by a query for a disclosure path may be visualised as described above. In addition, other types of query can be performed over the audit graph such as determining how a particular piece of data has been generated, determining ownership in case of dispute, understanding the cause of a confidentiality breach etc. Exploitation of the type of data we collect creates many opportunities for forensics and demonstration of compliance.

5.3.2 Performance

We tested the CamFlow-LSM module on Linux Kernel version 4.1.5 (08/2015) from the Fedora distribution. The tests were run on an Intel 2.6Ghz i7 CPU and 8GiB RAM machine.

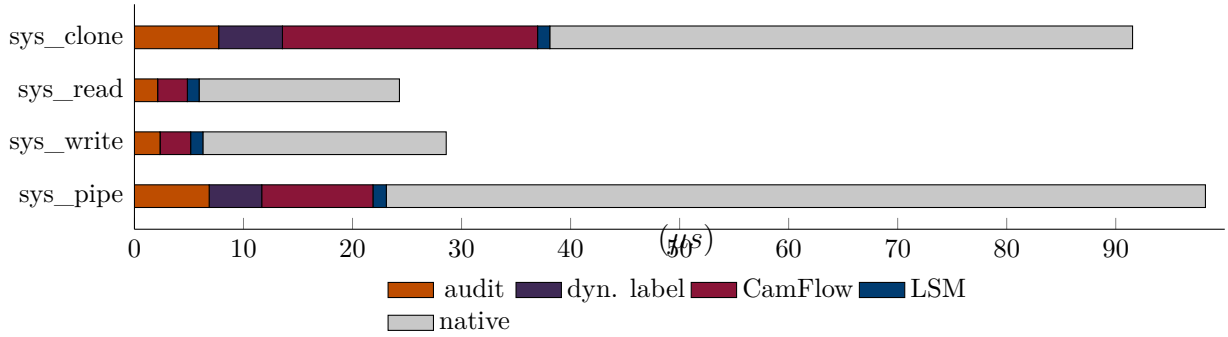

Figure 5.5: Overhead introduced into the OS by the CamFlow LSM

Measurements are done using the Linux tool ftrace [Bird 2009] to provide a microbenchmark, just as in §3.6.2. Two processes read from and write to a pipe respectively. Each has 20 tags in its security label, substantially more than we have seen a need for in current use cases. We measure the overhead induced by: creating a new process (sys_clone), creating a new pipe (sys_pipe), writing to the pipe (sys_write) and reading from the pipe (sys_read). The results are given in Fig. 5.5.

We can distinguish two types of induced overhead on core CamFlow IFC enforcement: verifying an IFC constraint (sys_read, sys_write) and allocating labels (sys_clone, sys_pipe). The sys_clone overhead is roughly twice that of sys_pipe, as memory is allocated dynamically for the active entity's labels and privileges. Recall that passive entities have no privileges. Audit on creation of a new entity is more costly as, in addition to the flow being logged, the new entity and its associated metadata are logged. Overhead measurements for other

system-calls/data-structures are essentially identical, as they rely on the same underlying enforcement mechanism, and are not presented.

The overhead on system calls is in line with IFC [Porter et al. 2014] or provenance [Bates et al. 2015b, Pohly et al. 2012] systems that operate at OS level. For most applications, the overhead imposed is minimal and hard to measure; the deviation between two executions is often greater than the overhead. On kernel compilation, which evaluates a typical combination of process execution and file manipulation, we incur an overhead of 3.6% compared[36] with 2.5% [Pohly et al. 2012], 2.7% [Bates et al. 2015b]. However, these systems only deal with provenance data capture, while our prototype also enforces IFC policies.

5.4 Summary

This chapter presents how Information Flow Control can be extended to provide provenance-like audit data. In particular, we discuss the mechanism to capture the audit data. The proposed approach has the advantage of leaving freedom on how platform level tools are implemented. Audit can be relatively easily integrated with existing open source solutions as demonstrated in §5.3.1. Further, integration of the IFC enforcement mechanism and the IFA audit mechanism reduces the amount of data captured in comparison with whole-provenance systems. Finally, the possibility of providing *disclosed* provenance data from different enforcement layers to complement *observed* provenance is a definite advantage when trying to understand information flow within a complex system.

Now that we have discussed how IFC can be enforced and how provenance data can be captured, we present a proof of concept PaaS built on top of these mechanisms.

[36]Here the values are as reported in their respective publications. Note that the kernel versions are different from ours, namely 3.2.15 (Arch Linux) and 2.6.32 (RedHat) respectively.

Chapter 6

Building web applications

One of the typical usages of cloud computing is the creation of web-services. Further, IFC research [Krohn et al. 2007, Vandebogart et al. 2007, Zeldovich et al. 2006; 2008] (see Chapter 8 for discussion) built a web server to demonstrate the feasibility of their approach. In order to evaluate the solution presented in this dissertation, we build a web server, using container [Soltesz et al. 2007] technology (as motivated in Chapter 1).

In this section, we present the implementation of such a solution built on the infrastructure described in Chapter 3 and Chapter 4. We aim in particular to run unmodified Ruby on Rails applications, similar to those that can be deployed on a platform such as Heroku.[37] This chapter is based on and extends work published in [Pasquier et al. 2014a; 2015d].

6.1 Trust assumptions

The following assumptions were made when building our IFC enforcement and audit mechanism:

Hardware Integrity: We assume that the cloud providers have taken sufficient technical and non-technical measures to ensure that the hardware has not been tampered with.

Physical Security: We assume that best practices are in place on physical access to hardware, when managed by the cloud provider or by a third party managing the underlying infrastructure [Cloud Security Alliance 2011].

Low-level software stack: We assume that the integrity of the low-level software stack is recorded and monitored, which includes BIOS/UEFI, boot loader code and configuration, host platform configuration, virtualisation hypervisor etc. We assume that such integrity measurements are kept safe through a hardware mechanism and cannot be tampered with.

Trusted Platform Module (TPM): We assume that TPM or vTPM [Berger et al. 2006] features are used to guarantee the integrity of the platform on top of which cloud

[37] `https://www.heroku.com/`

hosted applications and service are running. We further assume, that such configurations could be monitored in real-time [Berger et al. 2015] using remote attestation [Kil et al. 2009] to ensure that our security mechanism is in place at all times and is correctly configured. Hardware-assured software is relatively new for cloud services, and further work is needed, e.g., to consider issues such as continuous assessment. Without this, attack analysis may suffer from the disparity between time-of-attack and time-of-detection.

Cryptographic Security: We assume cryptographic functions to be secure and data exchange across machines to be encrypted. We assume that message integrity on exchange between machines can be verified.

6.2 Implementation

Our proof of concept platform (strongly inspired by **Dokku**[38]) is relatively simple and builds upon five open source solutions:

Docker:[39] the container environment powering our platform. Docker is the most popular container environment for Linux platforms and is widely adopted throughout the industry.

Buildstep:[40] a tool that builds docker containers for applications, using Heroku's buildpacks[41]. In our particular scenario it builds a docker container to run our constrained Ruby on Rails application.

Gitreceive:[42] provides a git user on a machine that can receive git push and execute a script on push. In our case, it builds/updates and starts a new container on the platform on a git push.

Httpd:[43] The Apache Foundation web server is extended by a custom module and used as a reverse proxy working as a "gateway" between an external client and the web application running under IFC constraints.

Supervisord:[44] a process management tool often used in Docker to run several processes within a single container. In our context, it is used to start the reverse proxy and spawn the IFC-contained web application instances.

6.2.1 Application architecture

Fig. 6.1 shows an application container and how it fits with IFC elements described in Chapters 3 and 4.

[38] `https://github.com/dokku/dokku`
[39] `https://www.docker.com/`
[40] `https://github.com/progrium/buildstep`
[41] `https://devcenter.heroku.com/articles/buildpacks`
[42] `https://github.com/progrium/gitreceive`
[43] `https://httpd.apache.org/`
[44] `http://supervisord.org/`

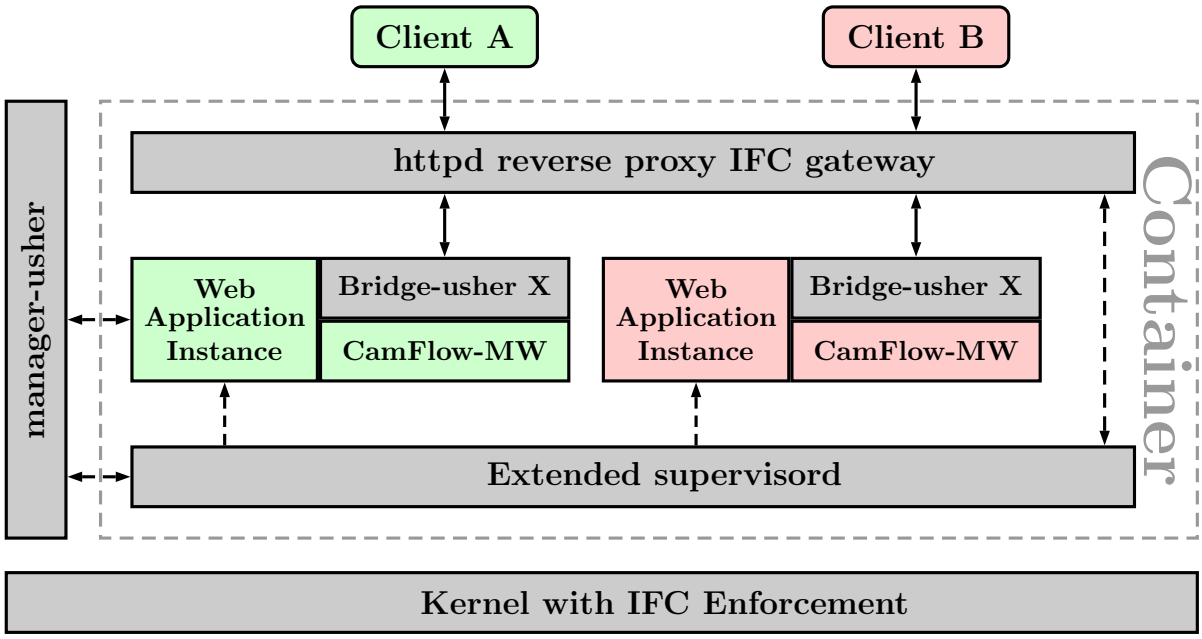

Figure 6.1: Web application architecture.

The httpd reverse proxy gateway maps between each client desiring an application running in a specific security context and an instance of the Ruby web servers. The gateway authenticates the end-user when a session is created and associates this session with an application instance running within the security context corresponding to the user. The proxy gateway and the web application instances interact through dedicated bridge-ushers (bridge-usher X in the figure).

Our platform provides an identity service based on OAuth2 [Hardt 2012] in our proof of concept implementation (not represented in the figure). When authenticating, the user selects an *identity* corresponding to the security context he wishes to use with the application. The reverse proxy retrieves the desired context from the authentication server and binds the client session with an instance running in that context, forwarding all future requests to this particular instance.

If no such instance exists, the httpd server requests supervisord (through the dedicated XML-RPC interface[45]) to spawn a new application instance in the desired security context. Supervisord interacts via the IFC LSM with the manager-usher to obtain a security context token. Supervisord spawns the application instance, providing as parameters the security context token and the Unix socket name on which the bridge-usher X should connect to the gateway. The application instance uses the token to set itself within the right security context and asks to be attached to the bridge-usher X. On connection, the gateway knows that the application instance corresponding to the session has been created and it verifies that it runs within the right security context. When all sessions relating to a particular

[45] `http://supervisord.org/api.html#xml-rpc`

context have expired, the corresponding instance is terminated.

An optimisation is to recycle the process rather than spawning a new process every time a new security context is required. Indeed in [Niu and Tan 2013], the authors suggest using self-checkpointing – using checkpointing and restoration mechanisms [Egwutuoha et al. 2013] – to save the state of the application instance after configuration and before its security context is set up. This "blank state" is restored when a session with an end user is terminated. The application instance then waits for another client for which a new security context is set up.

6.2.2 Building the application

The idea is to build a command line interface similar to that of Heroku. The user wanting to deploy a new application to the platform pushes a git repository to the platform that builds (or updates) the application and starts it.

```
1 git remote add camflow camflow@camflow.me:ruby-rails-sample
2 git push camflow master
```

Listing 6.1: Deploying an application.

Gitreceive is the open source tool backing up this process. The tool calls a script hook that builds (updates) the Docker container on receiving a push and starts/restarts the corresponding docker instance. **Buildstep** is the tool used to build the container corresponding to the application. We modified the default Buildstep container to use supervisord, to contain our httpd reverse proxy and to provide the IFC&A libraries necessary to interact with the software stack described in Chapters 3, 4 and 5. If the container has been successfully built we start/restart the container on docker.

6.3 Example

We run an example in a medical context, where General Practitioners (GPs) connect to a medical portal to access and edit their "patients' medical record". A GP wants to connect to our portal. He is transparently redirected to the authentication server, where he selects his role/identity as Alice's treating GP. The authentication server issues him with a token and redirects him to the application as per the OAuth2 protocol. The gateway verifies the token and retrieves the desired security context $S = [\mathsf{medical}, \mathsf{alice}], I = [\emptyset]$ from the authentication server. The GP session with the gateway is thereafter associated with the corresponding application instance and the interaction within the platform is constrained by IFC. When the GP wants to access applications on behalf of a new patient, he needs to close Alice's session, authorise as treating doctor for Bob and open a new session for Bob.

The enforcement described above is not achieved by the application, but by the platform itself and the security context is defined by the end user, subject to access control. That is, a medical application instanced on Alice's behalf runs in a security context in which data cannot flow to that of another patient. Furthermore, data can be shared between different applications running on behalf of a given user, without the risk of a buggy application leaking data between end-users. The flow of data is not controlled by the applications, but by the platform.

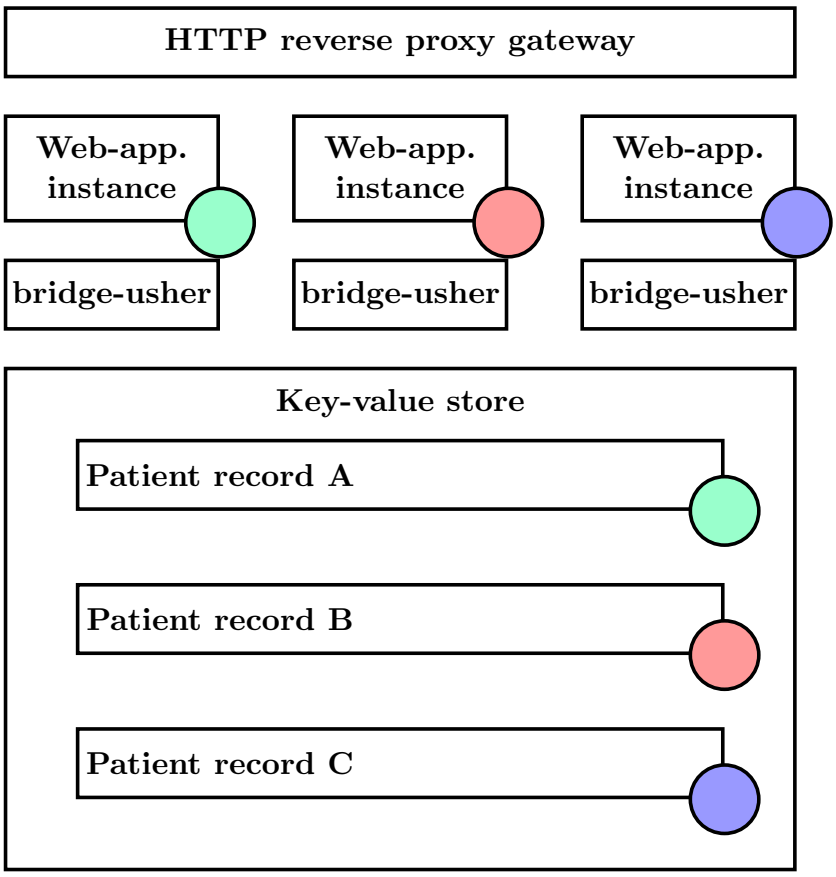

Figure 6.2: The medical portal.

Fig. 6.2 is a simplified representation of the medical portal built as described in §6.2.1. GP requests are directed to the correct instances that correspond to their patients. The patient records are stored in a key-value store alongside their corresponding security context. A request from the web-application passes through a bridge-usher (introduced in §3.5.1). The bridge-usher retrieves the patient record, and forwards the data if the record and the instance's security context match. Audit information is recorded by the bridge-usher as described in §5.2.2. The bridge-usher serves as a link between different IFC granularities, here to a key-value store, but could link for example, to an IFC-enforcing database [Schoepe et al. 2014, Schultz and Liskov 2013]. Here, the backing datastore becomes part of the trusted service which is part of the cloud provider offering.

