Floodgate: An Information Flow Control Platform for Distributed Mobile Applications

Sérgio Marques Moura

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisor: Prof. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof. Paulo Jorge Pires Ferreira
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. Ricardo Jorge Fernandes Chaves

November 2015

Acknowledgements

I would first like to sincerely thank Professor Nuno Santos his guidance during this last year of work, in which he really contributed with his extreme knowledge and motivation.

Second, I must thank my family for supporting me, motivating me and giving me the opportunity to learn in this institute.

I would also want to thank my girlfriend for her patience and motivation.

Finally, I have to thank all my friends and colleagues for always being by my side during the hard, but amazing, time spent at this Institute.

Lisboa, November 2015

Sérgio Marques Moura

For my parents and sister,

Resumo

Tipicamente, serviços *web* e aplicações móveis suportadas pela *cloud*, obrigam os seus utilizadores a confiar os seus dados sensíveis aos fornecedores desses serviços em troca de funcionalidades desejadas. No entanto, falhas de segurança podem ocorrer devido à presença de *bugs* no código das aplicações, que podem levar a que estes dados sensíveis sejam exportados de forma indesejada. Este problema tem sido abordado utilizando técnicas de controlo do fluxo de informação, que almejam assegurar que código potencialmente perigoso não viola as políticas de segurança definidas pela aplicação e aceites pelos utilizadores. No entanto, os sistemas de controlo do fluxo de informação existentes não conseguem fornecer uma solução ponto-a-ponto entre o lado servidor e móvel das aplicações, uma vez que se focam na sua grande maioria em aplicações estritamente *web* (e.g. *web sites*). Com isto em mente, apresentamos o Floodgate, um *middleware* que combina técnicas de controlo do fluxo de informação em ambos os lados móvel e servidor, para alcançar segurança dos dados ponto-a-ponto, desde que são acedidos pelas aplicações móveis até que são processados e guardados nos servidores destas aplicações. Comparamos o Floodgate com os sistemas existentes e descrevemos a sua arquitectura.

Abstract

Typically, web and mobile cloud-supported services, require users to entrust their sensitive data to service providers in exchange for desired functionality. However, security breaches can occur in the presence of *bugs* in applications' source code which might lead to undesired sensitive data leaks. This problem has been addressed using Information Flow Control (IFC) techniques, which aim at ensuring that untrusted code cannot violate user-defined access control policies. However, existing IFC systems fail to provide an end-to-end solution for both mobile and server sides of services, since they focus exclusively on web applications. With this in mind, we present Floodgate, a *middleware* system that combines IFC techniques on both mobile and server sides, so as to achieve end-to-end security between the point where the sensitive data is accessed by mobile applications on the user's device until the data is processed on the servers of the supported *cloud backend*. We compare Floodgate with current existing systems and describe its design.

Palavras Chave
Keywords

Palavras Chave

Controlo de Fluxo de Informação

Privacidade

Dados Sensíveis

Aplicações Móveis

Segurança Ponto-a-ponto

Keywords

Information Flow Control

Privacy

Sensitive Data

Mobile Applications

End-to-end Security

Contents

List of Figures

List of Tables

Acronyms

GPS: Global Positioning System

API: Application Programming Interface

IFC: Information Flow Control

AOP: Aspect-Oriented Programming

SIF: Servlet Information Flow

HTTP: HyperText Transfer Protocol

HTML: HyperText Markup Language

CIFC: Centralized Information Flow Control

DIFC: Decentralized Information Flow Control

SP: Security Policies

IMEI: International Mobile Equipment Identity

GSM: Global System for Mobile Communications

REST: Representational State Transfer

CRUD: Create, Read, Update, Delete

SDK: Software Development Kit

XML: Extensible Markup Language

HTTPS: HyperText Transfer Protocol Secure

URL: Uniform Resource Locator

JSON: JavaScript Object Notation

GB: GigaByte

RAM: Random Access Memory

GHz: GigaHertz

JVM: Java Virtual Machine

SIM: Subscriber Identity Module

IMSI: International Mobile Subscriber Identity

SSID: Service Set Identifier

PDF: Portable Document Format

XSL-FO: Extensible Stylesheet Language - Formatting Objects

AVR: Automatic Voltage Regulator

SQL: Structured Query Language

VM: Virtual Machine

TCB: Trusted Computing Base

IPC: Interprocess Communication

IP: Internet Protocol

SIF: Servlet Information Flow

MPVC: Model, Policy, View, Controller

MVC: Model, View, Controller

1 Introduction

In today's era, users of mobile devices are experiencing a flood of new mobile applications supported by online services through the internet. Such applications serve users by providing useful functionality that facilitate some common daily tasks, such as sharing photos [1], purchase items on online stores [2] or auctioning services [3], obtain movie [4] or restaurant recommendations, between others. A common feature among many of such apps is that they require users to entrust sensitive data to the backend service. For example, Yelp [5] access the user location provided by the device's GPS sensor and send this data over the network to their backend. The backend then returns a list of nearby restaurant suggestions to the user. Google Maps [6] may also use the user's location and a specified destination to calculate route alternatives and provide real-time directions between both places. However, despite application providers' best efforts in guaranteeing the security of their users' data, it is difficult to ensure the correctness of applications. In fact, there can be *bugs* in applications that might lead to the leakage of this sensitive information. In addition, the privacy policies applied by the service providers on the the users' data are usually very confusing and written in a complex way, making users misunderstand or simply overlook them. Moreover, such policies are normally specified in plain English making it hard to be interpreted by computers and consequently difficult to be enforced in practice.