6.4 Evaluation

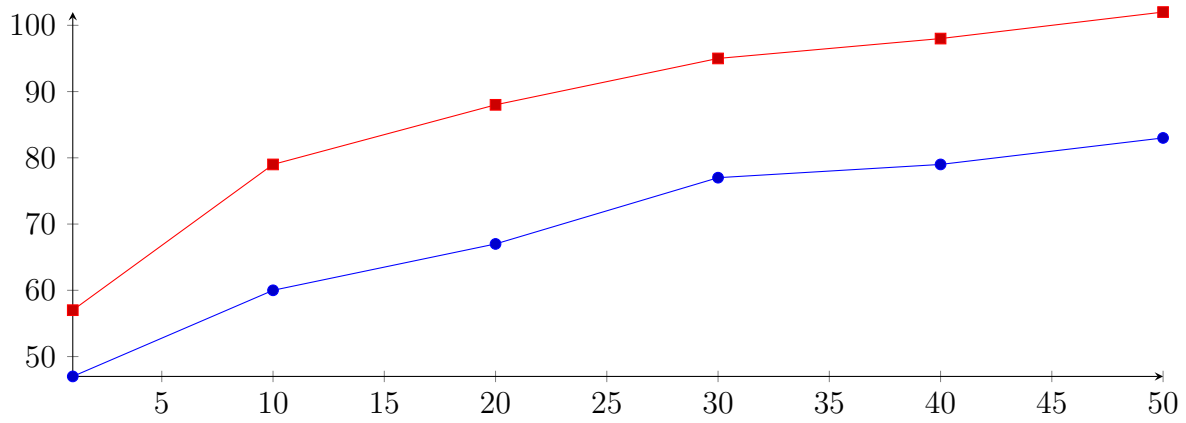

Figure 6.3: Performance of our architecture with IFC enforcement (red/square) vs native Ruby on Rails (blue/circle). Y-axis: latency in ms for 90th percentile, X-axis: number of concurrent requests. Results are averaged over 10,000 requests.

Following the example described in §6.3, an application allows GPs to retrieve the medical records of their patients. We have 50 records of around 9KiB in a key-value store from which records are selected at random. We measure the latency in ms as a function of the number of concurrent requests, see Fig. 6.3. A single security context is used for all requests in order to allow a direct comparison against the native application (both use the thin[46] web server).

The overhead measured for from 1 to 50 concurrent requests varies from 20% to 30%. The overheads measured are of the order of magnitude expected for similar OS-level IFC implementations [Efstathopoulos and Kohler 2008, Krohn et al. 2007]. The proposed example application does little computation, which further accentuates the non-negligible overhead introduced by the gateway and the bridge-usher X. A more efficient implementation, tailored for a specific application, could be provided (for example not using bridge-usher X as in our earlier work [Pasquier et al. 2014a]). However, this may make the proposed approach less versatile and less application-agnostic. The focus here has been mainly on the feasibility of the approach, rather than obtaining good performance.

6.5 Summary

In this chapter, we demonstrated through a proof-of-concept implementation that it is possible to build a simple PaaS that leverages IFC&A. The current platform has limitations: performance is not good; it runs on a single node; and integration with services is done through a bridge-usher. However, these are engineering and human resource issues, rather

[46] `http://code.macournoyer.com/thin/`

than matters of principle. We believe the approach taken to be extensible to more complex PaaS environments,[47]. Further, we believe this work to be the closest to a practical solution built upon an IFC system.

After describing the underlying mechanism to enforce and audit IFC, and how it was possible to build a cloud PaaS on top of it, we now need to investigate how IFC&A can be used to specify and demonstrate compliance with regulations.

[47]Which is not to ignore the potential challenge of building more complex scenarios, and we acknowledge (see Chapter 9) that more work is needed in this area.

Chapter 7

Towards the enforcement and auditing of complex policies

We demonstrated through a proof-of-concept implementation, in Chapter 6, that it is possible to provide a practical Information Flow Control and Audit (IFC&A). This chapter discusses how this mechanism can be exploited to enforce and demonstrate compliance with complex policy.

First, it is important to understand the roles of the actors involved in the handling of personal data. The UK Information Commissioner defines:[48]

1. the data controller as *"a person who (either alone or jointly or in common with other persons) determines the purposes for which and the manner in which any personal data are, or are to be processed"*;

2. the data processor *"in relation to personal data, means any person (other than an employee of the data controller) who processes the data on behalf of the data controller"*;

3. processing as *"obtaining, recording or holding the information or data or carrying out any operation or set of operations on the information or data"*.

Data protection laws require the data controller to comply with principles such as justifiable data processing or implementation of proper security measures. Further, some data may be considered particularly sensitive, for example medical data that requires explicit consent for it to be released for processing.[49] The data controller must retain the responsibility for the proper handling of data. The processor must only process the data in the context specified by the controller and must not retain the data beyond the time

[48]UK Information Commissioner's Office – Data controllers and data processors: what the difference is and what the governance implications are – 2014.
`https://goo.gl/FzUMf3`

[49]Article 29 Data Protection Working Party 2015

necessary for the purpose of processing. Further, the data processor should contractually be required to implement proper technical or organisational security/privacy measures.

However, as pointed out in [Flittner et al. 2016] the data controller (i.e. the tenant) has very little control or view of the data processor's internal working. This puts data controllers in an extremely uncomfortable position where they bear large legal responsibilities, with few technical means to ensure their requirements are being properly implemented by the cloud provider. This state of affairs explains the reluctance to adopt cloud computing in some heavily regulated sectors. IFC&A is a means for a data owner to:

1. express the desired workflows that data must follow in order to respect regulation;
2. provide a clear view of how the data are being handled within the cloud.

This chapter explores three examples taken from real-life regulation or recommendations by data protection agencies. These examples concern data geolocation, release of patient data for medical research and electricity smart metering. These were identified through collaboration with academic lawyers within the Microsoft Cloud Computing Research Centre[50]. The legal context of these use cases is briefly introduced with reference to the relevant legal text, and how IFC&A is used is then discussed. This chapter builds on the following work: [Pasquier and Eyers 2016, Pasquier and Powles 2015, Pasquier et al. 2016a, Singh et al. 2014a; 2015c].

7.1 General overview

As discussed in Chapter 2, declassifiers and endorsers are trusted entities that perform some operation on the data (e.g. analysis, transformation, etc.) and change its security context when the operation has executed successfully, transferring information across security contexts. These entities are allocated *privileges* that allow them to change their security context in order for data to be transferred from one context to another, see 2.2.3. As mentioned, IFC allows untrusted applications to run on top of the enforcement mechanism, and declassifiers and endorsers may be small, tightly-scoped elements of a trusted computing base that generally perform a one-way, well-defined task (e.g. encrypt, anonymise, etc.). They can be seen as security micro-services that are well separated and independent from application logic.

Endorsers and declassifiers can therefore be seen as trusted gateways between different security contexts, where the general IFC constraints would prohibit a direct flow. Such gateways, when accompanied by audit, can help ensure that regulation is enforced, e.g., medical data might only flow to a research context if it has gone through a declassifier that applies a well defined anonymisation algorithm (more on this in §7.3). IFA can be used to

[50] `http://www.mccrc.eu/`

demonstrate that no other path exists (see Chapter 5). Similarly, regulations may indicate that medical data must be encrypted before being stored in databases. IFC labelling and endorsement can ensure compliance and IFA can demonstrate this compliance.

IFC aims more generally to provide secrecy and integrity security primitives bound to data flows. Here, complexity and expressiveness emerge from the interaction of these primitives with the building blocks provided by the trusted declassifiers/endorsers. This removes the need of every participant in the system to be trusted; instead, trust is only placed in the underlying IFC enforcement and the declassifiers and endorsers. That is, IFC guarantees that a certain workflow is followed (i.e. a medical $\rightarrow$ research flow must go through well-defined processes). The policy can be encoded in small endorser/declassifier services. These microservices associated with IFC tags could be provided by the tag owner (here we assume the tag owner is the owner of the associated data) or the cloud provider (considered as a trusted third party by all participants). IFC constraints guarantee that these transformations are applied before data is allowed to flow between certain components of a system. The combination of IFC policy and declassifiers/endorsers allows the enforcement of policies such as those described in §7.2 and §7.3.

We assume an error-free IFC implementation. Only IFC constraint enforcement can be guaranteed. Complex policy needs to be expressed with care and verified. The audit mechanism introduced in Chapter 5 allows a feedback loop to be created that allows policies to be corrected, in line with the preferences, regulations and laws that inform them. Indeed, the captured audit data allow a detailed understanding of the system to be built, as well as implications of the policies. These policies can then be modified in light of this deeper understanding. Further, compliance with regulations or contracts can be challenged in court, and the interpretation of the regulatory and/or contractual constraints can be subject to interpretation until a clear jurisprudence is established. It is therefore vital to be able to understand how the policy affects the system.

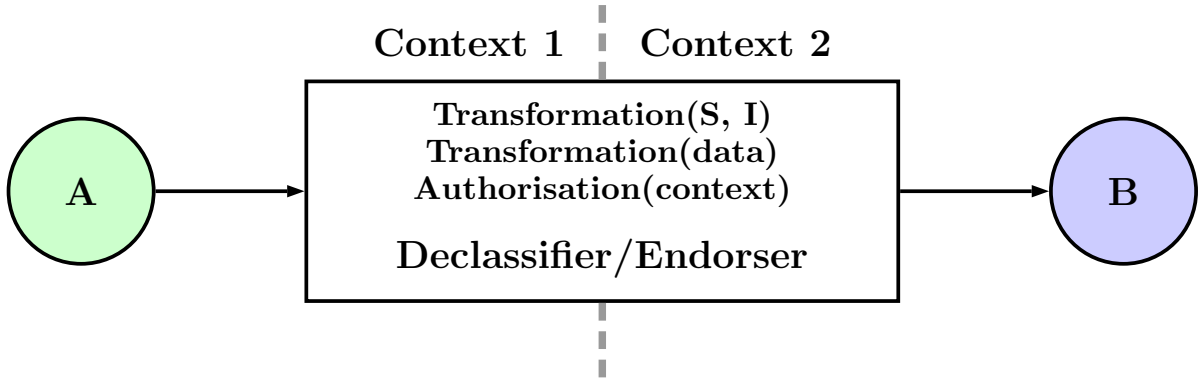

Figure 7.1: Endorser/declassifier micro-services as building blocks for complex policies.

Fig. 7.1 represents the basic behaviour of a declassifier/endorser micro-service. The

main purpose from an IFC perspective is to apply a function to transform the label. For example, applying the transformation $D[S, I] \rightsquigarrow D[S \setminus \{medical\}, I]$ using the notation introduced in Chapter 2, indicates declassification over *medical*. In addition, the micro-service applies a transformation to the data. For example, anonymising, encrypting or in some cases applying the identity function. However, these two transformations are only applied after authorisation has been carried out (the authorisation could obviously be always true if appropriate). This authorisation is carried out over the *context* of the data flow.

How this context is represented and interpreted may vary depending on the policy the security micro-service is enforcing. It may be based on attributes of the message (see Chapter 4), sender and receiver akin to attribute-based access control (ABAC) decision [Hu et al. 2015]. For other policies, it may depend on the provenance of the data and use the graph built by the IFA audit system (see Chapter 5) in a manner akin to provenance-based access control (PBAC) decision [Park et al. 2012].

The mechanism presented here allows the enforcement of relatively complex policy across untrusted parties in a fairly decentralised manner over an IFC-enforcing platform. Tools to manage, facilitate and automate the mechanism remain future work, and are discussed in Chapter 9. We now explore this concept through use cases.

7.2 EU data geolocation

Regulations concerning the geographic location of data are familiar in data protection law, particularly in the EU.[51] There are various reasons for a *cloud provider* to ensure the geographic location of its *tenants'* data, and reliably represent the origin of any data that may be processed in a given cloud. This section explores some of these aspects.

7.2.1 Brief summary of legal issues

The overall motivation for both data location requirements and their technical responses is to help establish a degree of certainty regarding applicable law, and therefore minimise compliance and litigation risk. Through the use of IFC, we demonstrate one way in which technical mechanisms can assist in enforcing well-specified policies. This might be in conjunction with, or as a reinforcement of, contract and certification-based mechanisms.[52]

Different aspects of regulations with implications for data location can be summarised as follows:

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

[52] Article 29 Data Protection Working Party, 05/2012 Opinion on Cloud Computing.

1. Some countries restrain the processing, storage and caching of data originating in that country to certain well-defined locations. For example, the European Union states that sensitive and confidential information should stay within its borders or in certain Safe Harbor destinations that are parties to the EU Data Protection Directive 95/46/EC. Following the Edward Snowden revelations, a possible 'EU cloud' has been discussed, with analogues in BRICS and other countries [Hon et al. 2014, Singh et al. 2014a].

2. Some countries explicitly state their right to access and intercept foreign data within their borders in order to preserve their security, economic or scientific interests.[53][54][55] Often this is a matter of executive discretion, based on fluid concepts of national security and interest, and may not be subject to rigorous democratic oversight and judicial safeguards. In order to maintain the trust and custom of companies/individuals, greater certainty regarding where data are processed, stored and cached is desirable.

3. Increasingly, nations are claiming the right to prosecute and investigate foreign companies that process the data of their citizens beyond their borders.[56] However, it should be possible for a company to be able to isolate information belonging to, for example, US customers from that belonging to EU customers. If the company is not able to clearly separate the two it may be forced to release both.

4. There is a growing concern in Europe over third party use of data, particularly regarding advertisement or recommendation systems. The developments in relation to the so-called 'right to be forgotten' have led to European data protection authorities requiring that US-based companies enforce EU law across global services.[57] This could potentially mean that data should be processed differently depending on their origin. For example, we could imagine that in a not too distant future, data used in conjunction with advertising-based services in Europe must go through a differential privacy algorithm before being used.

At present, such concerns are not enforced continuously and systematically by technical means. Therefore, data mismanagement practices and scandals tend only to be revealed after the fact, and presumably there are others that never reach the public eye.

[53] Loi de programmation militaire, 2013.
[54] USA PATRIOT Act, 2001.
[55] UK Data Retention and Investigatory Powers Act, 2014.
[56] Microsoft Corp. v. United States, No. 14-2985, am. notice of appeal 2nd Cir., 2014.
[57] Article 29 Data Protection Working Party, 2014.

7.2.2 Enforcement

The legal concerns can be separated into two broad categories: (1) concerns about where data is authorised to flow to; and (2) concerns about where data comes from. These two concerns are explicitly and accurately captured by IFC policies. Where data can go to is represented by *secrecy* labels and where data can come from is represented by *integrity* labels.

An international, US-headquartered cloud provider wishes both to comply with European law and to protect the privacy of its US customers' information. One of the steps towards achieving this is by enforcing a policy that EU customers' personal information should not leave the boundaries of the EU. This is achieved in our IFC system by labelling all EU data,[58] and all entities within the EU that process or store EU data, with the *secrecy* label $S = [\mathsf{location\text{-}EU}]$. Such labelled information cannot leave the EU and, with a proper audit log, the provider could demonstrate its compliance by presenting data flow records to an auditor.

One of the cloud provider's US users might raise the concern that, if the US user becomes legally implicated in Europe, the provider may be forced to disclose information that includes this US user. However, the US user does not operate outside the US, and it is only through their use of the international cloud service that such data is even potentially vulnerable to European authorities. To avoid exposing the US user to European authorities, the cloud provider decides that all information from this client will be marked as $S = [\mathsf{location\text{-}US}]$ and processed within the US by processes labelled in the same way. Not only does this guarantee to the US user that such processes are located in the US, but also that they will only process US data, therefore reducing the risk that a foreign authority will be able to request such data in the absence of overreaching extraterrorial application of law, or by warrant. Through IFC, the provider will be able to demonstrate through the audit log that data has been exclusively dealt with within US territory, and that there has been no processing, storage or caching in Europe.

Similarly, *integrity* constraints can be used to specify the location from which a service is willing to accept data for processing. We have seen that there is a risk that processing data from a certain location may expose cloud providers to scrutiny by the corresponding jurisdiction of *all* the data processed by the service. In the example, the US provider, wishing to limit the exposure of its non-US users, may enforce an integrity constraint to ensure that its processing of European data, for example, is not in contact with data originating in the US, and that the outputs of its European processing do not derive from any US input data. This can be further guaranteed by imposing 'separation of duty' policy

[58] We assume that entities are labelled correctly according to their geographical position. A hardware mechanism such as presented in [Jayaram et al. 2014] can be leveraged as a source of trustworthy label. Such considerations are further discussed in Chapter 9.

as described in §2.2.5, guaranteeing that US and EU data are processed separately.

By setting both *secrecy* and *integrity* tags on a given application to reflect the geographic location under which a service operates, or under which data has been generated, we can provide transparency and assurances about geographic location of data.

IFC policy can be used to guarantee that certain paths are followed, and certain transformations are applied, before information reaches its destination, thus constraining the data to well-defined workflows. For example, a cloud service could provide storage and make no assumptions about the data being stored there. Such a store will be labelled $S = \emptyset$. European information labelled $S = [\mathsf{location\text{-}EU}]$ could not flow there. A declassifier based in the EU could potentially provide *declassification* from $S = [\mathsf{location\text{-}EU}]$ to $S = \emptyset$, by anonymising the information, for example. If this was implemented, the IFC policy would ensure that unless the data has been anonymised, it cannot reach this location.

Similarly *endorsement* can be used to prevent a service from exposing itself to other jurisdictions. Indeed, an endorsement from $I = [\mathsf{location\text{-}EU}]$ to $I = [\mathsf{location\text{-}US}]$, through the anonymisation of the data set, would allow the US service to reduce exposure to liability for manipulatating EU personal information, assuming that anonymisation is sufficient in legal terms.[59]

IFC can guarantee non-interference of data from different geographic locations, providing tenants with stronger guarantees from the jurisdictions under which they are operating. The requirement of declassification/endorsement for cross-location data flow ensures that decisions requiring such exchanges are explicit. All cross-location data exchanges become either intentional or are prevented from occurring at all. Further, the audit mechanism may assist in determining complex jurisdictional issues about the applicable law and attendant obligations.