1.1 Motivation

In this thesis, we propose to improve data privacy in distributed mobile applications by leveraging information flow control (IFC) [9] techniques. More specifically, we envision a dis-

[1] http://www.instagram.com
[2] http://www.amazon.co.uk
[3] http://www.ebay.co.uk
[4] http://www.imdb.com
[5] http://www.yelp.com
[6] http://maps.google.com

tributed system that guarantees the security of private user's data since when it is accessed at sources located on mobile devices (e.g. GPS location) to when this information is sent over the network to the server and when accessed there. To support privacy policies that are simple to understand by both humans and computers, we propose policies that state how certain types of data (e.g. the device's IMEI or phone number) may be accessed: *private* - only by the mobile application; *protected* - by the mobile application and its corresponding *backend*; or *public* - with no restrictions on this. Our envisioned system would track this information and enforce the specified policies on both the mobile and server sides.

This way, we conclude that it is necessary to extend the IFC protection between the mobile and the server endpoints. Currently, there are IFC systems working exclusively on data protection on mobile applications and others achieving data security on server-side applications. There are some cases which achieve protection on both sides, but suffer from limitations of compatibility with current widely-used mobile platforms or very high complexity in policies' definition. So, there is the need of tools which might work in both sides, and despite we could take one system of each side and build an end-to-end system, this raises some challenges:

In first place, to perform fine-grained (highly accurate) end-to-end IFC taint tracking while preserving the compatibility with current platforms. On mobile-side, systems like TaintDroid [1] obligate users to alter the underlying Android operating system. This turned into a limitation since we could not apply our system to out-of-the-box Android devices, requiring them to be firstly modified with TaintDroid-ready operating system builds. In the end, adopting a solution based on Taintdroid would undermine the practical adoption of our system.

In second place, these policies are often very complex for users to understand which rules they will apply to data and also may lead to programmers committing errors which can result in the leakage of sensitive data. This complexity is evident in SIF [4], a *framework* to develop web applications which implements dynamic taint-tracking at language-level, whose policies are specified throughout the code as exemplified in Listing 1.1. This Listing represents a simple policy associated with a method that comprises the invocation of a given action on the server. For example, the policy might define how data entering the web application, like the client's IMEI, is allowed to be accessed on the server.

Detailing the information contained on the method's header, the first strange element is the `{*lbl}` after the method's name. This element is the begin label of the method, which

```
public void invoke{*lbl}(label{*lbl} lbl, Request[HellloServEP]{*lbl} req)
    where caller(req.session),
    {*lbl} <= {*:req.session} {
   ...
}
```

Listing 1.1: SIF security policy example

specifies an upper bound on the method's caller (the caller's program counter cannot be more restrictive that this label). Also, the begin label specifies a lower bound on the method's side effects (during the method's instructions only information at least as restrictive may be updated). Following, the annotation `where caller(req.session)` is another way for the method to acquire authority for the method's execution, meaning that it can only be called from code that is statically known to have the authority of principal `req.session`, which must have an established session with the application. Finally, the annotation `{*lbl} <= {*:req.session}` states that this method's call is only allowed if the label `lbl` is no more restrictive `(<=)` than the label `{ _:_; *:req.session}`, which has no confidentiality concerns but states that only the principal representing the session owner and principals who can act-for him, apart from the top principal, are allowed to alter information with the specified label. Another SIF limitation is the fact that like in most language-based IFC systems, SIF applications have their security policies specified alongside the code. That means that every variable and method along the code is annotated with Jif labels, which makes it hard for the programmer to modify security requirements on the application, requiring the whole code auditing even for minor changes.

Finally, a third challenge regards the amount of computation resources consumed by the system. Logically, the more fine-grained the IFC system is, the higher the taint tracking accuracy, but also the higher the impact it has on the applications' memory and CPU usage. Taking into account our mobile-server architecture, in Floodgate we must have two criteria for overhead acceptance. Despite we can accept some considerable consumes on the server-side, we must avoid this on the more resource-constrained mobile-side, which could lead to bad behavior of mobile applications. This way, we intend to perform more fine-grained tracking on the server-side than on the mobile-side.

1.2 Goals

The goal of this project is to implement a distributed IFC system which allows to control the data flow in distributed mobile applications, according to access control policies defined by the programmers. Also, we have the additional requirements:

- **Compatibility** with current Android mobile platform (min. API 15) thus not requiring changes on the mobile OS;
- **Simple policy specification methods**, in a way easy to express by the developers and understandable by the users;
- **Reduced impact** on the application model, requiring as less as possible re-engineering effort to the developers;
- **Efficient** use of resources, specially on the mobile-side.