7.3 Release of medical data for research

Personal health data are intrinsically sensitive. However, their exploitation by researchers or policy makers is essential, and greatly beneficial to society. In order for medical data to be used in research and as a basis for policy, there are generally strict governance frameworks specifying requirements such as informed consent, anonymisation processes and appropriate ethics.

[59]This will depend of the nature of the handled data, the disclosure purpose, and needs to be considered within a specific context.

7.3.1 Brief summary of legal issues

The UK Medical Research Council (MRC) has complex research procedures to allow researchers access to medical data.[60] Requirements differ for access to identifiable and non-identifiable information. In this example we focus on the process to disclose non-identifiable information. The MRC makes the distinction between pseudonymised information and anonymised information.[61] Pseudonymised information allows data to be re-linked to a patient if the corresponding key is known, while anonymised data does not allows such a link to be made.[62]

In the rest of this section an *anonymiser* refers to a process carrying either pseudonymisation or anonymisation.

In order for data to be released, the purpose, the nature and the anonymisation process to be used must be carefully examined in order to determine if consent is required.[63] When applicable, consent must be ensured and demonstrable.

7.3.2 Enforcement

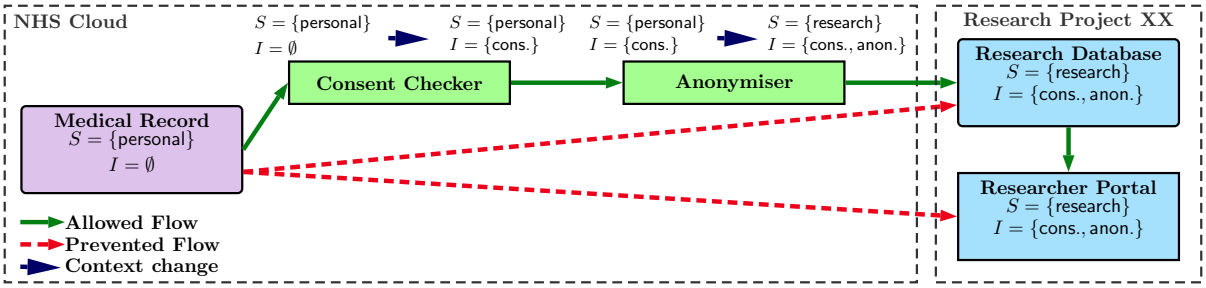

Figure 7.2: Medical data declassified and endorsed for research purposes (figure presented earlier in Chapter 2).

IFC&A can help ensure and provide evidence that these constraints and requirements are respected. For example, Fig. 7.2 illustrates a simplified scenario concerning consent and anonymisation (in practice, more checks may be necessary) where:

1. A consent checker verifies if consent is required, based on data type and research purpose. In such a case, the patient consent is thereafter verified.

[60] `http://www.mrc.ac.uk/documents/pdf/obtaining-hscic-data-guidance-v041115/`

[61] Medical Research Council: `http://www.dt-toolkit.ac.uk/routemaps/station.cfm?current_station_id=415`

[62] There exist techniques (e.g. [Benitez and Malin 2010, Cassa et al. 2008, Kifer 2009, Malin and Sweeney 2004]) for the potential re-identification of anonymised data. The success of such techniques is dependent on the attacker's background knowledge and the anonymisation algorithm used [Narayanan and Shmatikov 2010]. However, anonymisation algorithms destroy some information and the quality of the anonymisation needs to be balanced against data utility [Brickell and Shmatikov 2008].

[63] Medical Research Council: `http://www.dt-toolkit.ac.uk/routemaps/station.cfm?current_station_id=427`

2. An anonymisation process (Anonymiser) takes personal medical data as input, producing an anonymised (according to some approved algorithm) version as output.

3. *Integrity* constraints ensure that the Anonymiser can only receive data where consent criteria are met for its use in medical research (i.e. with the tag consent).

4. The anonymiser outputs data with an *integrity* tag (anon) marking that the data has gone through the designated anonymisation process. The (personal) *secrecy* tag is removed, given the changed level of sensitivity.

5. The Medical Research Database is tagged such that it will only receive—or be willing to accept, e.g. for reasons of liability and responsibility—data that has been anonymised, and where consent criteria had been met. In practice, an individual research project should be issued specific secrecy tags to ensure the respect of the data disclosure purpose [Kumar and Shyamasundar 2014].

6. Individual researchers/projects are bound by the same IFC constraints as the Medical Research Database.

The only way for personal data to flow to researchers is through a well-defined workflow containing a designated anonymisation process. As all flows are recorded, this can be audited. Audit also allows patients who gave consent to see whether their data has actually been staged for research use. The audit mechanism, presented in Chapter 5, can also help in asserting that policy on retention time (e.g. data has been deleted and therefore not used after the lifetime of the project) or proper handling of data has been enforced. This could be particularly useful if regular accreditation inspections must take place, and may provide useful means to verify that proper procedures are followed at all times.

7.4 Electricity smart meters

This section explores the use of Information Flow Audit (IFA) for demonstrating compliance. The use case explored derives from a CNIL[64] report. The report[65] describes best practice for smart metering services used for electricity supply, including those mediated by the cloud. These scenarios and best practices could apply to products such as Nest thermostat devices.[66] While previous sections focused mostly on the enforcement aspect, this section focuses on the audit aspect.

[64]the French Data Protection Agency.

[65]Pack de conformité sur les compteurs communicants, published in May 2014, available at `http://www.cnil.fr/fileadmin/documents/Vos_responsabilites/Packs/Compteurs/Pack_de_Conformite_COMPTEURS_COMMUNICANTS.pdf`.

[66]`https://nest.com/`

7.4.1 Brief summary of legal issues

This section highlights the $IN \rightarrow OUT$ and $IN \rightarrow OUT \rightarrow IN$ scenarios described in the report. In the first scenario data are collected and processed in the cloud to provide services to the customer. In the second scenario, in addition, actuation commands may be *sent by and through* a cloud application to devices situated in a customer's house, in order to control energy consumption. Commands *sent by* a cloud application cover occurrences where the decisions are made by cloud applications, potentially relying on cloud-stored historical data. Commands *sent through* the cloud cover occurrences where interactions between the end users and the smart metering system are mediated by a cloud application.

We extract four recommendations from the report:

1. anonymous data can be freely transferred to a third party;
2. personal data can be transmitted with explicit consent;
3. when a contract is terminated, data must be deleted, anonymised or archived (archiving is for litigation purposes and limited to a duration specified by law, and archived data should not be used in a commercial endeavour);
4. detailed consumption and actuation data can be conserved for three years, but must be aggregated after this period.

7.4.2 Enforcement

We assume that compliance data is stored in a graph database and can be queried as discussed in Chapter 5, for example, to obtain paths between *sources* and *destinations*.

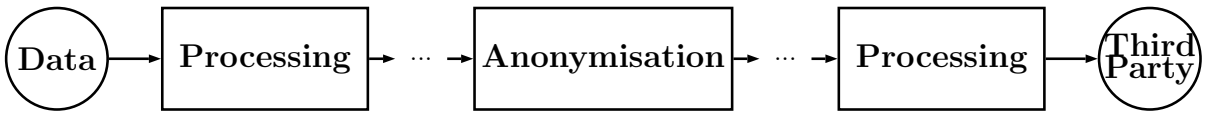

Figure 7.3: Disclosure path to a third party.

Recommendation 1: the sources are the customer devices, and the destinations are third party services. Verifying compliance with the first recommendation is equivalent to writing a query to find a path – between a customer device and a third party service – that does not contain an anonymisation process. If such a path exists, the data processor is in violation of the recommendations. An illustration of a path following the recommendation is presented in Fig. 7.3.

Recommendation 2: As discussed in §7.3, consent verification can be handled by a specific component. Again, if a path that does not contain a component to verify consent can be found, the data processor is potentially in violation of the recommendation.

Recommendation 3 and 4: queries can be made to verify that data used after three years are only in their aggregated form (i.e. there is no path between a commercial process and a data event older than three years without an aggregation procedure). Similarly, the use of data after contract termination can be verified, given the date of the termination is known.

Further, as the 'Heartbleed' vulnerability [Durumeric et al. 2014] demonstrated, no implementation is guaranteed to be error proof, even if it is widely deployed, tested and examined. Regulators often stipulate that best practice and a state-of-the-art approach must be used, as appropriate to the sensitivity of the data. As an example, this may mean verifying that no software library version impacted by the Heartbleed vulnerability is loaded by applications, after a reasonable period of time following discovery of the vulnerability. In our current use case this could mean that, in addition to the existence of the anonymisation procedure, we may want to verify the algorithm and implementation version that it uses. The audit graph may also facilitate:

1. the identification of data that has been processed by the buggy implementation;
2. the determination of the customers impacted by the vulnerability.

The data processor may be required to notify the affected customers, as for example, mandated in the US-CERT guidelines.[67]

Events in the system are represented by edges in the audit graph. For example, an actuation command is a flow (or a succession of flows) between some entity and another. If an actuation command causes physical or financial damage it may be necessary to determine who is responsible. Was the algorithm used to issue the command erroneous? Were the data captured to reach the decision inaccurate? Were errors introduced in the chain between the decision to actuate and actuation? In order to answer such questions, it is first necessary to identify the system components and persons involved in the process. In the presence of a complex ecosystem, where multiple devices' manufacturers and service providers interact, this may not be a trivial task. Indeed, an actuation command should not be seen in isolation, but as the result of a potentially complex chain of events linked by causal relationships. Query of the audit graph allows this complex chain to be visualised, analysed and understood. Which in turn, can help in determining where responsibility lies.

Fig. 7.4 presents a partial graph[68] leading to a command that caused a fire, damaging the customer's property. While in itself it may not be sufficient to determine responsibility,

[67] `https://www.us-cert.gov/incident-notification-guidelines`

[68] This graph, in addition to active entities (e.g. a computer process) and passive entities (e.g. files, sockets, pipes etc.), also represents *agents* (i.e. a contextual entity acting as an enabler, catalyst or controller of a process execution) and *artefacts* (i.e. immutable digital or physical objects) as defined by the Open Provenance Model [Moreau et al. 2011]. Agents may be natural or legal persons, that is respectively a "real" human being or a legal entity such as a company.

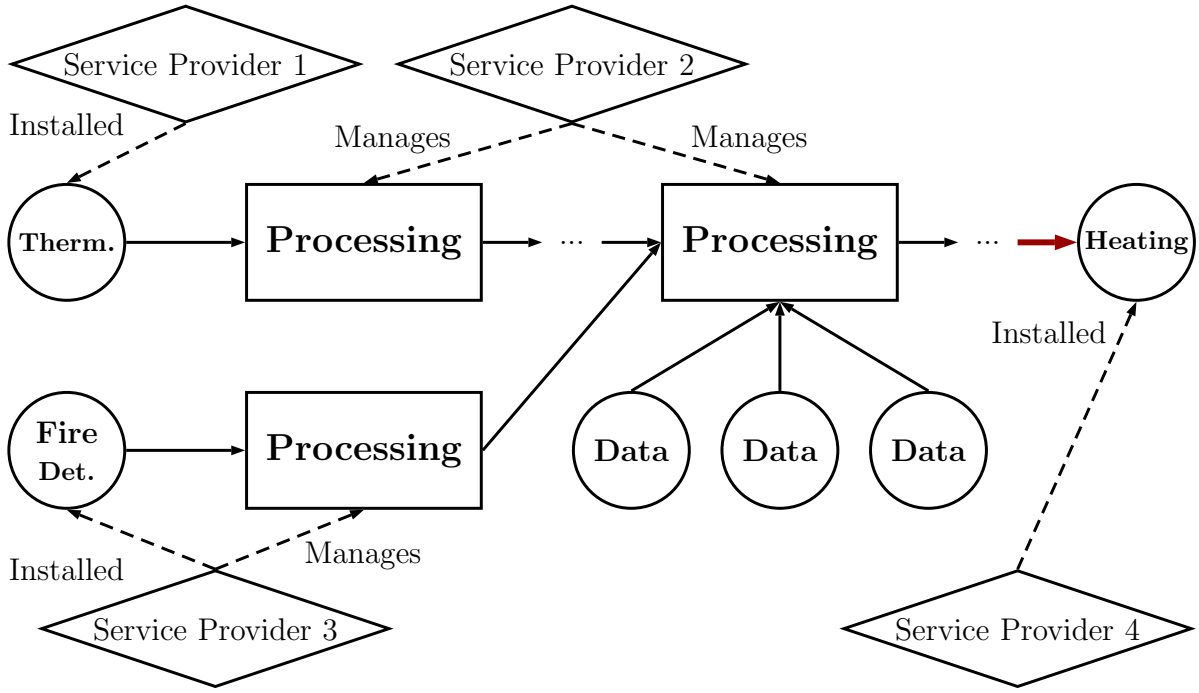

Figure 7.4: Partial graph leading to a command causing a fire (red edge).

it allows all parties involved to be identified and their participation in the chain of events to be explored. This has great potential in facilitating the investigation.

7.5 Summary

In this chapter we discussed how Information Flow Control and Audit can help in ensuring that regulation requirements are followed, and can provide audit data to help in investigation, accreditation renewal or more generally to improve transparency. We explored three different use cases, extracted from Data Protection agencies' regulations or recommendations.

The work presented in this dissertation focuses on building the core underpinning mechanisms of an IFC-based system. Higher-level mechanisms to facilitate the implementation of such policies are beyond the scope. However, future research directions are discussed in Chapter 9, and show how this work is a first step towards ongoing work to build systems that provide demonstrable compliance with regulation.

Chapter 8

Related work

This Chapter explores data/information-flow-centric techniques proposed for the protection of sensitive data in complex systems. In particular, Information Flow Control, Taint Tracking and Provenance mechanisms are discussed. Approaches such as encryption (e.g. [Catrein et al. 2012]), access control (e.g. [Almutairi et al. 2011]) or sandboxing (e.g. [Lee et al. 2013]) are seen as complementary rather than as alternatives.

8.1 Information Flow Control

It is vital for computer systems to control how information flows through the system. Historically, Access Control (AC) mechanisms were the main means to control the dissemination of information. The purpose of AC is to verify programs' access rights at access points, and to grant or deny access to system objects. Once access to data is granted, no further control is made to ensure the data is handled properly. This all-or-nothing approach is inadequate when it is important to remain in control of data after access; indeed, it requires full trust in applications accessing information.

Information Flow Control (IFC) tracks the flow of information throughout the whole system and ensures that data is handled according to the associated policy. Research on IFC dates back to the 1970's [Cohen 1978, Denning 1976, Fenton 1974] in the context of military systems. IFC relates to two data properties: its *confidentiality* and its *integrity*, that is, where the data is allowed to flow to and where it can flow from. The core principle of IFC is *non-interference* [Goguen and Meseguer 1982], that requires the independence/separation of public data from secret data and/or data that is trusted for integrity from untrusted data.

8.1.1 IFC in programming languages

This section provides a brief overview of IFC in programming languages. An in-depth survey on the issue can be found in [Sabelfeld and Myers 2003].

One of the means to enforce IFC is through security-typed languages where data flow requirements are explicitly declared as part of the type of each variable, an active research area [Myers and Liskov 1997, Orbaek and Palsberg 1997, Volpano et al. 1996, Zdancewic and Myers 2001]. In their seminal work, Volpano and Smith [Volpano and Smith 1997] suggested augmenting a traditional language type system with data flow annotations. This allows developers to express secrecy and integrity data flow policies that are enforceable by the compiler. This work inspired Jif [Chong et al. 2007, Myers 1999, Myers and Liskov 2000] augmenting Java with IFC policies, which uses the JFlow policy language.

Static methods consist of analysing the source code at compilation time to detect any unsafe flow [Denning and Denning 1977, Liu and Milanova 2010, Xiao et al. 2012]. The main limitation of static analysis is the lack of runtime information. This means that static 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.

Static techniques have their place in the approach presented in this disseration, for example, for verifying the correctness of long-lived IFC system components (e.g. bridge-usher implementations discussed in Chapter 3, or the implementation of the reverse proxy discussed in Chapter 6). Indeed, static techniques while generally more taxing in engineering effort, do not create enforcement overhead as dynamic techniques do.

Dynamic tracking is the tracking of information flow at runtime. The earliest known reference to such an approach is [Fenton 1974]. Myers introduced some dynamic elements [Myers 1999] to allow the definition of policy during the execution of the program. Since then, other hybrid implementations [Beres and Dalton 2003, Chandra and Franz 2007, Nair et al. 2008, Vachharajani et al. 2004] – using both runtime and static evaluation – have been proposed. More recent work such as SafeWeb [Hosek et al. 2011] and the work presented in Appendix A, provides strictly dynamic Information Flow Control to instrument web-applications. The accuracy of strictly dynamic IFC is less than static analysis, as implicit flows (e.g. resulting from branching) are extremely hard to track. GIFT [Lam and Chiueh 2006] is a compiler for programs written in C. GIFT takes programmer-specified rules for tag initialization, propagation and combination, and automatically instruments programs so as to execute these rules as part of the program execution. Again, this approach may be of particular relevance when implementing ushers described in Chapter 3 or the proxy described in Chapter 6.

This is of interest since language level enforcement is combined with OS/System level enforcement such as in [Porter et al. 2014, Roy et al. 2009]. In such an approach, IFC is

enforced at the OS level (using techniques similar to those described in Chapter 3) and extended to language level enforcement through a modified JVM (Java Virtual Machine). The JVM enforces IFC within the application memory space and across threads, while the modified OS protects kernel objects. Only the OS and the JVM need to be trusted to guarantee IFC constraints enforcement.