Also, as every IFC system, we accept that Floodgate introduces some performance overhead in its applications. Although, it must be controlled so that it does not negatively modify the user application experience. To address these requirements, we admit to relax the IFC taint tracking granularity, specially on the more resource-constrained mobile-side, in order to lower the incurred overhead and the allow the desired compatibility with current Android platform.

We assume that application providers are not explicitly malicious when offering a service to the user. Although, applications or other third-party services or libraries in which they trust may contain either vulnerabilities intentionally developed by malicious developers or implementation and design bugs that might compromise user's sensitive data. Our focus is on providing data-privacy protection against this type of vulnerabilities, providing reliability to applications.

1.3 Contributions

This thesis makes the following contributions:

- The design of a distributed IFC system called Floodgate, which will stand on an architecture that operates at the *middleware*-level on both mobile and server endpoints. Floodgate

tracks the information flow across the application using a combination of two techniques, aspect-oriented programming (AOP) [10] and dynamic taint tracking using static source code instrumentation [11]. AOP entails breaking down program logic into "concerns" (cohesive areas of functionality). With AOP, developers can add executable blocks to some source code without explicitly changing it. This programming paradigm pretends that "cross-cutting concerns" (the logic needed at many places, without a single class where to implement them) should be implemented once and injected it many times into those places. Dynamic taint tracking, by its way, allows data to be assigned with taints which specify their privacy policies, performing taint propagation when tainted data interacts with other data (i.e. variables assignment, method calls). We offer coarse-grained and fine-grained taint tracking on the mobile and server sides, respectively, with some optimizations on the mobile-side in order to enhance its granularity. Also, we define our own communication model between the mobile and server endpoints in order to transfer the policies along with data between them. To the best of our knowledge, Floodgate is the first system to combine AOP and dynamic taint tracking through source code instrumentation in order to control of how data may be accessed within the system.

- A publicly available implementation of a Floodgate prototype. For AOP taint tracking on the mobile-side, we apply the AspectJ *framework* [12] on the Android platform. On the server-side, we perform source-code instrumentation using Phosphor [11], a tool which modifies Java code and outputs a taint-tracking-enabled version of the same code. Our prototype supports a set of data types which we consider as sensible data. Currently, every type is a resource of the mobile device (i.e. IMEI, phone number) or of its user (i.e. country, saved accounts). However, our system is designed to support other different data types.

- A detailed evaluation of the system, along with some criticism regarding the presented results, showing Floodgate's efficiency, policies simplicity and security against our threat model. For instance, Floodgate shows an overhead of about 15% on the server throughput and 32% on the mobile-side performance. Also, we developed a completely functional use-case application which shows the end-to-end system's functionality when applied to a real scenario, for instance an auctioning service.

- As a side-effect of this work, we also contributed to the improvement of the Phosphor tool.

Concretely, we contributed to its instrumentation capabilities on Apache Tomcat *servlets*, which were part of a mid-term solution that in the end we did not adopt.

1.4 Research History

During this work, we changed multiple architecture components and implementation choices presented in the project report. This happened due to an *a posteriori* in-depth study of those components. This study revealed some difficulties we weren't aware of, some new technologies we didn't know and also some ideas of how the system we idealized could change for the better.

At first, our system was architecturally based in two essential components: (1) On the **mobile-side**, Floodgate worked at the *middleware-level*, in order to keep applications unchanged. This *middleware* was responsible for enforcing security policies on the mobile-side and propagating them to the backend; (2) on the **server-side**, Floodgate acted as a host for secure containers featuring *servlets* holding the applications' logic. Also, here Floodgate was responsible for enforcing the defined security policies and controlling data sharing with other *servlets* operating under the same container or third-parties operating over the network.

To materialize this design on the server-side, we started by adopting Servlet Information Flow (SIF) [4] to build our secure container. SIF is a *software framework* to develop web applications deployed using conventional Linux distributions and containers like Apache Tomcat. SIF allows web applications to satisfy explicit confidentiality and integrity security policies, ensuring they respect users' security requirements. SIF is designed to support only web applications written in Jif 3.0 [13]. However, after a hands-on with SIF, we identified some limitations of this framework which clearly did not fit our purpose. First of all, defining security policies regarding confidentiality and integrity concerns is a very complex task in SIF. Also, SIF's security policies are spread alongside the code, instead of defined in a separate file. Also, SIF was primarily targeted at applying IFC to purely web applications, instead of native mobile apps that comprise Floodgate's target. Due to both reasons, we concluded that it would not be the best path to follow.

After that, we decided to use Phosphor to instrument Apache Tomcat *servlets* running server-side applications. We built a prototype using this method, but we left it in favor for out final solution due to the fact that it requires more configuration work to instrument and deploy

an application on Tomcat than with Floodgate's current model, which offers an abstraction level that lets developers focus only on developing the application logic.