8.1.2 IFC models

As discussed in Chapter 2, a first distinction between models is standard IFC as introduced by Denning [Denning 1976] and Decentralised Information Flow Control (DIFC) as introduced by Myers [Myers and Liskov 1998]. In standard IFC, policy is centrally defined and cannot be managed by individual entities; a widely deployed example is SELinux.[69] DIFC has been examined in several OS research projects, e.g. [Cheng et al. 2012, Krohn et al. 2007, Roy et al. 2009, Zeldovich et al. 2006], and in this work. However, to our knowledge, there is no commercially used implementation.

A second distinction can be made over the label structure. A first approach is a simple label structure where every tag represents a certain security concern. This is the approach introduced by Myers [Myers and Liskov 1997], described in §2.2 or in other OS implementations such as [Krohn et al. 2007]. Another approach relies on a more complex tag structure. Tags are composed of a category c and a sensitivity s such that $t = \langle c, s \rangle$. Categories correspond to the security concern in the first approach described above and the sensitivity to the clearance level required to access the data (e.g. public, internal, confidential, strictly confidential etc.). In order for information to flow between entities, the category must match and the sensitivity level over each category accord. Such a model was introduced by Denning [Denning 1976] and is in use in SELinux and Asbestos [Vandebogart et al. 2007].

8.1.3 IFC in operating systems

We can distinguish between clean slate OS implementations such as Asbestos [Efstathopoulos and Kohler 2008, Krohn et al. 2005, Vandebogart et al. 2007] or HiStar [Zeldovich et al. 2006],[70] from approaches that extend standard OS such as Linux, for example Flume [Krohn et al. 2007] and Laminar[71] [Porter et al. 2014, Roy et al. 2009].[72] While the clean slate approach has the advantage of reducing the TCB and designing the OS from

[69] see `https://wiki.gentoo.org/wiki/SELinux/Information_flow_control` Note that SELinux only implements the Bell-LaPadula model [Bell and LaPadula 1973] and does not provide integrity constraints.

[70] HiStar source code is available at `http://www.scs.stanford.edu/histar/`.

[71] Laminar was designed for the specific task of running a modified JVM enforcing IFC and extending this enforcement to the whole OS.

[72] Laminar source code is available at `https://github.com/ut-osa/laminar`.

scratch with IFC in mind, practical adoption is difficult. Indeed, they require applications and their complex dependencies to be redeveloped or ported to this new environment.

Our approach has been to maintain the Linux system call interface as it is, to minimise the need for application-awareness of IFC. In contrast, in Flume [Porter et al. 2014, Roy et al. 2009], Khron introduced IFC-specific variations of system calls, arguably to prevent *"complicated application development"* [Krohn 2008]. We believe this is unnecessary because, apart from declassifiers and endorsers, applications are likely to run – and probably should run – within a single unmodified security context during their whole life cycle. Indeed, Chapter 6 shows an approach where an application runs several instances in separate contexts rather than constantly switching between security contexts for each request (which prevents privilege escalation and risk of data leaks). Further, Chapter 2 and 7 point out that, from design and policy points of view respectively, it was better to have a distinct entity performing declassification and endorsement. This reduces the privilege needed by standard applications and makes the audit of privileged declassifiers and endorsers easier, due to greater simplicity and a smaller footprint.

The few additional security context manipulation API calls, described in Chapter 3, while slightly complicating the code of applications that manipulate their security contexts, do not justify drastic changes to the system call interface and the related engineering and maintenance cost. The approach preferred in this work is to be able to run everything as it is, constraining IFC enforcement to the LSM module and accepting a slight engineering effort on declassifiers and endorsers.

8.1.4 IFC in distributed systems

As in the approach presented in this dissertation, DStar [Zeldovich et al. 2008], the earliest implementation of IFC in a distributed system, relies on a local IFC enforcement, HiStar [Zeldovich et al. 2006], to ensure that the policies are respected within the local machine. This allows the enforcement of IFC policy at a much finer-grained level than the whole-VM approach. As discussed in Chapter 1, this approach seems to fit better with current cloud computing trends. Beyond the IFC model (as mentioned in §8.1.2), one of the main differences between our approach and that of DStar is the completeness of the messaging system proposed for inter-machine communication (see Chapter 4). Further, HiStar is a clean-slate OS implementation (as discussed in §8.1.3), while we extend the widely deployed Linux OS.

Aeolus [Cheng et al. 2012] implements IFC over a distributed system independently of the OS. Indeed, Aeolus policies are implemented within the Aeolus runtime, which runs on top of a JVM. While this approach is more fine-grained than the one presented in this dissertation, being constrained to Java applications is certainly limiting.

8.2 Taint Tracking

Taint Tracking (TT) systems, sometimes referred to as Data Flow Tracking systems, aim at tracking how information flows from some *source* to some *sink* within an application or a system. Entities in TT systems are generally associated with a single label representing their current taint, which is a collection of tags representing the data's origin (e.g. outside connection). As information is exchanged between entities within the system, tags propagate from entity to entity following information flow. Data flow constraints are only enforced at specified sink points, for example, to prevent certain data leaving a mobile phone [Enck et al. 2010]. There are a number of taint or data flow tracking implementations such as Libdft [Kemerlis et al. 2012], TaintCheck [Newsome and Song 2005], TaintTrace [Cheng et al. 2006], LIFT [Qin et al. 2006], Dytan [Clause et al. 2007] or TaintDroid [Enck et al. 2010].

TT can also be used for integrity purposes. For example, to taint data from untrusted sources, e.g., user input from a TCP stream in a web application environment, and enforce that it is sanitised before being processed [Papagiannis et al. 2011]. This simple mechanism prevents injection attacks that plague badly designed web applications. An example of TT used for confidentiality purposes is to taint sensitive information, e.g. a list of contacts in a mobile phone, and track it through this closed system [Enck et al. 2010]. Sensitive information should only leave the system to go to a number of closely controlled destinations, such as the cloud backup contact list. This approach aids the detection of malicious applications attempting to steal user-sensitive information and send it to third parties. Equally, this type of concern could be captured through the use of IFC policies.

TT techniques have been applied in a cloud computing environment in projects such as [Pappas et al. 2013], FlowWatcher [Muthukumaran et al. 2015] or Silverline [Mundada et al. 2011]. They share similar goals to the work presented in this dissertation, of protecting personal data and offering users a view of how their data is handled within the cloud environment. Further, Cloudopsy [Zavou et al. 2013] explored how those flows of data can be presented to the end user, selecting the radial plot method. Visualisation tools for data flow are further discussed in §9.6.

One concern with TT systems is that there is a gap in time between the occurrence of an issue (e.g. a leak, an attack) and when it is detected [Schwartz et al. 2010], i.e. problems become evident only when the tainted data reaches a sink (enforcement point). Depending on the degree of isolation between the different parts of the system, and the number of system components involved, this tainted data may have 'contaminated' much of the system. While this can be managed in smaller, closed environments, it is less appropriate for cloud services in general. IFC policies present the clear advantage to *prevent* problems as they occur and to stop their effects propagating to a potentially large part of the system.

Some argue that TT is simpler to use than IFC, and incurs lower overhead, but

when the enforcement is systemic and the granularity identical, the overheads are similar (compare with [Enck et al. 2010] for example and the evaluation in Chapter 3). Indeed, the complexity of verifying IFC policy (see Chapter 2) is comparable to the cost of propagating taint.

8.3 Provenance

Provenance typically answers questions such as: how was the data manipulated? who has manipulated it? where does the data come from? Provenance research has historical roots in the database field [Carata et al. 2014]. The nature of databases made it relatively easy to derive provenance information from queries [Buneman et al. 2001] and to provide formalism [Green et al. 2007]. Provenance later generated interest in the tracking of scientific work flow to improve, among other things, reproducibility of experiments [Tylissanakis and Cotronis 2009].

8.3.1 Provenance in operating systems

PASS [Muniswamy-Reddy et al. 2006] collects data within the Linux OS. PASS's solution is centred mostly around the file system, recording relationships between processes and files, and does not capture whole-system data flows. SPADE [Gehani and Tariq 2012] captures OS-level provenance in a distributed environment. SPADE leverages access decisions recorded in the native audit mechanism of the Linux OS,[73] specific hooks in FUSE[74] and process information from /proc, to provide observed provenance. The main shortcoming of these approaches is that they fail to capture the entirety of all interactions occurring within the OS (see §8.3.2) [Pohly et al. 2012]. Macko et al. [Macko et al. 2011], also proposed the capture of provenance data at the hypervisor level, capturing provenance information from the host VM without requiring its modification.

On Android, Quire [Dietz et al. 2011] captures provenance on IPC and transfers the information from process to process, allowing individual processes to take security decisions based on this provenance information. The authors argue that it simplifies management compared to IFC systems (as security context need not be defined). However, they acknowledge that some corner cases are not so readily captured. Further, we argue that this approach suffers from the time-of-detection, time-of-enforcement gap issue. Indeed, access decisions are only made on privileged calls and therefore do not differ much in that regard from Taint Tracking systems (discussed in §8.2).

[73] Spade also supports Windows and Mac OS X.
[74] `http://fuse.sourceforge.net/`

8.3.2 Whole-system provenance

Pohly et al. [Pohly et al. 2012] argued that understanding interaction between processes and files was not enough to capture provenance within an OS. Indeed, in order to understand the complexity of an OS and the complex interactions occurring, it is necessary to capture the data exchanges on all system calls. A number of interactions (e.g. shared memory, message passing etc.) are not observable by earlier approaches. Hi-Fi [Pohly et al. 2012] collects whole-system provenance data at the kernel level, using the LSM framework, and is therefore able to observe all interactions between kernel objects. As discussed in Chapter 5, the amount of data collected by a provenance system is hard to manage and collection should be limited, based on the policy in place. Bates et al. [Bates et al. 2015a] used SELinux policy to reduce the amount of data collected by Hi-Fi.

Expanding on this work, Bates et al. [Bates et al. 2015b], propose a Linux Provenance Module (LPM) providing hooks akin to LSM,[75] but for the specific purpose of provenance data collection and enforcement of Provenance-Based Data Loss Prevention—i.e. preventing sensitive data from leaving a corporate domain—policy that can easily be expressed in IFC. The proposed mechanism duplicates the LSM mechanism. This duplication seems dubious from a software architecture point of view and LSM stacking approaches [Edge 2010; 2011; 2012, Quaritsch and Winkler 2004, Schaufler 2014] may be a more sensible alternative. Recent changes to LSM stacking in the kernel mainline is making such an approach feasible [Edge 2015]. Our most recent implementation relies on these improvements.[76] The approach proposed in Chapter 5 is compatible with the LSM stacking approach and could easily be adapted to LPM if it made its way into the kernel mainline.

8.3.3 Provenance-based access control

Systems have been designed to make access control decisions based on provenance data. This approach is called provenance-based access control (PBAC)[77] [Bates et al. 2013, Nguyen et al. 2012; 2014, Park et al. 2012, Sun et al. 2014]. PBAC uses provenance data to make decisions on user access to data objects. More complex systems can use provenance to determine the policy to apply to a data object by exploring the policy applied to its ancestors [Park et al. 2012]. In [Bates et al. 2015b] early work on such a system for the Linux OS is presented (see §8.3.2).

The PBAC approach is interesting and has its place in complementing IFC enforcement (this aspect is further discussed in §9.6). It cannot be enforced continuously, as querying a

[75] LPM source code is available at `https://bitbucket.org/uf_sensei/redhat-linux-provenance-release`.

[76] see `http://camflow.org/`

[77] PBAC is not to be confused with Provenance Access Control (PAC), which concerns the access to provenance data such as in [Cadenhead et al. 2011].

provenance graph is costly compared to the simple subset-relationship-based DIFC policy. It therefore suffers from the same issue of time-of-event vs time-of-detection as TT systems (see §8.2). However, it has its place in building declassifiers and endorsers as discussed in Chapter 7.

A general observation is that dataflow-based mechanisms can be seen as a spectrum of approaches with a high level of similarity. Their usage and convergence are worth further investigation (see §9.7).

Chapter 9

Conclusion & future work

This dissertation presented the work on CamFlow, a proof-of-concept implementation to assess the potential of Information Flow Control and Audit for PaaS, in order to facilitate compliance with regulation. An IFC model introducing separation of duty, parametrisation and removing the risk introduced by implicit declassification and endorsement (as used in earlier work), was presented in **Chapter 2**. **Chapter 3** presented the Linux implementation for local IFC enforcement. The implementation is self-contained and easily maintainable, providing compatibility with existing legacy applications. Further, API are provided for developers to provide customised management aspects, audit capture and connection to services implementing IFC at a different layer (e.g. database, datastore, cache etc.). **Chapter 4** presented the messaging middleware enforcing IFC – developed within the CloudSafetyNet project – to provide a standardised interface for interactions between applications or with services. **Chapter 5** described in detail the audit mechanism deriving from research on whole-system provenance. It presented implementation details and how the audit graph can be exploited to verify the respect of regulations. **Chapter 6** presented a proof-of-concept, container-based PaaS platform, using a number of open source tools, in order to demonstrate the feasibility of the proposed approach. A streamlined mechanism to provide IFC-constrained web applications is provided. Finally, **Chapter 7** discussed how complex policies can be enforced, and explored use cases based on regulations and recommendations from data privacy agencies, to demonstrate how Information Flow Control and Audit help with complying with regulations. The remainder of this chapter discusses future work required in order to realise the vision of a practical IFC&A system.

9.1 Achievements to date and steps towards a fully-fledged PaaS platform

This dissertation presented the basic building blocks necessary to move towards creating a PaaS platform. The Linux enforcement and audit mechanisms presented in Chapter 3 and Chapter 5 have been implemented and are available for download.[78] The middleware – presented in Chapter 4 – to provide IFC-compliant message exchange between machines has been partly implemented and experimented on by my colleagues. This component should join the available source code by 2017. The audit capture mechanisms focused around the OS were described in Chapter 5. Chapter 6 discussed early investigations around building a fully-fledged PaaS platform.

On the OS enforcement front, we aimed for an implementation which was the least disruptive. In past work we can identify two major approaches:

1. a microkernel that enforces IFC, on top of which a user-space POSIX API is built [Zeldovich et al. 2006];

2. preventing normal usage of the POSIX API in the standard OS and forcing the use of an IFC-aware, use-space-mediated API [Krohn et al. 2007].

The first approach could be argued to provide the best guarantees, as the small size of the kernel has the potential to allow formal verification. However, microkernels have been available for some time, and adoption is proving difficult. The emergence of unikernels may bring a change to the status quo (unikernel solutions are further discussed in §9.4). The second approach is in the author's opinion not viable, since deviating from the standard API increases the engineering effort required. The solution described in this dissertation therefore focused on providing IFC in a standard OS, maintaining the standard POSIX API, with the aim of facilitating adoption by minimising the effort required.

Communication middleware is commonplace in cloud environments; so inter-machine communication by this means in such a context is therefore justifiable. However, in order to guarantee adoption, work remains to be done in order to ensure enforcement over the standard socket API. This is being actively developed[79] and we expect future improvement of LSM stacking [Edge 2015] to help in that regard. Until such progress is made, this aspect represents one of the major adoption hurdles.

Integration of the audit capture in a cloud environment and more generally in a distributed environment is actively being pursued. The author is currently working on integration with a stream processing framework to allow processing of audit log data in real time, subscription to specific aspects and long term storage. The aim is to allow

[78] `http://camflow.org/`

[79] Development can be followed at `https://github.com/CamFlow/camflow-dev`.

customisable and potentially complex processing chains to be built around the generated audit data. This particular aspect may prove an essential selling point as far as adoption is concerned. [80]

The work presented in Chapter 6 is a proof-of-concept rather than a fully-fledged PaaS solution. The research discussed in this thesis mostly focused on core implementation issues at the system level and issues arising when scaling to many nodes were not considered. This is a limitation that future work should address. It remains to be seen how the IFC&A system-level API can be used by platforms such as, for example, OpenStack[81], Cloud Foundry[82] or AppScale [Chohan et al. 2010].

9.2 IFC&A for IaaS offerings

In this dissertation we focused particularly on the steps towards the implementation of IFC&A as a Service for PaaS offerings. However, could IFC&A be applied to IaaS offerings? Here, the granularity at which it should be provided needs to be considered.

- At the granularity of the entire Virtual Machine, constraints enforcement and audit of data flow have been proposed e.g. [Ermolinskiy 2011, Ganjali and Lie 2012, Lee et al. 2015];
- At application granularity within VMs, which would rely on a mechanism similar to those presented in this thesis, but would require the use of remote attestation to verify the presence of the enforcement mechanism (this is further discussed in §9.3).

As further discussed in §9.5, a satisfactory IFC&A mechanism will need to function across multiple layers. The granularity should be able to vary and coexist from entire virtual machine granularity to an individual entry in a database, as most appropriate. Some steps have been taken in this direction [Porter et al. 2014], but much work remains.

9.3 Leveraging hardware roots of trust

IFC protection is only guaranteed above the technical layer in which the IFC mechanism operates. Safe exchange of data in an IFC-context relies on the trust placed in this mechanism. In a cloud context, the enforcement mechanism is provided and guaranteed by the cloud provider. If trust can be established with the cloud provider, no other party needs to be trusted to guarantee secrecy and integrity of data. This trust relationship can be established as the cloud provider is bound by contract, regulation and economic interest.

[80] The code will be made available at `http://camflow.org/` as soon as a working prototype is ready.
[81] `https://www.openstack.org/`
[82] `https://www.cloudfoundry.org/`

This trust is demonstrated every day by companies adopting the cloud. However, when moving towards a potentially self-managed 'things' infrastructure, this trust relationship becomes more complex to establish. There is a need to demonstrate, reliably, that a particular machine has the appropriate untampered-with IFC enforcement mechanisms in place.

One such approach is to leverage Trusted Platform Modules (TPM) [Morris 2011], as used for remote attestation [Santos et al. 2009]. TPM is used to generate an unforgeable hash representing the state of the hardware and software of a given platform, that can be remotely verified. Therefore, a company could audit the implementation of an IFC enforcement mechanism and ensure that the kernel security module, messaging middleware and the configuration they provide are indeed running on the platform. Any difference between the expected state of the software stack and the platform would be detected and might represent a breach of trust.

TPM, with remote attestation, is reaching maturity for cloud computing [Berger et al. 2006], with IBM proposing[83] a scalable trusted platform based on virtual TPM [Berger et al. 2015]. Their work describes a mechanism allowing TPM and remote attestation to be provided for VM- and container-based solutions, covering the whole range of contemporary cloud offerings. Furthermore, the approach not only allows the state of the software stack to be verified at boot time, but also during execution, and can thus prevent run-time modification of the system configuration. Similar mechanisms exist for mobile phones [Nauman et al. 2010], embedded systems [Aaraj et al. 2008], or for "swarms" of IoT devices, as in SEDA [Asokan et al. 2015].

In addition to verifying a platform, the same techniques can also be used to verify the integrity of a remote system at run-time, to ensure that an entity has in place the appropriate IFC enforcement mechanisms before data is exchanged. This can be achieved through standard remote attestation techniques. Furthermore, integrity tags can be derived from a hardware-backed source, such as GPS-based location, using the mechanism described in [Jayaram et al. 2014].

9.4 IFC&A for Unikernels

As discussed in Chapter 1, containers are a current trend in cloud computing, giving a powerful, easy way to package applications in a cloud computing environment. However, "library operating systems" are also gaining traction as **Unikernels** for cloud computing environments [Kurth 2015].

As shown in Fig. 9.1, a) the hypervisor may support separated VMs, each with its own OS; b) a single shared OS may support containers that isolate applications; or c) the

[83] see `http://www.ibm.com/support/knowledgecenter/P8ESS/p8hat/p8hat_enablevtpm.htm`

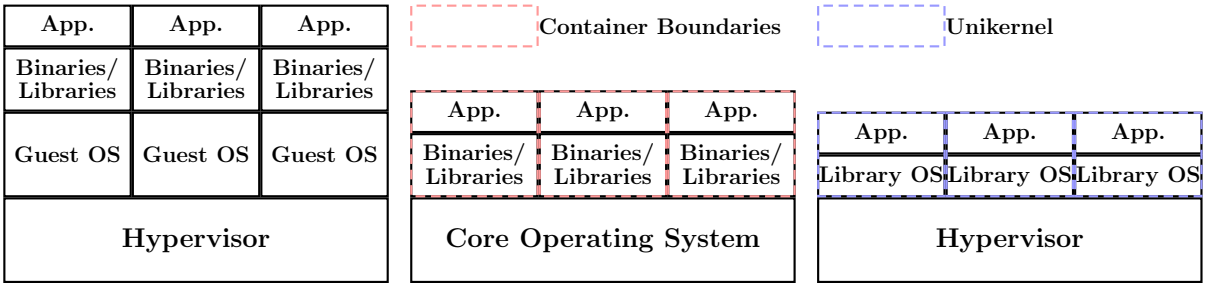

Figure 9.1: Comparison of Virtual Machine, Container and Unikernel approach.

hypervisor may directly support applications, each with its own library OS (Unikernel). That is, each Unikernel runs a single application directly over the hypervisor without the need for an OS. Unikernel applications are built using library OS [Madhavapeddy and Scott 2014] and only contain the minimum functionality for the application to work. Unikernels, as containers, are easily deployed, but come with a number of advantages:

1. extremely small footprint;

2. reduced attack surface by deploying less code and therefore improving security;

3. extremely fast deployment and migration to support edge-computing [Madhavapeddy et al. 2015];

4. whole-system optimisation targeted to the specific application [Madhavapeddy et al. 2010].

There exist a number of implementations focusing on different aspects: HaLVM[84] in Haskell, Mirage OS [Madhavapeddy et al. 2010] in OCaml and IncludeOS [Bratterud et al. 2015] taking a clean state approach; ClickOS [Martins et al. 2014] emphasing speed; and Rumprun built upon rump kernel drivers,[85] providing compatibility with legacy Linux software.

Questions that need to be addressed in future work on integrating Unikernels and IFC are:

1. How best to implement IFC in such an environment?
   IFC might be provided within the application, e.g. using FlowCaml [Simonet and Rocquencourt 2003] for Mirage OS, or a similar mechanism in Haskell [Li and Zdancewic 2006] for HaLVM.

2. How best to handle the audit mechanism in such a case?

[84] `https://galois.com/project/halvm/`
[85] `https://github.com/rumpkernel/rumprun`

3. How can trust be achieved if the application is responsible for the correct implementation of IFC? Alternatively, IFC and Audit could be implemented in the hypervisor; indeed, solutions exist for IFC [McCune et al. 2006] and provenance for the Xen Hypervisor [Barham et al. 2003].

9.5 Multi-layer IFC&A and legacy applications

As discussed in Chapters 3, 4 and 5 there is often a need to link between several layers of enforcement or audit. This is due to the fact that the enforcement and audit granularity achievable in the OS (i.e. process) is not adapted to some applications (e.g. database, key-value store, cache etc.). While we proposed mechanisms to facilitate this cross-layer enforcement and audit (see §3.5.1 and §5.2.2), work remains on how such enforcement should be achieved within applications. In Laminar [Porter et al. 2014, Roy et al. 2009] a customised JVM ensures application IFC enforcement at the thread granularity. However, a high level of work is required to annotate legacy applications. In the domain of causal monitoring, the Pivot project [Mace et al. 2015] instruments legacy applications, through aspects independent from the application code. This approach is similar to the work described in Appendix A. Such an approach seems more appropriate for dealing with legacy software and could extend to compiled languages through a project such as AspectC [Coady et al. 2001] or AspectC++ [Spinczyk and Lohmann 2007, Spinczyk et al. 2002]. Alternatively, static analysis techniques could be employed (see §8.1.1). However, a mechanism to build trust from such analyses in an *ad hoc* interaction must be provided.

9.6 Future work on Information Flow Audit

A number of issues are left for investigation where Information Flow Audit is concerned:
Controlling access to audit data: The audit data collected are in their own right sensitive and may potentially constitute personal data, creating the need for specific access control (AC) and privacy preserving mechanisms [Xu et al. 2009]. Mechanisms must be developed to ensure that users (such as customers, regulatory agencies, and certification bodies) are able to perform queries and obtain the required insight, while guaranteeing that the audit does not introduce further privacy issues. Braun et al. [Braun et al. 2008], point out the difficulty inherent in the protection of provenance data. A number of different languages have been proposed to model provenance AC [Cadenhead et al. 2011, Ni et al. 2009], however there is no well-accepted standard nor model.
Solving the storage problem: A central challenge for the management of provenance-like audit data is their size. As every flow in the system may potentially be recorded, the size of the provenance data tends to grow very quickly. Pruning techniques [Muniswamy-

Reddy et al. 2006] may be used, such as deleting the audit data of an entity with no descendant when this entity is deleted, or compressing a long chain into a single node (the *super node* as described in [Braun et al. 2006]). Effective pruning would require application of research from information flow analyses and programming language 'garbage collection' techniques within this particular area. Further, Braun et al. [Braun et al. 2006] propose to reference certain attributes through items in a separate database or as virtual nodes (as discussed under AC). They also suggest the deletion of irrelevant attributes, which would need to be identified, based on the specific context of the application. Approaches for pruning the data according to security policy have been proposed, based on SELinux policies [Bates et al. 2015a], or our current method based on IFC policy. Here, only audit data of *sensitive* entities are collected, removing the "noise" generated by irrelevant, routine system operations. An approach for trying to filter and manage very large collections of data is to apply machine learning algorithms. In our context, the machine learning approach could help filter the provenance data to record only those flows that are at odds with the standard operating pattern of the system.

However, pruning techniques might delete information that would have proved useful for certain investigations. Further research on pruning mechanisms that are aware of data utility constraints is required.

Data visualisation and abstraction: Fig. 5.4 shows how information flow audit data can be visualised in our current prototype. Further information is displayed when hovering over the objects. While this may be understandable by a system engineer, it may not convey any directly useful information to an end user or an auditor wishing to examine in which context a certain item of data is being used. Mechanisms need to be developed to abstract the audit graph in a manner relevant to a particular user. Approaches similar to those discussed in pruning, such as the generation of *super nodes* [Braun et al. 2006] may be worth investigating. Borkin et al. [Borkin et al. 2013] explored the representation of provenance as a graph or a radial plot (across multiple criteria). It would be interesting to explore which approaches are most effective for visualising key nodes in the context of auditing compliance with regulations.

Applying 'big data' analyses: The data collected by our audit mechanism by its sheer size can be considered big data. They are naturally represented as graphs, which are mathematical constructs that can be reasoned about and easily analysed. Further, it is likely that data flow will generally follow certain patterns. Machine learning approaches can also be used to analyse the audit data. This type of approach has been used within intrusion detection systems [Tsai et al. 2009], and could be applied to reason automatically about conformance with regulatory or contractual requirements.

Widely distributed IFA: Federation of cloud providers can reasonably be envisaged (with the caveat that the interoperability of solutions [Angelino et al. 2011] will be a

challenge when the components are not uniform). However, extension beyond cloud boundaries (e.g. to cloudlets [Crowcroft et al. 2011], IoT devices, etc.) requires work on, among other things, the root of trust (we discuss a potential hardware solution in §9.3) or widely distributed provenance analyses [Gehani and Tariq 2012], with the specific integrity needs of regulation verification.

Augmenting an IFA data model with legal relationships: The IFA data model in its current form allows relationships between virtual data and software elements, hardware and persons to be represented. However, in a legal context it may also be useful to be able to represent contractual relationships as edges that link legal or natural persons to each other, and with software or physical artefacts. Indeed, this may prove useful in improving transparency and helping customers navigate the complex service provision model introduced by the cloud and the Internet of Things (IoT).

Provenance-based policy: As seen in the use case (§7.4), some compliance requirements cannot be fully captured from simple IFC primitives (e.g. constraints applying after certain periods of time). One solution is to allow the application to specify Provenance-based access control policy [Bates et al. 2013]. However, as with access control, the policy will only be enforced at a particular point in the system and it is hard to guarantee that no other path exists. A possible solution is to force data that is destined for a third party to first flow through an element that enforces AC, using an appropriate composition of IFC enforcers/declassifiers. AC decisions could be made based on queries over the provenance engine, or by applying reasoning techniques to metadata, such as the 'baggage' used in the Pivot system [Mace et al. 2015]. The notion of 'baggage' can be seen as an extended version of taint tracking [Enck et al. 2010], where instead of simple taint, more complex metadata are attached to data items and flow with them through the system. This may improve performance, as no query over the provenance graph is required, but would necessitate prior knowledge of the metadata required to make policy decisions.

Using Information Flow Audit for digital forensics: As mentioned in Chapter 1 there is potential for Information Flow Audit data to be used for forensic purposes. There are a few publications – e.g. [Abbadi and Lyle 2011, Turner 2005, Zhou et al. 2008] – suggesting the exploitation of provenance for such a purpose. However, while some research in this direction has been carried out, much work remains to ensure the trustworthiness and integrity of provenance data [Hasan et al. 2009, Zhang et al. 2009], sufficient to be admissable in a court of law. We suggest using hardware roots of trust for IFC (see §9.3). The use of TPM has also been proposed to guarantee provenance integrity [Xu et al. 2012].

Sakka et al. [Sakka et al. 2010] discuss provenance in a cloud, relating to document life cycles. The context is banking under French regulation,[86] to ensure the probative value of electronic documents. This requires the emitter of any document to be identified and

[86]Code Civil Article 1316-1.

guarantees its integrity, which is achieved through provenance in a particular closed system. Curbera et al. [Curbera et al. 2008] proposed to use provenance to demonstrate compliance of businesses with regulation such as the *Sarbanes-Oxley Act* or *HIPAA*. Contrary to our approach, that captures whole-system provenance and does not require trust in involved applications, the above approaches require existing applications to be instrumented.

In Section 1.4.2 we defined the scope of this work to exclude covert channels and malicious attacks. Instead, the focus is enforcement of and compliance with policy. However, IFC potentially assists in containing the effects of attacks and IFA, through logging both allowed and rejected flows, potentially helps in post-attack forensics. It is left for future work to investigate to what extent IFC and IFA can contribute to cybersecurity.

9.7 Towards a unified data flow mechanism

As we discussed earlier, there is a clear similarity between the mechanisms employed to enforce Information Flow Control and those to collect provenance data [Pohly et al. 2012], taint tracking [Enck et al. 2010], causal monitoring [Mace et al. 2015] etc. Such mechanisms, while having clearly different purposes, belong to a spectrum of mechanisms monitoring the flow of information through a system. The Linux Provenance Module [Bates et al. 2015b] may be a step towards providing a standard framework for data flow mechanisms, although duplicating LSM may not be a long-term viable approach (as discussed in §8.3). Further work on data flow based policy and/or monitoring for the Linux kernel may lead to the development of a general purpose framework.

9.8 Concluding remarks

Laws, policies and regulations are increasingly being defined by national governments and bodies such as the EU, concerning the proper usage of data. At present it is unclear whether and how such policies can be enforced, and whether and how a responsible entity can routinely demonstrate compliance with them. Yet heavy fines can be levied on companies and institutions when leaks occur.

We have shown that Information Flow Control and Audit is a practical solution to at least some of these problems. We have investigated in detail, via a proof-of-concept implementation, how IFC&A can be designed and deployed in PaaS or SaaS cloud services. We have shown the trust assumptions that are necessary for such solutions to be relied on. This work paves the way to a fully-fledged platform that can and should be adopted by such cloud service providers. We have also speculated on the feasibility of extending IFC&A to wider distributed deployments.

Bibliography

Najwa Aaraj, Anand Raghunathan, and Niraj K Jha. Analysis and Design of a Hardware/-Software Trusted Platform Module for Embedded Systems. *Transactions on Embedded Computing Systems (TECS)*, 8(1):8, 2008.

Imad Abbadi and John Lyle. Challenges for provenance in cloud computing. In *Workshop on Theory and Practice of Provenance*. USENIX, 2011.

Sherif Akoush, Lucian Carata, Ripduman Sohan, and Andy Hopper. MrLazy: Lazy Runtime Label Propagation for MapReduce. In *6th Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2014.

Abdulrahman Almutairi, Muhammad I Sarfraz, Saleh Basalamah, Walid G Aref, and Arif Ghafoor. A distributed access control architecture for cloud computing. *IEEE Software*, (2):36–44, 2011.

Elaine Angelino, Uri Braun, David A Holland, and Daniel W Margo. Provenance integration requires reconciliation. In *Workshop on Theory and Practice of Provenance*. USENIX, 2011.

N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. Seda: Scalable embedded device attestation. In *SIGSAC Conference on Computer and Communications Security*, pages 964–975. ACM, 2015.

Jean Bacon, Ken Moody, and Walt Yao. A Model of OASIS Role-based Access Control and its Support for Active Security. *ACM Transactions on Information and System Security (TISSEC)*, 5(4):492–540, 2002.

Jean Bacon, David Eyers, Thomas Pasquier, Jatinder Singh, Ioannis Papagiannis, and Peter Pietzuch. Information Flow Control for Secure Cloud Computing. *Transactions on Network and System Management SI Cloud Service Management*, 11(1):76–89, 2014.

Sruthi Bandhakavi, Charles Zhang, and Marianne Winslett. Super-sticky and declassifiable release policies for flexible information dissemination control. In *Workshop on Privacy in Electronic Society*, pages 51–58. ACM, 2006.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

Adam Bates, Ben Mood, Masoud Valafar, and Kevin Butler. Towards secure provenance-based access control in cloud environments. In *Conference on Data and Application Security and Privacy*, pages 277–284. ACM, 2013.

Adam Bates, Kevin Butler, and Thomas Moyer. Take only what you need: leveraging mandatory access control policy to reduce provenance storage costs. In *Workshop on Theory and Practice of Provenance*, pages 7–7. USENIX, 2015a.

Adam Bates, Dave Tian, Kevin Butler, and Thomas Moyer. Trustworthy Whole-System Provenance for the Linux Kernel. In *Security Symposium*. USENIX, 2015b.

Mick Bauer. Paranoid Penguin: an Introduction to Novell AppArmor. *The Linux Journal*, 2006(148):13, 2006.

David E. Bell and Leonard J. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, The MITRE Corp., Bedford MA, 1973.

Martin Bellamy. Adoption of Cloud Computing services by public sector organisations. In *World Congress on Services*, pages 201–208. IEEE, 2013.

Kathleen Benitez and Bradley Malin. Evaluating re-identification risks with respect to the HIPAA privacy rule. *Journal of the American Medical Informatics Association*, 17(2): 169–177, 2010.

Yolanta Beres and Chris I Dalton. Dynamic label binding at run-time. In *Workshop on New Security Paradigms*, pages 39–46. ACM, 2003.

Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Security Symposium*, pages 305–320. USENIX, 2006.

Stefan Berger, Kenneth Goldman, Dimitrios Pendarakis, David Safford, Enriquillo Valdez, and Mimi Zohar. Scalable Attestation: A Step Toward Secure and Trusted Clouds. In *International Conference on Cloud Engineering (IC2E)*. IEEE, 2015.