On the mobile-side, we first started by applying Phosphor to instrument Android applications. Since the Android platform outputs `.dex` files after compilation, we had to *"de-dex"* them, converting to common `.class` Java files, instrument them, and finally *"dex"* them again. However, Android applications are not completely self-contained, and rely on Java libraries which are contained in the device's operating system itself. So, to completely instrument Android applications with Phosphor we would also need to instrument the OS on which applications run, which would violate our requirement of applying Floodgate to unmodified Android devices.

Lastly, before adopting AOP, we tried to use Aurasium, a Python tool that repackages `.apk` Android packages and adds checks on some sensitive information sources (i.e. library to fetch IMEI address or phone number) and sinks (i.e. socket connections, local database writes). Aurasium inserts dialogs to be presented to users whenever such situations are detected, asking whether they want to let it proceed or not. Although, it didn't achieve real taint tracking; instead, it only detected the execution of potentially sensitive methods during the application flow and then presented a warning to the user, which made us give up on this solution.

1.5 Document Roadmap

The rest of this document is organized as follows. Chapter 2 presents the state of the art on IFC, in its mobile-side, server-side and end-to-end strands. Then, Chapter 3 describes the architecture and the algorithms used in Floodgate. Chapter 4 describes the implementation details for the described architecture and the tools we used during this process. Then, Chapter 5 presents the results of our evaluation. Finally, Chapter 6 concludes this document by summarizing its main points and future work in sight.

2 Related Work

In this chapter, we present the related work on IFC systems. We start by presenting an overview of IFC-related topics in Section 2.1. This overview consists in presenting the most relevant IFC models in Section 2.1.1 and 2.1.2, and a discussion of the IFC system design space in Section 2.1.3. Then, we provide an introduction of the most representative IFC systems, covering three main architectures: Operating on the mobile-side (Section 2.2), on the server-side (Section 2.3), and distributed approaches, on Section 2.4.

2.1 Overview of Information Flow Control

Information Flow Control (IFC) [9] is a general technique for tracking how information is propagated throughout the execution of programs or software systems. To provide an introduction on this broad topic, we start by presenting two relevant IFC models – Centralized IFC and Distributed IFC – and then providing an overview of IFC systems' design space.

2.1.1 Centralized Information Flow Control

Centralized Information Flow Control (CIFC) was the first IFC model proposed in the literature. Originally derived from military information management techniques, in which one unprivileged user can give information it owns to a privileged user but cannot have access to information owned by the privileged user ("no-read-up, no-write-down"), CIFC enforces sensitive data protection by associating *security labels* with data and tracking the flow of information inside a program, avoiding the execution of actions that might compromise the confidentiality of this data.

The CIFC model allows for the specification of security policies based on principals and labels. Principals represent entities (e.g., processes or users) that can interact with data. Labels are annotations that address both confidentiality and integrity constrains of data owned

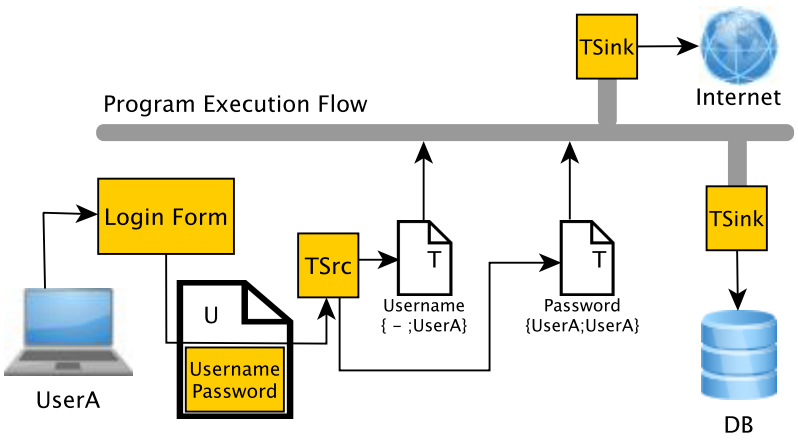

Figure 2.1: CIFC operation scheme

by principals, possibly defining different restrictions to each one. By labelling data, a fine-grained control is achieved through different labels or set of labels for each purpose, forming a pair {`confidentiality,integrity`} assigned to a given data item. Confidentiality labels determine the paths that data is allowed to flow to, while integrity labels constrain the paths where it is permitted to come from. Labels must be attached to data from the beginning and must not be removable or forged without the right permissions. Apart from data labels, users also must have their own labels, in order to compare them against the data labels for permission verification. For example, a label defined as {{`Alice;Bob`},`Alice`} would mean that both Alice and Bob are able to access data for read purposes but only Alice can alter this data. Also, the label { `–` ,`UserA`} (as represented in Figure 2.1) means that the associated data is public for read purposes but only `UserA` is able to alter it.