David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing Magazine*, (3):81–84, 2014.

K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR 76-372, MITRE Corp., 1977.

Tim Bird. Measuring Function Duration with ftrace. In *Japan Linux Symposium*, pages 47–54, 2009.

Michelle Borkin, Chelsea S Yeh, Madelaine Boyd, Peter Macko, Krzysztof Z Gajos, Mike Seltzer, Hanspeter Pfister, et al. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2476–2485, 2013.

Alexandre Bouard, Benjamin Weyl, and Claudia Eckert. Practical information-flow aware middleware for in-car communication. In *Workshop on Security, privacy & dependability for cyber vehicles*, pages 3–8. ACM, 2013.

Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *International Conference on Cloud Computing Technology and Science (CloudCom'15)*. IEEE, 2015.

Uri Braun, Simson Garfinkel, David A Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I Seltzer. Issues in automatic provenance collection. In *Provenance and annotation of data*, pages 171–183. Springer, 2006.

Uri Braun, Avraham Shinnar, and Margo I Seltzer. Securing provenance. In *Summit on Hot Topics in Security (HotSec'08)*. USENIX, 2008.

David Brewer and Michael Nash. The Chinese Wall security policy. In *Symposium on Security and Privacy*, pages 206–214. IEEE, 1989.

Justin Brickell and Vitaly Shmatikov. The cost of privacy: destruction of data-mining utility in anonymized data publishing. In *International Conference on Knowledge Discovery and Data Mining (SIGKDD'08)*, pages 70–78. ACM, 2008.

Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *International Conference on Database Theory*, pages 316–330. Springer, 2001.

Serdar Cabuk, Carla E Brodley, and Clay Shields. IP covert timing channels: design and detection. In *Conference on Computer and Communications Security*, pages 178–187. ACM, 2004.

Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham. A language for provenance access control. In *Conference on Data and Application Security and Privacy*, pages 133–144. ACM, 2011.

Lucian Carata, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Selter, and Andy Hopper. A primer on provenance. *Communications of the ACM*, 57(5):52–60, 2014.

Marco Casassa-Mont, Siani Pearson, and Pete Bramhall. Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In *International Workshop on Database and Expert Systems Applications*, pages 377–382. IEEE, 2003.

Marco Casassa-Mont, Vaibhav Sharma, and Siani Pearson. EnCoRe: dynamic consent, policy enforcement and accountable information sharing within and across organisations. Technical Report HPL-2012-36, HP Laboratories, 2012. URL `http://www.hpl.hp.com/techreports/2012/HPL-2012-36.pdf`.

Christopher Cassa, Shannon Wieland, and Kenneth Mandl. Re-identification of home addresses from spatial locations anonymized by Gaussian skew. *International Journal of Health Geographics*, 7(1):45, 2008.

Daniel Catrein, Martin Henze, Klaus Wehrle, and René Hummen. A Cloud Design for User-controlled Storage and Processing of Sensor Data. In *International Conference on Cloud Computing Technology and Science (CloudCom'12)*, pages 232–240. IEEE, 2012.

David Chadwick and Stijn Lievens. Enforcing sticky security policies throughout a distributed application. In *Workshop on Middleware Security*, pages 1–6. ACM, 2008.

David Chadwick and Alexander Otenko. The PERMIS X. 509 role based privilege management infrastructure. *Future Generation Computer Systems*, 19(2):277–289, 2003.

Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Computer Security Applications Conference (ACSAC'07)*, pages 463–475. IEEE, 2007.

Adriane Chapman, M David Allen, and Barbara T Blaustein. It's About the Data: Provenance as a Tool for Assessing Data Fitness. In *Workshop on the Theory and Practice of Provenance*. USENIX, 2012.

Andrew Charlesworth. Clash of the data titans? US and EU data privacy regulation. *European Public Law*, 6(2):253–274, 2000.

Peng Chen and Beth A Plale. Big data provenance analysis and visualization. In *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 797–800. IEEE/ACM, 2015.

Peng Chen, Beth Plale, Y Cheah, Devarshi Ghoshal, Soren Jensen, and Yuan Luo. Visualization of network data provenance. In *International Conference on High Performance Computing (HiPC)*, pages 1–9. IEEE, 2012.

Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Symposium on Computers and Communications (ISCC'06)*, pages 749–754. IEEE, 2006.

Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for Usable Information Flow Control in Aeolus. In *USENIX Annual Technical Conference*, Boston, 2012.

Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. Appscale: Scalable and open AppEngine application development and deployment. In *Cloud Computing*, pages 57–70. Springer, 2010.

Stephen Chong, K Vikram, and Andrew Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Security Symposium*, Boston, MA, 2007. USENIX.

James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206. ACM, 2007.

Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing v3.0, 2011. URL `https://cloudsecurityalliance.org/research/security-guidance/`.

Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *SIGSOFT Software Engineering Notes*, volume 26, pages 88–98. ACM, 2001.

Ellis S. Cohen. Information transmission in sequential programs. *Foundations of Secure Computation*, pages 297–335, 1978.

Jon Crowcroft, Anil Madhavapeddy, Malte Schwarzkopf, Theodore Hong, and Richard Mortier. Unclouded Vision. In *Distributed Computing and Networking*, pages 29–40. Springer, 2011.

Francisco Curbera, Yurdaer Doganata, Axel Martens, Nirmal K Mukhi, and Aleksander Slominski. Business provenance–a technology to increase traceability of end-to-end operations. In *On the Move to Meaningful Internet Systems: OTM 2008*, pages 100–119. Springer, 2008.

Dorothy Denning. A lattice model of secure information flow. *Communication of the ACM*, 19(5):236–243, 1976.

Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communincation of the ACM*, 20(7):504–513, 1977.

Gowri Dhandapani and Anupama Sundaresan. Netlink sockets, overview. Technical Report KS 66045-2228, University of Kansas, 1999. URL `http://qos.ittc.ku.edu/netlink/netlink.pdf`.

T. Dierks and C. Allen. The TLS Protocol Version 1.0. Technical report, IETF, 1999.

Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Security Symposium*, page 24. USENIX, 2011.

Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs Containerization to Support PaaS. In *International Conference on Cloud Engineering (IC2E)*, pages 610–614. IEEE, 2014.

Bob Duncan and Mark Whittington. Enhancing Cloud Security and Privacy: Broadening the Service Level Agreement. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 1088–1093. IEEE, 2015.

Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of Heartbleed. In *Internet Measurement Conference*, pages 475–488. ACM, 2014.

Jake Edge. Kernel key management. *Linux Weekly News*, 2006.

Jake Edge. LSM stacking (again). *Linux Weekly News*, 2010.

Jake Edge. Supporting multiple LSMs. *Linux Weekly News*, 2011.

Jake Edge. Another LSM stacking approach. *Linux Weekly News*, 2012.

Jake Edge. Progress in security module stacking. *Linux Weekly News*, 2015.

Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime verification of authorization hook placement for the Linux Security Modules framework. In *Conference on Computer and Communications Security*, pages 225–234. ACM, 2002.

Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *European Conference on Computer Systems (EuroSys'08)*, pages 301–313. ACM, 2008.

Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

Rania Fahim El-Gazzar. A Literature Review on Cloud Computing Adoption Issues in Enterprises. In *Creating Value for All Through IT*, pages 214–242. Springer, 2014.

Tzilla Elrad, Robert E Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Conference on Operating systems design and implementation (OSDI'10)*, pages 1–6. USENIX, 2010.

Andrey Ermolinskiy. *Design and Implementation of a Hypervisor-Based Platform for Dynamic Information Flow Tracking in a Distributed Environment*. PhD thesis, University of California at Berkeley, 2011.

Stephen Farrell and Russell Housley. An Internet Attribute Certificate Profile for Authorization. Technical report, IETF, 2002.

JS Fenton. An abstract computer model demonstrating directional information flow. Technical report, University of Cambridge, 1974.

David Ferraiolo, Ravi Sandhu, Serban Gavrila, Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-based Access Control. *ACM Transaction on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

Matthias Flittner, Silvia Balaban, and Roland Bless. CloudInspector: A Transparency-as-a-Service Solution for Legal Issues in Cloud Computing. In *IC2E International Workshop on Legal and Technical Issues in Cloud Computing (CLaw'16)*. IEEE, 2016.

Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *Conference on Computer and Communications Security*, pages 330–339. ACM, 2005.

Afshar Ganjali and David Lie. Auditing cloud management using information flow tracking. In *Workshop on Scalable Trusted Computing*, pages 79–84. ACM, 2012.

A Manas Garcia, R Sanz Requena, A Alberich-Bayarri, G Garcia-Marti, Marina Egea, and C Mediavilla Martinez. Coco-Cloud project: Confidential and compliant clouds. In *International Conference on Biomedical and Health Informatics (BHI)*, pages 227–230. IEEE-EMBS, 2014.

Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Network and Distributed System Security Symposium*, volume 3, pages 163–176. Internet Society, 2003.

Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. In *Middleware Conference*, pages 101–120. IEEE/ACM, 2012.

Luigi Giuri and Pietro Iglio. Role Templates for Content-Based Access Control. In *Workshop on Role-based Access Control*, pages 153–159. ACM, 1997.

Joseph Goguen and José Meseguer. Security policies and security models. In *Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 31–40. ACM, 2007.

Dick Hardt. The OAuth 2.0 Authorization Framework. Technical Report RFC 6749, IETF, 2012.

Ragib Hasan, Radu Sion, and Marianne Winslett. Preventing history forgery with secure provenance. *ACM Transactions on Storage (TOS)*, 5(4):12, 2009.

Kevin Kaichuan He. Kernel Korner: why and how to use Netlink socket. *The Linux Journal*, 2005(130):11, 2005.

Richard Hillestad, James Bigelow, Anthony Bower, Federico Girosi, Robin Meili, Richard Scoville, and Roger Taylor. Can electronic medical record systems transform health care? Potential health benefits, savings, and costs. *Health Affairs*, 24(5):1103–1117, 2005.

Kuan Hon, Christopher Millard, Chris Reed, Jatinder Singh, Ian Walden, and Jon Crowcroft. Policy, Legal and Regulatory Implications of a Europe-Only Cloud. Technical report, Queen Mary University of London, School of Law, 2014. URL `http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2527951`.

Petr Hosek, Matteo Migliavacca, Ioannis Papagiannis, David Eyers, David Evans, Brian Shand, Jean Bacon, and Peter Pietzuch. SafeWeb: A Middleware for Securing Ruby-based Web Applications. In *International Middleware Conference*, pages 491–512. ACM, 2011.

Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *IEEE Computer*, (2):85–88, 2015.

David Ingram. Reconfigurable middleware for high availability sensor systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 20. ACM, 2009.

Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):175–205, 2004.

K. R. Jayaram, David Safford, Upendra Sharma, Vijay Naik, Dimitrios Pendarakis, and Shu Tao. Trustworthy Geographically Fenced Hybrid Clouds. In *International Conference on Middleware*. ACM, 2014.

Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 121–132. ACM, 2012.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*. Springer, 1997.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

Daniel Kifer. Attacks on privacy and deFinetti's theorem. In *International Conference on Management of Data (SIGMOD'09)*, pages 127–138. ACM, 2009.

Chongkyung Kil, Emre Can Sezer, Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *Dependable Systems & Networks (DSN'09)*, pages 115–124. IEEE, 2009.

Ryan Ko, Markus Kirchberg, and Bu Sung Lee. From System-centric to Data-centric Logging-accountability, Trust & Security in Cloud Computing. In *Defense Science Research Conference and Expo*, pages 1–4. IEEE, 2011.

Maxwell Krohn. *Information Flow Control for Secure Web Sites*. PhD thesis, Massachusetts Institute of Technology, September 2008.

Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazieres, Robert Morris, Michelle Osborne, Steven VanDeBogart, and David Ziegler. Make Least Privilege a Right (Not a Privilege). In *Hot Topics in Operating Systems (HotOS'05)*. USENIX, 2005.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *Symposium on Operating Systems Principles*, pages 321–334. ACM, 2007.

NV Kumar and RK Shyamasundar. Realizing Purpose-Based Privacy Policies Succinctly via Information-Flow Labels. In *Big Data and Cloud Computing (BDCloud'14)*, pages 753–760. IEEE, 2014.

Lars Kurth. 7 Unikernel Projects to Take On Docker in 2015, 2015. `https://www.linux.com/news/enterprise/cloud-computing/819993-7-unikernel-projects-to-take-on-docker-in-2015`.

Lap Chung Lam and Tzicker Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Annual Computer Security Applications Conference*, pages 463–472. IEEE, 2006.

Brian Lee, Abir Awad, and Mirna Awad. Towards secure provenance in the cloud: A survey. In *International Conference on Utility and Cloud Computing (UCC)*, pages 577–582. IEEE/ACM, 2015.

Sangmin Lee, Edmund L Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. πBox: A Platform for Privacy-Preserving Apps. In *Symposium on Networked System Design and Implementation (NSDI'13)*, pages 501–514. USENIX, 2013.

Michael LeMay, Carl Gunter, et al. Cumulative attestation kernels for embedded systems. *IEEE Transactions on Smart Grid*, 3(2):744–760, 2012.

Peng Li and Steve Zdancewic. Encoding information flow in Haskell. In *Workshop on Computer Security Foundations*, pages 12–16. IEEE, 2006.

Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using Aspect-Oriented Programming. In *International Conference on Software Engineering*, pages 418–427. ACM, 2000.

Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 146–155. IEEE, 2010.

Emil Lupu and Morris Sloman. Reconciling Role Based Management and Role Based Access Control. In *Workshop on Role-based Access Control*, pages 135–141. ACM, 1997.

Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Symposium on Operating Systems Principles (SOSP '15)*, pages 378–393. ACM, 2015.

Peter Macko, Marc Chiarini, and Margo Seltzer. Collecting Provenance via the Xen Hypervisor. In *Workshop on Theory and Practice of Provenance*. USENIX, 2011.

Anil Madhavapeddy and David J Scott. Unikernels: the rise of the virtual library operating system. *Communication of the ACM*, 57(1):61–69, 2014.

Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. Turning down the LAMP: software specialisation for the cloud. In *Conference on Hot topics in Cloud Computing (HotCloud'10)*, volume 10, pages 11–11. USENIX, 2010.

Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-in-time summoning of Unikernels. In *Symposium on Networked System Design and Implementation (NSDI'15)*. USENIX, 2015.

Bradley Malin and Latanya Sweeney. How (not) to protect genomic data privacy in a distributed network: using trail re-identification to evaluate and design anonymity protection systems. *Journal of biomedical informatics*, 37(3):179–192, 2004.

Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Programming Languages and Systems*, pages 129–145. Springer, 2004.

Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 459–473. USENIX, 2014.

Jonathan M McCune, Trent Jaeger, Stefan Berger, Ramon Caceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *Computer Security Applications Conference (ACSAC'06)*, pages 23–32. IEEE, 2006.

M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software: Practice and Experience*, 22(8):673–694, 1992.

Matteo Migliavacca, Ioannis Papagiannis, David M Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. DEFCon: High-performance event processing with information security. In *USENIX Annual Technical Conference*, Boston, MA, USA, 2010.

Christopher J Millard, editor. *Cloud Computing Law*. Oxford University Press, 2013.

Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, et al. The Open Provenance Model Core Specification (v1.1). volume 27, pages 743–756. Elsevier, 2011.

Thomas Morris. Trusted Platform Module. In *Encyclopedia of Cryptography and Security*, pages 1332–1335. Springer, 2011.

Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi. An Aspect-Oriented approach for the systematic security hardening of code. volume 27, pages 101–114. Elsevier, 2008.

Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. Silverline: Data and network isolation for cloud services. In *Workshop on Hot Topics in Cloud Computing (HotCloud'11)*. USENIX, 2011.

Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference*, pages 43–56, 2006.

Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A Holland, Peter Macko, Diana Maclean, Daniel W Margo, Margo I Seltzer, and Robin Smogor. Layering in provenance systems. In *USENIX Annual technical conference*. USENIX, 2009.

Divya Muthukumaran, Dan O'Keeffe, Christian Priebe, David Eyers, Brian Shand, and Peter Pietzuch. FlowWatcher: Defending against Data Disclosure Vulnerabilities in Web Applications. In *Conference on Computer and Communications Security (SIGSAC)*, pages 603–615. ACM, 2015.

Andrew Myers. JFlow: Practical Mostly-static Information Flow Control. In *Symposium on Principles of Programming Languages*, pages 228–241. ACM, 1999.

Andrew Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Symposium on Security and Privacy*, pages 186 –197. IEEE, 1998.

Andrew Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *Symposium on Operating Systems Principles*, pages 129–142. ACM, 1997.

Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Workshop on Cloud Computing Security*, pages 113–124. ACM, 2011.

Srijith K Nair, Patrick ND Simpson, Bruno Crispo, and Andrew S Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.

Arvind Narayanan and Vitaly Shmatikov. Myths and fallacies of personally identifiable information. *Communication of the ACM*, 53:24–26, 2010.

Mohammad Nauman, Sohail Khan, Xinwen Zhang, and Jean-Pierre Seifert. Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform. In *Trust and Trustworthy Computing*, pages 1–15. Springer, 2010.

James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed Systems Security Symposium*. Internet Society, 2005.

Dang Nguyen, Jaehong Park, and Ravi Sandhu. Dependency path patterns as the foundation of access control in provenance-aware systems. In *Workshop on Theory and Practice of Provenance*. USENIX, 2012.

Dang Nguyen, Jaehong Park, and Ravi Sandhu. Adopting provenance-based access control in OpenStack cloud IaaS. In *Network and System Security*, pages 15–27. Springer, 2014.