Figure 2.1 illustrates how CIFC models the flow of information within a system. This process is called labeling or tainting process. It begins when there is an input of data into the system, possibly from direct user introduction (for example, UserA introduces a username and the corresponding password in a login form). This raw input generates untainted data (marked with `U`), which is data not analyzed by the labelling mechanism. Untainted data is automatically forwarded to a taint source (represented as `TSrc`), a system component responsible for assigning a taint marking (represented as `T`) specifying the data type, and a confidentiality and integrity label, converting the raw data into tainted data (marked with `T`). Then, the now tainted data enters the IFC system, and starts being tracked across the program execution flow. As tainted

data items are assigned from variable to variable, it results in exchanging labels to continue security enforcement, either increasing security-level (endorsement) or decreasing it (declassification). Finally, tainted data is verified at taint sinks (represented as `TSink`), normally network interfaces or database connectors, which mean points where data leaves the system, representing potential privacy threats.

So, CIFC allows data-centric security policies to be defined by a central authority, specifying who can read or alter each data item in the system, allowing user's sensitive data to be protected against unauthorized accesses, therefore assuming the central authority responsible for policies definition is trusted and cannot be compromised.

2.1.2 Decentralized Information Flow Control

The CIFC model described in Section 2.1.1 was primarily designed for simple environments where security policies are held and decided by a trusted central authority. However, this model is not so well suited with operations performed across multiple domains. For example, a university where multiple departments coexist, being each department responsible for defining rules for data owned by itself. As we show below, this scenario cannot be modeled by CIFC, because there is no notion of data ownership, being a central authority responsible for controlling every access. Decentralized Information Flow Control [9] is a proposed model that improves CIFC by decentralizing the policy checking and holding from a central authority to the users themselves.

DIFC labels take the form of {`owner:readers`} and extend classic IFC labels by adding a notion of ownership to each label. The owner is a principal or group of principals that is able to declassify its own data but not to change policies held by other principals. Principals may delegate all their power to other principals, through acts-for relations specifying the new principals that can act on behalf of the original one.

Since information can enter or leave the system from/to remote locations, DIFC extends CIFC's policy checks to the communication channels through which data is exchanged, associating them labels that operate on the input and output of data through these channels. When some value is read from an input channel, this value will be initially assigned with the label of the channel. A value can only be written to an output channel if the channel's label is at least as restrictive as the value's label, thereby preventing information leaks through unprivileged

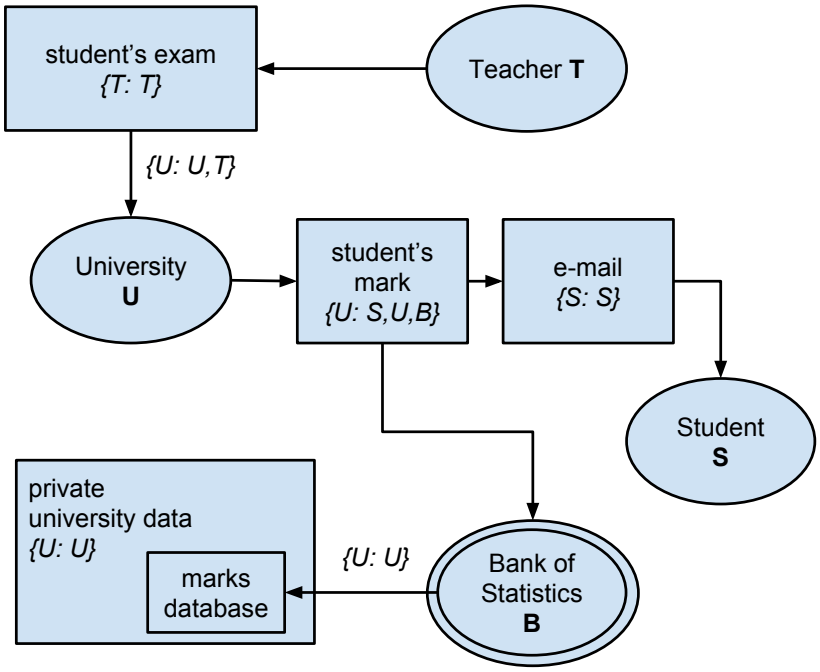

Figure 2.2: DIFC operation example

communication channels.

DIFC base concepts and operation are demonstrated in Figure 2.2, taking as example scenario an university management platform in which their students receive all the exam marks by e-mail. This example illustrates how some common scenarios cannot be modeled by CIFC. The teacher corrects the exams, and when he finishes, he marks the exams as corrected. At this time, the exams are automatically sent to the university secretary that is responsible for sending the e-mails to the students with the corresponding marks. Also, the university has a private statistics bank where all the marks are compiled every year on performance reports that compare the students' success with previous years. This is an example of a scenario where DIFC suits better than simple IFC, since the system is composed by multiple parts that not necessarily trust each other, as for example, a student may want to be sure that his marks are not leaked to other students. Also, a teacher may not want the university to be able to access the exam before he finishes the correction. Modeling these restrictions in a DIFC model, we can find four principals, represent by ovals, which can hold data and need protection for that data: Teachers (`T`), University (`U`), that represents the university secretary, Student (`S`), and the Bank of Statistics (`B`) responsible for compiling and saving the statistics reports and which benefits from special trust by the university and the students, the reason why it's represented