Qun Ni, Shouhuai Xu, Elisa Bertino, Ravi Sandhu, and Weili Han. An access control language for a general provenance model. In *Secure Data Management*, pages 68–88. Springer, 2009.

Ben Niu and Gang Tan. Efficient User-space Information Flow Control. In *Symposium on Information, Computer and Communications Security (SIGSAC'13)*, pages 131–142. ACM, 2013.

Keisuke Okamura and Yoshihiro Oyama. Load-based covert channels between Xen virtual machines. In *Symposium on Applied Computing*, pages 173–180. ACM, 2010.

P. Orbaek and J. Palsberg. Trust in the λ-calculus. volume 7, pages 557–591, New York, 1997. Cambridge University Press.

Zizi Papacharissi and PaigeL. Gibson. Fifteen minutes of privacy: Privacy, sociality, and publicity on social network sites. In *Privacy Online*, pages 75–89. Springer, 2011.

Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. PHP Aspis: Using partial taint tracking to protect against injection attacks. In *Conference on Web Application Development*, page 13. USENIX, 2011.

Vasilis Pappas, Vasileios P Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D Keromytis. Cloudfence: Data flow tracking as a cloud service. In *Research in Attacks, Intrusions, and Defenses*, pages 411–431. Springer, 2013.

Jaehong Park, Dang Nguyen, and Ravi Sandhu. A provenance-based access control model. In *Annual International Conference on Privacy, Security and Trust*, pages 137–144. IEEE, 2012.

Joon S Park and Ravi Sandhu. Binding Identities and Attributes Using Digitally Signed Certificates. In *Annual Conference on Computer Security Applications*, pages 120–127. IEEE, 2000.

Thomas Pasquier and David Eyers. Information Flow Audit for Transparency and Compliance in the Handling of Personal Data. In *IC2E International Workshop on Legal and Technical Issues in Cloud Computing (CLaw'16)*. IEEE, 2016.

Thomas Pasquier and Julia Powles. Expressing and Enforcing Location Requirements in the Cloud using Information Flow Control. In *IC2E International Workshop on Legal and Technical Issues in Cloud Computing (CLaw'15)*. IEEE, 2015.

Thomas Pasquier, Brian Shand, and Jean Bacon. Information Flow Control for a Medical Web Portal. In *e-Society 2013*. IADIS, 2013.

Thomas Pasquier, Jean Bacon, and David Eyers. FlowK: Information Flow Control for the Cloud. In *International Conference on Cloud Computing Technology and Science (CloudCom'14)*. IEEE, 2014a.

Thomas Pasquier, Jean Bacon, and Brian Shand. FlowR: Aspect Oriented Programming for Information Flow Control in Ruby. In *International Conference on Modularity*. ACM, 2014b.

Thomas Pasquier, Jatinder Singh, and Jean Bacon. Information Flow Control for Strong Protection with Flexible Sharing in PaaS. In *IC2E, International Workshop on Future of PaaS*. IEEE, 2015a.

Thomas Pasquier, Jatinder Singh, and Jean Bacon. Clouds of Things need Information Flow Control with Hardware Roots of Trust. In *International Conference on Cloud Computing Technology and Science (CloudCom'15)*. IEEE, 2015b.

Thomas Pasquier, Jatinder Singh, Jean Bacon, and Olivier Hermant. Managing Big Data with Information Flow Control. In *International Conference on Cloud Computing (CLOUD)*. IEEE, 2015c.

Thomas Pasquier, Jatinder Singh, David Eyers, and Jean Bacon. CamFlow: Managed Data-Sharing for Cloud Services. *IEEE Transactions on Cloud Computing*, 2015d.

Thomas Pasquier, Jean Bacon, Jatinder Singh, and David Eyers. Data-Centric Access Control for Cloud Computing. In *Symposium on Access Control Models and Technologies*. ACM, 2016a.

Thomas Pasquier, Jatinder Singh, Jean Bacon, and David Eyers. Information Flow Audit for PaaS clouds. In *International Conference on Cloud Engineering (IC2E)*. IEEE, 2016b.

Siani Pearson and Marco Casassa-Mont. Sticky Policies: An Approach for Managing Privacy across Multiple Parties. *Computer*, 44, July 2011.

Colin Percival. Cache missing for fun and profit. In *Technical BSD Conference (BSDCan'05)*. University of Ottawa, 2005.

Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-Fi: Collecting High-Fidelity whole-system provenance. In *Annual Computer Security Applications Conference*, pages 259–268. ACM, 2012.

Donald E Porter, Michael D Bond, Indrajit Roy, Kathryn S McKinley, and Emmett Witchel. Practical Fine-Grained Information Flow Control Using Laminar. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(1):4, 2014.

Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *International Symposium on Microarchitecture*, pages 135–148. IEEE, 2006.

Markus Quaritsch and Thomas Winkler. Linux Security Modules Enhancements: Module Stacking Framework and TCP State Transition Hooks for State-Driven NIDS. *Secure Information and Communication*, 7:7–13, 2004.

Roshan Ramachandran, David J Pearce, and Ian Welch. AspectJ for multilevel security. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software*, volume 20, pages 13–17. ACM, 2006.

Bill Roscoe and MH Goldsmith. What is intransitive noninterference? page 228, 1999.

Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical Fine-grained Decentralized Information Flow Control. volume 44, pages 63–74. ACM, 2009.

Andrei Sabelfeld and Andrew Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Area in Communication*, 21(1):5–19, 2003.

Mohamed Amin Sakka, Bruno Defude, and Jorge Tellez. Document provenance in the cloud: constraints and challenges. In *Networked Services and Applications-Engineering, Control and Management*, pages 107–117. Springer, 2010.

Ravi Sandhu. Separation of Duties in Computerized Information Systems. In *Database Security IV: Status and Prospects*, 1990.

Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

Nuno Santos, Krishna P Gummadi, and Rodrigo Rodrigues. Towards Trusted Cloud Computing. In *Conference on Hot Topics in Cloud Computing (HotCloud'09)*, pages 3–3. USENIX, 2009.

Casey Schaufler. LSM: Generalize existing module stacking. *Linux Weekly News*, 2014.

Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Fast Software Encryption*, pages 191–204. Springer, 1994.

Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. SeLINQ: Tracking Information Across Application-Database Boundaries. In *19th SIGPLAN Conference on Functional Programming*, pages 25–38. ACM, 2014.

David Schultz and Barbara Liskov. IFDB: Decentralized Information Flow Control for Databases. In *European Conference on Computer Systems (Eurosys'13)*, pages 43–56. ACM, 2013.

Edward Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Symposium on Security and Privacy*. IEEE, 2010.

V. Shah and F. Hill. An Aspect-Oriented security framework. In *DARPA Information Survivability Conference and Exposition*, volume 2, pages 143–145, 2003.

Yogesh L Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance in e-Science. *ACM SIGMOD Record*, 34(3):31–36, 2005.

Vincent Simonet and Inria Rocquencourt. Flow Caml in a Nutshell. In *Applied Semantics workshop*, pages 152–165. Information Society Technologies, 2003.

Jatinder Singh and Jean Bacon. SBUS: A Generic, Policy-enforcing Middleware for Open Pervasive Systems. (UCAM-CL-TR-847), 2014. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-847.pdf`.

Jatinder Singh, Jean Bacon, Jon Crowcroft, Anil Madhavapeddy, Thomas Pasquier, W. Kuan Hon, and Christopher Millard. Regional Clouds: Technical Considerations. Technical Report UCAM-CL-TR-863, University of Cambridge, 2014a. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-863.pdf`.

Jatinder Singh, Jean Bacon, and David Eyers. Policy enforcement within emerging distributed, event-based systems. In *International Conference on Distributed Event-Based Systems*, pages 246–255. ACM, 2014b.

Jatinder Singh, Thomas Pasquier, and Jean Bacon. Securing Tags to Control Information Flows within the Internet of Things. In *International Conference on Recent Advances in Internet of Things (RIoT'15)*. IEEE, 2015a.

Jatinder Singh, Thomas Pasquier, Jean Bacon, and David Eyers. Integrating Middleware and Information Flow Control. In *International Conference on Cloud Engineering (IC2E)*, pages 54–59. IEEE, 2015b.

Jatinder Singh, Julia Powles, Thomas Pasquier, and Jean Bacon. Data Flow Management and Compliance in Cloud Computing. *IEEE Cloud Computing Magazine, SI on Legal Clouds*, 2015c.

Kapil Singh, Sumeer Bhola, and Wenke Lee. xBook: Redesigning Privacy Control in Social Networking Platforms. In *Security Symposium*, pages 249–266. USENIX, 2009.

Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux Security Module. *NAI Labs Report*, 1:43, 2001.

Michael E Smoot, Keiichiro Ono, Johannes Ruscheinski, Peng-Liang Wang, and Trey Ideker. Cytoscape 2.8: New features for data integration and network visualization. volume 27, pages 431–432. Oxford University Press, 2011.

Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems*, 20(7):636–651, 2007.

Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 53–60. Australian Computer Society, 2002.

David Strauss. Containers–not virtual machines–are the future cloud. *The Linux Journal*, 228:118–123, 2013.

Lianshan Sun, Jongho Park, and Ravi Sandhu. Towards provenance-based access control with feasible overhead. In *International Conference on Information Science, Electronics and Electrical Engineering*, volume 2, pages 1043–1047. IEEE, 2014.

Rodolfo Toledo and Éric Tanter. Secure and modular access control with aspects. In *12th International Conference on Aspect-oriented Software Development*, pages 157–170. ACM, 2013.

Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994–12000, 2009.

Andrew Tucker and David Comay. Solaris Zones: Operating System Support for Server Consolidation. In *Virtual Machine Research and Technology Symposium*. USENIX, 2004.

Philip Turner. Digital provenance–interpretation, verification and corroboration. *Digital Investigation*, 2(1):45–49, 2005.

Giorgos Tylissanakis and Yiannis Cotronis. Data provenance and reproducibility in grid based scientific workflows. In *Grid and Pervasive Computing Conference*, pages 42–49. IEEE, 2009.

N. Vachharajani, M.J. Bridges, J. Chang, R. Rangan, G. Ottoni, J.A. Blome, G.A. Reis, M. Vachharajani, and D.I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *International Symposium on Microarchitecture*, pages 243–254. IEEE, 2004.

Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos Operating System. *ACM Transactions on Computer Systems*, 25(4), 2007.

John Viega, JT Bloch, and Pravir Chandra. Applying Aspect-Oriented Programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.

Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *International Joint Conference Theory and Practice of Software Development (TAPSOFT)*. CAAP/FASE, 1997.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

Dean Wampler. Aquarium: AOP in Ruby. In *Aspect Oriented Software Development (AOSD)*, volume 4, 2008.

Robert NM Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. volume 7, pages 1–8. USENIX, 2007.

Simon Woodman, Hugo Hiden, Paul Watson, and Paolo Missier. Achieving reproducibility by combining provenance with service and workflow versioning. In *workshop on Workflows in Support of Large-scale Science*, pages 127–136. ACM, 2011.

Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Foundations of Intrusion Tolerant Systems*, pages 213–213. IEEE, 2003.

Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan De Halleux, and Michal Moskal. User-aware privacy control via extended static-information-flow analysis. In *International Conference on Automated Software Engineering*, pages 80–89. ACM, 2012.

Kui Xu, Huijun Xiong, Chehai Wu, Deian Stefan, and Danfeng Yao. Data-provenance verification for secure hosts. *IEEE Transactions on Dependable and Secure Computing*, 9(2):173–183, 2012.

Shouhuai Xu, Qun Ni, Elisa Bertino, and Ravi Sandhu. A characterization of the problem of secure provenance management. In *International Conference on Intelligence and Security Informatics (ISI'09)*, pages 310–314. IEEE, 2009.

Arturo Zambrano, Alejandro Alvarez, Johan Fabry, and Silvia Gordillo. Aspect Coordination for Web Applications in Java/AspectJ and Ruby/Aquarium. In *International Conference of Chilean Computer Society*. Chilean Computer Society, 2009.

Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Linux Symposium*, page 494, 2003.

Angeliki Zavou, Vasilis Pappas, Vasileios P Kemerlis, Michalis Polychronakis, Georgios Portokalidis, and Angelos D Keromytis. Cloudopsy: An autopsy of data flows in the cloud. In *Human Aspects of Information Security, Privacy, and Trust*, pages 366–375. Springer, 2013.

Steve Zdancewic and Andrew Myers. Secure Information Flow and CPS. In *European Symposium on Programming Languages and Systems*, pages 46–61. Springer-Verlag, 2001.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 19–19. USENIX, 2006.

Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing Distributed Systems with Information Flow Control. In *Symposium on Networked System Design and Implementation (NSDI'08)*, pages 293–308. USENIX, 2008.

Chi Zhang, Jinyuan Sun, Xiaoyan Zhu, and Yuguang Fang. Privacy and security for online social networks: challenges and opportunities. *IEEE Network*, 24(4):13–18, 2010.

Jing Zhang, Adriane Chapman, and Kristen Lefevre. Do you know where your data has been? Tamper-evident database provenance. In *Secure Data Management*, pages 17–32. Springer, 2009.

Wenchao Zhou, Eric Cronin, and Thau Loo. Provenance-aware secure networks. In *International Conference on Data Engineering*, pages 188–193. IEEE, 2008.

Appendix A

Augmenting web applications with Information Flow Control

This appendix is based on the following published work [Pasquier et al. 2013; 2014b] and introduces early work on adding Information Flow Control to existing Ruby applications.

A.1 Background

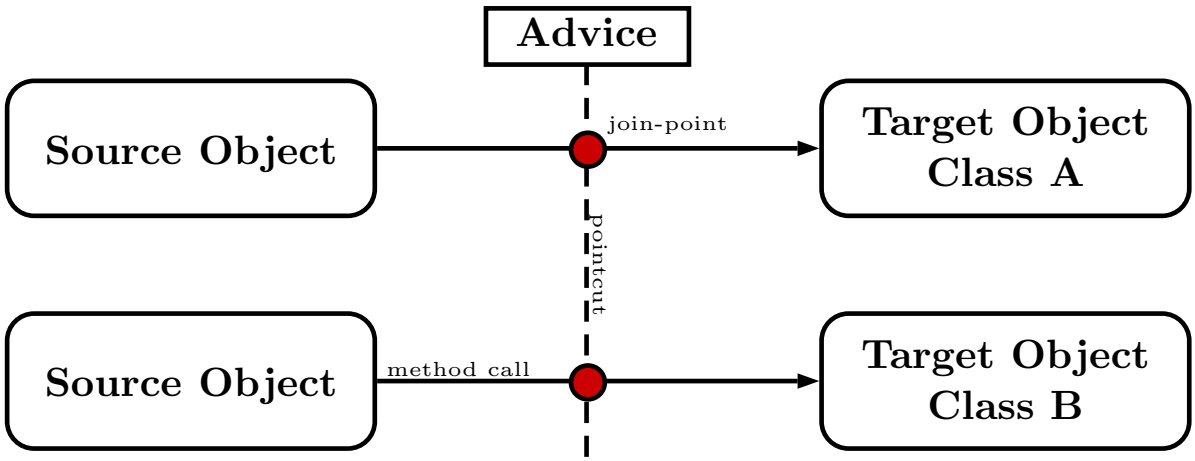

Figure A.1: Visual representation of an aspect.

Aspect Oriented Programming was introduced in [Kiczales et al. 1997]. It is a programming paradigm extending Object Oriented Programming (OOP) by allowing cross-cutting aspects to be expressed. An aspect is a piece of code named an *advice* together with a *pointcut* determining when it should execute. The pointcut is used to determine the join-points (object methods) where the *advice* code will be executed, as shown in Fig. A.1. In the original specification an *advice* could be executed either *before* or *after* the join-point code is executed. The paradigm was later extended with an *around advice* [Kiczales et al. 2001] which has control over whether or not the join-point code should be executed.

```ruby
around method: :write, type: File
    do |join_point, object, *args|
  puts 'hello'
  returned_value = join_point.proceed
  puts 'goodbye'
  return returned_value
end
```

Listing A.1: An Advice in Ruby using Aquarium

An *advice* is composed of a primitive to express when the *advice* should be executed (i.e. *before*, *after*, *around*), a pointcut describing where the *advice* should be executed and a block of instructions to specify the behaviour of the *advice*. This is shown in Listing A.1, using the Aquarium Ruby library [Wampler 2008], where we define an *advice* to be executed around a call to the method *write* of instances of *File*. The parameters passed to the *advice* are the *join-point* to be executed, the *object* the method belongs to and the *arguments* passed to the method.

A pointcut can be made more expressive by using a regular expression (some implementations may not provide this, however this is provided by the framework used in this work) to define the methods and classes to which the *advice* should be applied. A list of methods to be ignored, and special behaviour when exceptions are raised can also be specified.

AOP has been used to implement security features: access control [Ramachandran et al. 2006, Toledo and Tanter 2013], error detection and handling [Lippert and Lopes 2000], automatic login [Viega et al. 2001], hardening the security of existing libraries [Mourad et al. 2008] or preventing buffer overflow [Shah and Hill 2003]. For example, Ramachandran et al. [Ramachandran et al. 2006] proposed to implement access control using AspectJ [Kiczales et al. 2001] around object method calls. In their work all threads and objects are associated with a certain level of clearance. If the current thread level of clearance matches the object on which the program is trying to perform a method call, then the program executes, otherwise it fails. AOP used to improve security is a powerful tool as it allows the expression of concerns that should apply to the whole application while completely decoupling their specifications from the application functionalities.

A.2 Enforcing IFC with AOP

IFC is used to constrain the flow of information within a system (see Chapter 2). In this chapter, we put a particular focus on the aspects of IFC relating to enforcement within a single application rather than a distributed, multi-application environment.

In the DEFCon project [Migliavacca et al. 2010], AOP was used with Java to enforce IFC by inserting IFC policy around selected methods. In FlowR, we extend these ideas by

providing IFC at the level of objects, classes and methods, and provide basic primitives to enforce IFC. While implemented in Ruby (see §A.3), the proposed approach is not specific to it and can be used with any object oriented language that supports AOP. Furthermore, our techniques can work with arbitrary libraries, without programmers having to know about their inner workings, so requiring little effort from them. Our proposed framework provides IFC on all objects, classes and methods.

The proposed framework has the notions of method labels, object labels and class labels (most previous work on dynamic IFC enforcement focused on functions/methods and primitive types). Object labels are associated with a particular instance of a class, while class labels are associated with all instances of the class or inherited class. Finally, method labels are associated with a particular method of an object or class.

In OO languages classes inherit from their parents. To maintain this logic, the labels defined in a parent class are inherited by its children. Similarly, an object inherits the labels of its class and a method inherits the labels of its object or class (in the case that this is a static method). As for methods, attributes of an object inherit its labels. Variables defined in a method inherit the labels of the method. This inheritance relationship also implies that a class, object or method can only be further restricted than its "parents". The current implementation in Ruby only supports multilevel hierarchical inheritance. However, the model could be extended to support multiple inheritance if implemented in a language that supports this feature.

Flow of information occurs through the assignment operator or through method calls. As labels are part of an object, the assignment operator is straightforward. Using the AOP *around* advice we have IFC constraints verified *before* and *after* the *join-point* is called. *Before* we verify that the parameters $p_1, ..., p_n$ (Ps) information is allowed to flow to the methods. *After* we assign the proper label the returned value r and apply declassification and/or endorsement label operations associated with the method. We then verify that the returned value is allowed to flow to the caller C. Indeed, r may be the result of computation involving a variable with more constraints than M. If there is no returned value, this phase is skipped. This is described more succently in algorithm 1.

A.3 Implementation

Our framework is implemented on top of the AOP library Aquarium for Ruby [Wampler 2008]. Advices are placed around objects' methods, which enforce IFC regardless of the actual object implementation.

Table A.1 describes the API required to enforce IFC constraints as described in Chapter 2 and §A.2). We have instructions to start and stop IFC enforcement. Indeed, in some situations, it may be required to activate enforcement only on some portion of

Algorithm 1 IFC enforcement as around advice.

function AROUND(O, C, Ps, *join_point*)
  if $\neg ALLOW(C, M)$ **then** ▷ before
    FAIL
  end if
  for all p_i in Ps **do**
    if $\neg ALLOW(p_i \rightarrow M)$ **then**
      FAIL
    end if
  end for
  $r = join\_point.execute$ ▷ joint-point
  $S(r) := S(r) \cup S(M)$
  $I(r) := I(r) \cup I(M)$
  $ENDORSE\_DECLASSIFY(r, M)$
  if $\neg ALLOW(r \rightarrow C)$ **then** ▷ after
    Fail
  end if
  return r
end function

```
1 FlowR.start_enforcement
2 FlowR.protect_object $stdout, :for_stdout, nil
3 puts 'nothing happens here' # no problem here
4 s = 'I can say that!'
5 s.add_integrity_tag :for_stdout
6 puts s # no problem here
7 password = '123456789'
8 password.add_secrecy_tag :credential
9 puts password # here the program fails
```

Listing A.2: Applying flow constraints on standard output.

the code. For example, the loading of a large configuration file could be done before the enforcement is activated in order to improve performance (note that we assume single threading in such a scenario). Similarly, execute_procedure_unenforced allows a single procedure to be executed with IFC enforcement deactivated (we discuss performance implications in §A.5). Although unenforced procedures are executed in Ruby safe mode, the programmer is relied upon to understand the IFC implication of executing a portion of code outside of the IFC enforcement.

Table A.2 presents the methods added to all objects in order to manipulate their security context. There are methods to add secrecy/integrity tag(s), methods to access the security context of an object and finally a method to remove a particular tag from an object.

Listing A.2 illustrates basic functioning. The last flow is prevented as the subsetting constraint on the secrecy label is not respected. A developer should be able to design an

FlowR API call	Description
start_enforcement	Start IFC enforcement.
stop_enforcement	Stop IFC enforcement.
protect_class / protect_classes	Protect all public method of a class(es).
protect_object / protect_objects	Protect all public method of an instanc(es).
protect_methods_in_class	Protect a defined set of methods in a class.
protect_methods_in_object	Protect a defined set of methods in a single instance.
context_change_in_class	Apply context change (i.e. endorsement/declassification) on returned value of the specified method of the class.
context_change_in_object	Apply context change (i.e. endorsement/declassification) on returned value of the specified method of the object.
execute_procedure_unenforced	Allow a procedure to execute without variable tracking for performance reasons detailed in section A.5.

Table A.1: General FlowR API

Object methods	Description
add_integrity_tag/add_integrity_tags	Add a single or a set of tags to the integrity label associated with an object instance or class depending on the context of the call.
add_secrecy_tag/add_secrecy_tags	Add a single or a set of tags to the secrecy label associated with an object instance or class depending on the context of the call.
declassify	Remove specified tag from the integrity or secrecy label.
get_secrecy_label/get_integrity_label	Get the secrecy or integrity label associated with the object/class

Table A.2: New objects method to manipulate security context.

application without initially being concerned about IFC, and with the ability to use a legacy library that was built without IFC in mind. Once the application is developed, the original developer, or an expert, can add IFC rules to ensure that the application behaves correctly with respect to information flow.

Listing A.3 shows the use of the API to protect credentials. In this simple example we consider how to protect the user password from being disclosed unintentionally within our application by printing it out "in clear" in the log, displaying it on a page or saving it "in clear" in a database. We first add a method which is executed before the processing of any request received from a client. In this method we associate with the parameter password sent by the client, the secrecy tag credential. We also specify that the method Digest.digest declassifies the secrecy tag credential. Credentials are allowed to go through

```ruby
before do
  params[:password].add_secrecy_tag :credential
    unless params[:password].nil?
  params[:verify_password].add_secrecy_tag :credential
    unless params[:verify_password].nil?
end

FlowR.context_change_in_class
  ([:digest], Digest::Class, {credential: false}, nil)
FlowR.start_enforcement
```

Listing A.3: Preventing password leak with FlowR

an IO (in Ruby any external connection: file, pipe, socket etc.) only after the application of the function digest which removes the secrecy tag credential.

We were able to express policy to protect the user password in six lines of code, with minimal knowledge of the application implementation and without modifying the functional implementation. In addition, we also successfully separated security concerns from the implementation itself.

We now look at another example. In this case a user class is trying to access an order made on a website and stored in a database. In addition to the usual information associated with the order, we maintain in our database the label associated with each entry. When writing to or reading from the database, we ensure that the labels associated with instances of orders are propagated to the database by modifying the ActiveRecord::Base implementation. Listing A.4 shows a simplified implementation. Again, here we do not need to modify the original implementation of ActiveRecord or its children, and IFC constraints can easily be added after application development.

It is also possible to assign labels to each attribute. This would represent the different secrecy and integrity requirements of the different fields of a structured document. For example, medical records might be shared between medical professionals and social services. Some sensitive information such as HIV status may be restricted to medical professionals only, while more general information may be accessible to social services, for example to detect signs of child abuse. The modifications to make to ActiveRecord are slightly more complex, but restricted to a few dozen lines of additional code (again no modification of the original implementation is required).

A.4 Use case: building a medical web portal

In 2012 we developed a web portal, in collaboration with the Eastern Cancer Registry and Information Centre (ECRIC) to grant brain cancer patients access to their records [Pasquier et al. 2013]. ECRIC led the amalgamation of all the eight English regions' registries to form the English Cancer Registry of Public Health England. Our original implementation

```
module ActiveIFC
  def before_save
    secrecy_label = self.get_secrecy_label
    integrity_label = self.get_integrity_label
    # save labels to database
  end

  def after_initialize
    # read label from database
    FlowR.protect_object self, secrecy_label, integrity_label
  end

  def after_create
    secrecy_label  = self.get_secrecy_label
    integrity_label = self.get_integrity_label
    # save labels to database
  end
end

class ActiveRecord::Base
  include ActiveIFC
end
```

Listing A.4: Integrating IFC in ActiveRecord

relied on taint tracking using RubyTrack from the SafeWeb project [Hosek et al. 2011].

In 2012, all cancer patients within an English administrative region had their data stored in a data centre managed by their regional cancer registry. Patients within the Eastern Region who have a brain tumour can opt to have their data made accessible to them on an external website managed by the BrainTrust charity.

The data of these patients are encrypted with a unique key per patient. The keys are stored in a dedicated key server, while the individual patients' anonymised medical data, in transit to them, are stored in a separate server. Any patient-provided data is also held separately, thus maintaining a clear separation between patient data associated with the web portal application and the local image of the data held by the cancer registries. Furthermore, patients are invited periodically to respond to a quality of life survey, in order to track the evolution of their condition over time. Those data are regularly retrieved and added to the Cancer Registry's database to improve statistical data about the patient.

We ensure through the use of IFC, that even in the case of unexpected program behaviour, the integrity of patient data is assured, and patients can access only their own data. That is, we ensure isolation of data per purpose and per user. A single request of the data store can manipulate only data for one patient; moreover, the medical data is anonymous and the associated personal data is held separately in isolation.

The requirement for isolating data per patient is an obvious necessity as we want to ensure that patients can only access or amend their own data. The separation and isolation of medical and personal data for a single patient is there to decrease the risk

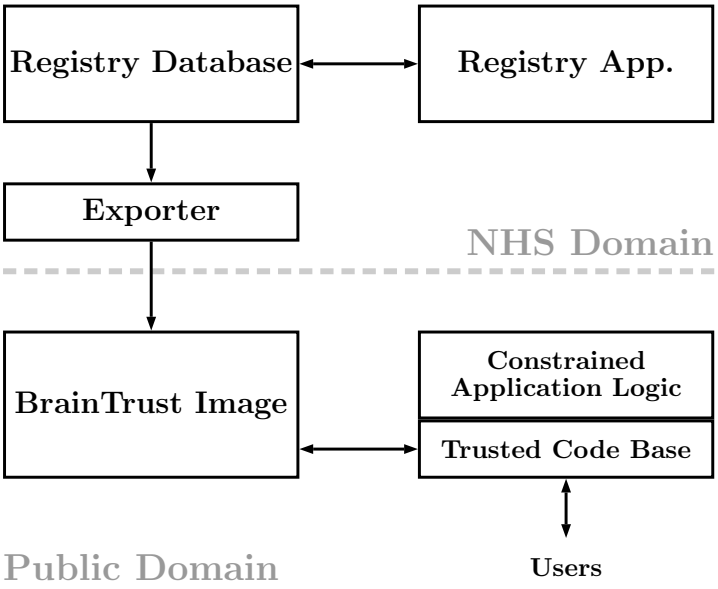

Figure A.2: Data store architecture.

of re-identification. Indeed, very little information is required to uniquely identify an individual [Benitez and Malin 2010]. Through encrypting data with a unique key per patient and per usage, and through ensuring isolation of information per patient and per usage, we reduce this risk.

Fig. A.2 shows the architecture of our data store. The trusted code base (TCB) associates with an authenticated user's data the appropriate labels for the request's context (i.e. medical/personal). In addition, our database models and our controllers have their own set of IFC constraints.

We create an isolation bubble by limiting application access to IO classes according to the user context labels and controller labels (in a similar fashion as shown in §A.3). In order to propagate labels into and out of the database we store the labels along with the record, i.e. in a row in the database. We do not support an individual label per column (record field), only per database entry. We intercept database read and write method calls using the ActiveRecord library feature and add the necessary IFC labelling. Further, the exporter assigns the label corresponding to a patient (e.g. $S = \{medical, alice\}$), when the data is copied from the Registry database to the BrainTrust image.

Supporting IFC was again done separately from building the actual application, allowing a clear distinction and separation between functional and security concerns.

A.5 Evaluation

Our tests measure the performance of our solution, FlowR, compared with an equivalent solution, that extends native Ruby with RubyTrack, developed for the SafeWeb

	RubyTrack	FlowR
Label	a single label	integrity and secrecy
Tag	simple string	symbol + capability
Enforcement	manual by developer at strategic points	at public method call on tracked objects
Engineering	requires overwriting of classes that need to be tracked	minimal

Table A.3: Feature comparison of FlowR and RubyTrack

project [Hosek et al. 2011]. It is important to note the feature differences that explain the performance difference of FlowR when compared with RubyTrack, as illustrated in Table A.3.

Our first series of tests concern compute-intensive tasks. We demonstrate that FlowR does not perform significantly worse than its equivalent using RubyTrack. In addition, no performance optimisation has been attempted for FlowR, which is beyond the scope of this work.

Our second series of tests is made on a web application, built to provide patient medical records and similar to the one described above [Pasquier et al. 2013]. We demonstrate that the performance loss compared with native Ruby is of the same order as the earlier implementation, and acceptable from an end user point of view.

All tests have been performed on an i7 2.2GHz 6Gb RAM Fedora 17 GNU/Linux Machine.

A.5.1 Compute-intensive tasks

We designed two simple tests. The first consists of counting the number of words in text stored in a file on disk ("Les Contemplations" by Victor Hugo). The second test consists of calculating the first n prime numbers. The execution time of the native Ruby code is our time unit. We compare RubyTrack, FlowR and FlowR using untracked procedure calls (Section A.3).

test	native	RubyTrack	FlowR	untracked
word count	1	6.3	9.8	7.7
prime	1	27	70	1.8

Table A.4: Performance comparison of compute-intensive tasks

The results, shown in Table A.4, show the same order of magnitude for RubyTrack and FlowR. We did not attempt to optimise performance, and the Aquarium library is known to suffer from performance issues [Zambrano et al. 2009]. This is because, at

present, Aquarium applies advices at runtime whereas AspectJ [Kiczales et al. 2001] and AspectC++ [Spinczyk et al. 2002] apply them at compile time. Furthermore, it is commonly accepted that performing IFC is inappropriate for compute-intensive tasks. Using untracked procedure calls provides much better performance. This figure includes the switching of tracking on, off and on again which induces some overhead. However, this overhead becomes negligible as the execution time becomes large. Therefore, untracked procedure calls can provide performance identical to native Ruby in the case of long compute-intensive tasks.

We also measured the execution time for some key primitives which were: starting tracking, 325 ms; stopping tracking, 108 ms; protecting an additional class, 180 ms; adding protection to a single method, 5 ms. As mentioned above, adding AOP *advices* at runtime, as in Ruby/Aquarium, incurs performance overhead, and care should be taken in deciding when this is necessary. IFC advices should be added during initialization as much as possible.

On the other hand, the cost of adding a label to an existing object is insignificant (it is simply adding an entry to a hash table). Therefore, adding or removing labels during the lifetime of an application does not amount to a significant performance loss.

A.5.2 Web application

test	native	RubyTrack	FlowR
hello world	4.1 ms	4.4 ms (+7%)	4.6 ms(+12%)
medical record	62 ms	68 ms (+10%)	71 ms (+15%)

Table A.5: Performance comparison for a web portal

In order to evaluate our library under realistic conditions we used the data store described in Section A.4. In order to evaluate the performance of our implementation we queried our data store 1000 times, asking for 50 different, randomly chosen data items. We compare the averaged values obtained with native Ruby, RubyTrack and FlowR, as shown in Table A.5.

We used the "thin" Ruby web server as it provides quite good performance. We first display an unlabelled static page to measure the influence of tracking without flow enforcement. RubyTrack and FlowR add an overhead of 7% and 12% respectively compared to native Ruby. The performance penalties for retrieving a medical record from our database are of the same order (10% and 15% respectively).

We add the IFC advice at initialization; the web server executes the initialization script only once. This removes the very significant overhead generated when creating the advices. Furthermore, as discussed previously, our tracking algorithm is slightly more

complicated than RubyTrack and flows are controlled for every protected object (including basic variables), while RubyTrack only enforces flow at strategic points. This explains our performance decrease compared to RubyTrack.

A.6 Summary

In this appendix we demonstrated how Ruby applications can be augmented with IFC constraints through Aspect Oriented Programming without the need to modify existing code. We believe that the separation between the application logic and the IFC policy to be a step forward in comparison to other approaches (such as JFlow [Myers 1999]) that force the programmer to annotate the code. While the implementation focused on Ruby the principle and approach presented here is more generally applicable.

Appendix B

Code example

```
int public, secret; // file descriptors
uint8_t value, i, tmp;
pid_t child;
// open public file in parents
public = open("public.dat", O_RDONLY);
child = fork();
if(child==0){ // child process
  // in CamFlow change security context in order to read secret file
  // add_secrecy(SECRET);
  // open secret file in child
  secret=open("secret.dat", O_RDONLY);
  // we read the value from the secret file
  read(secret, &value, sizeof(value));
  close(secret);
  // read (value-1) times from public
  for(i=0; i<value-1; i++){
    read(public, &tmp, sizeof(tmp));
  }
}else if(child>0){ // parent process
  // we wait to be sure the child finished its job
  sleep(1);
  // we read the value from public at current offset
  read(public, &value, sizeof(value)); // fail with CamFlow
  // this was the secret
  printf("The secret was %u\n", value);
}
close(public);
```

Listing B.1: Leaking data through a shared file descriptor as presented in §3.3.2.