with a double oval. The teacher corrects the exams, and when he finishes, the label {`T: T`} is assigned to the exam, specifying that the teacher is the owner of the exam and also only him is able to read it. When he finishes the correction, he gives the ownership of the exam to the university, but still keeps his ability to consult the exam. The university then takes the mark from the exam, composes an e-mail that is owned and only readable by the student {`S: S`} who will receive it, being sure that his information was not leaked anywhere else. Apart from that, the university makes the mark available for the bank of statistics to read, so it can be stored in the university database. The bank of statistics has the authority to act on behalf of the university (`U`), so it can replace the student mark's policy from {`U: U,B,S`} to the statistics policy {`U: U`}, such that the mark can be saved in the statistics database and only accessible by the university.

In this scenario, there are four situations where data's authority is transferred between principals. In first place, when the teacher transfers the exams ownership from himself to the university to process them. Then, the university composes an e-mail to be sent to each student carrying his mark, declassifying the e-mail to be owned and readable by the student only. At the same time, the university declassifies the mark, making it available for the bank of statistics to read, in order to include it in the yearly report. Finally, the bank of statistics endorses the mark's security, transferring the ownership of the mark to the university again, so it can be stored in the private database. In each situation, we are applying the rule that a principal may only modify the policies owned by itself, not present in centralized information flow control, which is the reason why it cannot model this scenario.

Multiple similarities can be spotted between this scenario that suits the DIFC model for privacy protection and the examples of applications that exchange sensitive information about their users and that we approach as targets for the Floodgate. Foremost, the multiple principals that comprise the scenarios can be compared as current applications that rely on mobile and server components where each one has the responsibility for a part of the whole system functionality. Also, in Floodgate data is exchanged through secure channels between the mobile and server sides. During these exchanges may occur data declassification, like on the example when one principal transfers the authority over some data to another. For these reasons, we argue that the DIFC model best fits our needs in terms of data tracking, policies specification and enforcement, and so must be seen as good basis for the Floodgate system model.

2.1.3 Design Space of IFC Systems

In addition to deciding on the IFC model to adopt, when designing an IFC system it necessary to make several design options. In this section, we base ourselves on the work of Bacon et al. [14] to describe the most important design options that an IFC system architect is faced with, and provide examples of how these options have been made in representative IFC systems. Since Floodgate is itself a distributed IFC system, this study provides us an overview of the decisions we have to take.

1. When the IFC System Operates?

Static methods An IFC system may use a *static* method if its operation occurs at the program compilation time. As examples of static methods we outline the following ones: Static taint analysis, implemented in Pixy [15], a technique that analyses a target program in order to check that every accepted input is sanitized before it can be considered as safe to be applied in the system operation. It prevents attacks like SQL injection or Cross-Site Scripting (XSS) [16]. Security-typed Languages, such as Haskell [17], allow application developers to explicitly define security policies attached to the type of each variable while writing the application code. The compiler enforces to take these confidentiality and integrity policies at program compilation time.

Runtime IFC methods are techniques to achieve data labelling and taint tracking while a program is running, despite they add a runtime overhead due to the taint tracking mechanisms. TaintDroid [1] is an example of a system that uses runtime methods for monitoring Android applications while running. It detects when applications try to exfiltrate data coming from sensitive information sources (i.e., device location sensor, user phonebook), reporting the responsible application, data type and network destination to the user.

Apart from purely static and purely runtime systems exemplified above, there are special cases known as ***hybrid* methods** because of their operation both at compile and runtime. Jif [13] must be seen as a hybrid system that divides its functionalities between both compile and runtime. Jif is an example of a security-typed language which extends Java by adding DIFC labels to the data type system. These labels define a set of rules that a program must follow in order to prevent sensitive data leaks. Then the compiler verifies all program statements based on DIFC labels, which include compile-time checks (static component) of whether data

with a certain associated label reaches another variable or a communication channel associated with a more permissive label, generating compilation errors in such case. Apart from that, Jif allows ownership and permissions over data to be granted and checked by the system in runtime, which is useful in systems where information flow cannot be verified statically and privileges over information may change over time.

When designing Floodgate, we have to decide which of these approaches to take: Static, runtime or hybrid. When evaluating these choices, we must keep in mind that Floodgate is a mobile-server system. Also, its requirements comprise no modifications to mobile operating system and acceptable performance overheads, which gets even stricter on the mobile-side, due to its resource-constrained characteristics.

2. Which Granularity the System Operates at?

IFC tracking can be enforced at different granularities and systems must determine a balanced way of providing effective data tracking while minimizing additional verified overhead.

A) Tracking at VM Level: Considering an architecture where multiple virtual machines controlled by a hypervisor run different applications, there is the need to control how data flows between virtual machines and and between virtual machines and the hypervisor. Payne et al. [18] proposed a layered model for security policies, in which every request between VMs running over the same hypervisor is associated with two labels. One label is assigned by a guard component responsible for inspecting the request type and content, and the other one, derived from the first, is assigned by the hypervisor stating whether the request may reach the destination VM. This distribution is done in order to lower complexity by reducing the relations verified at each layer.

B) Tracking at Process Level: A more fine-grained approach, tracking at process level allows taint tracking to intersect and verify data exchanges between different applications installed on the same device, requiring each one to be encapsulated in its own process. This way, every interaction between different processes is intercepted and analyzed by the data flow tracking system. This scheme is used in Asbestos [19], Flume [20], HiStar [21], Nexus [22], Laminar [23], DStar [24] and almost all DIFC-driven operating systems, but suffers from little adoption since it requires a deep change on the application's development environment and it is not capable of tracking suspicious interactions within the same process as a variable-level approach does.

C) Tracking at Variable Level: Providing data flow tracking at variable level, allows every variable defined in the context of an application to be monitored, preventing unsafe uses. Jif [13] language attaches to each variable a pair of labels comprising a confidentiality label and an integrity label. These labels are checked on each interaction between components and data holding them may be endorsed or declassified, based on the privacy needs at each moment, defining the possible flows an application can securely allow. Even in the presence of incorrect or malicious software implementations, if labelling is correctly defined, a user trying to read another's confidential data will be presented with an exception indicating that an invalid request made. The downside of this technique is the need of more computational power to perform the fine-grained taint tracking, which might result in higher performance overheads.

There are also some other systems, such as the ones operating exclusively at the mobile-side, which usually combine several of these approaches to achieve efficient data tracking. Uppermost, the OS running on mobile devices is slightly different from general Linux distributions. Also, mobile applications must be written in one specific language that provide interfaces to communicate with the device itself, in order to gather information. On Android systems, applications are written in a modified version of Java language, and thus, IFC systems designed to operate over Android [1, 25, 26] know *a priori* which language libraries and variables to track. Apart from variable/library-level tracking, Android-supported IFC systems also perform tracking at interprocess message-level, since each application runs on its own Dalvik VM encapsulated in a process, which results in a way of controlling how taints are propagated between applications. Finally, as every Android device runs its own file-system where application data is often persistently stored in files, these systems also perform tracking at file-level, keeping security labels hooked to the corresponding files. In conclusion, mobile IFC systems maintain users privacy by acting as complete privacy protection mechanisms for mobile devices.

In Floodgate's context, the tracking granularity is also a key question that we need to answer. Since we know that finer granularity requires more computational power and deeper changes on the underlying system, we have to keep in mind the requirements of compatibility with current Android platform and acceptable performance overhead when taking decisions for both mobile and server sides.

3. How the System Enforces Data Isolation? Data isolation is an essential mechanism to complement data flow tracking and analysis, since it prevents data from being exported through

unexpected channels without previous security monitoring and sanitization. To achieve data isolation, the following approaches have been implemented:

A) Isolation Through Virtualization: On a virtualized environment, isolation is performed at the virtual machine level and enforced by both the hardware and the hypervisor. The CLAMP system [27] provides a virtualized environment to applications built under the popular Linux, Apache, MySQL, PHP/Perl (LAMP) stack, allowing developers to continue using the operating systems, platforms and languages they are used to. CLAMP uses the Xen hypervisor to isolate each user's data on a virtual dedicated web server instance, only accessible through authentication on a developed module. The database queries performed by the user are restricted a trusted Query Restrictor that guarantees that only data owned by the user assigned to the querying web server can be accessed.

B) Isolation Through Programming Languages and Libraries: In opposition to isolation achieved at the VM-level, recent implementations of IFC systems allow developers to explore the taint-tracking facilities present in languages like Ruby and Perl to enforce data isolation in a best-effort fashion. These approaches have the advantages of isolating data just by interacting with already defined language libraries, thereby avoiding the need to adapt the runtime environment to accommodate additional isolation features. SAFE WEB [6], for example, uses Ruby dynamic features, like safe levels. Safe levels can restrict the execution mode of Ruby code on different kinds of isolated environments. Relying on this, SAFE WEB achieves label propagation and data flow enforcement and is primarily suitable for detection and protection against unintentional software bugs, which is also our primary goal.

C) Isolation at *Middleware* Level: Designed for multi-tier web applications in which data has to flow between multiple sub-systems, called domains, event-based IFC systems define messages between these domains as events. DEFCon [28] is an example of an event-based IFC system which apply an event-level or message-level granularity in data flow isolation. In such type of systems, events between domains are often placed in an event broker by the sender domain. The event broker handles the task of delivering events only to interested domains that are allowed to read the event information, being this event broker the only way how domains are able to exchange messages.

Since Floodgate intends to protect both mobile and server sides of applications without changing the operating system, isolation at *middleware*-level, not necessarily on an event-based

model, seems to be a good decision. Also, both the solutions of isolation at language and virtualization levels lack in compatibility between both sides, which would require us to adopt two different solutions instead of one.

4. How the System Specifies Data Security Policies? A basis option to take in the designing of an IFC system is to specify how security policies are unambiguously defined. Currently used examples are policy-definition languages or *ad-hoc* definitions, specific to each system.

A) Policy-Definition Languages:

Policy-definition languages allow the specification of "can-flow-to" relations between DIFC labels, applicable in the context of every application, just by having the knowledge of principals present in the application. JFlow [29] is a policy-definition extension for Java language that allows the programmer to define allowed relations between labels under a usable programming model, making it simple to use.

B) *Ad-hoc* Policy Definition: In contrast to what happens with the usage of languages to specify security policies, an ad-hoc policy definition means that security policies are specified by the developer itself, that is responsible for writing policy checking code. RESIN [5], is an example of an *ad-hoc* policy definition system, allowing programmers to create *policy objects*, which specify assertions associated with application data to be checked when data crosses certain data flow boundaries (i.e., in an attempt to send data over the network). We believe that *ad-hoc* policy-definition mechanisms best suit Floodgate's model, since we also intend to give developers the control to write the policies, and also don't base our policies in "*can-flow-to*" relations between DIFC labels.

Over time, there emerged some real IFC systems. We now present the most representative cases, based on Floodgate's context which implicates three different types: (1) systems which operate on the mobile-side only; (2) others that do it exclusively on the server-side; and (3) the distributed approaches, which implicate data communication between two entities.

2.2 Mobile-side IFC Systems

For the mobile-side, we chose three systems which take different approaches: TaintDroid, Aurasium and FlowDroid. Their operation occur either at *runtime* (TaintDroid) or in a *static*

way (Aurasium and FlowDroid), work at different granularity and let us show examples of systems which require modification of the operating system (TaintDroid) or are compatible with out-of-the-box Android devices. This way, we provide an overview of how different systems approach the key decisions that we have to make on Floodgate's mobile component.

2.2.1 TaintDroid

Presented in 2010, TaintDroid [1] is an extension to the Android mobile operating system capable of tracking the flow of sensitive data across downloaded third-party applications. TaintDroid has two main goals: one is to detect when sensitive data, such as the user's location or information about the device, leaves the system through third-party applications that might send it to remote locations on the network. The other one is to facilitate application security analysis by device users or external security auditors. With TaintDroid, a user can easily verify whether an application performs active intents to leak its private data.

Given that most applications are closed source, TaintDroid uses *dynamic taint analysis* to monitor privacy sensitive data on mobile devices. TaintDroid knows *a priori* the sets of taint sources and sinks present in the device. Firstly, information coming from a *taint source* (source of sensitive information, like the GPS sensor) is identified and a *taint marking* indicating the information type (location-sensitive, contact-sensitive) is assigned. Then, dynamic taint analysis perform an instruction-level tracking on how labelled data interacts with other data over the application information flow in a way that might cause the leakage of the original data. Finally, all the impacted data is identified when an intent of leaving the system through a *taint sink* (normally the network interface) is detected.

Taking advantage of virtual-machine-based smartphones architectural characteristics, TaintDroid provides an efficient, system-wide taint tracking mechanism, illustrated in Figure 2.3. In first place, the VM interpreter is instrumented to provide *variable-level* tracking within the application code. Then, *message-level* tracking between applications is used, since applications are allowed to share data between themselves through Binder, i.e., the Android component responsible for IPC. Third, for trusted system-native libraries, TaintDroid uses *method-level* tracking, running code without instrumentation and applying the taint propagation on the method return. Finally, *file-level* tracking is used to ensure that information stored persistently also keeps the assigned taint markings.

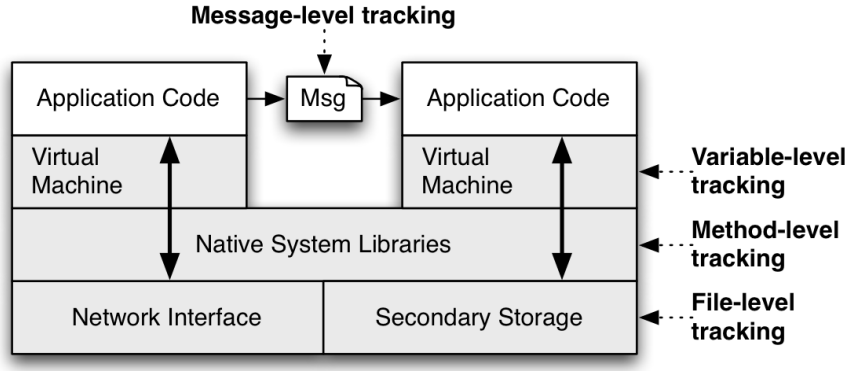

Figure 2.3: TaintDroid tracking system overview, from TaintDroid [1]

To ensure that applications are not capable of maintain sensitive untracked data flowing in the system, TaintDroid assumes that the TCB is comprised only of the virtual machine (not the code) executed in user-space where applications operate in and the operating system, or more specifically any native system libraries loaded by the untrusted interpreted application. This way, TaintDroid modified the original Android platform to only allow applications to escape the virtual machine through native libraries that are, therefore, trusted.

Against the stock Android platform, TaintDroid revealed a 3% performance overhead in starting an application, 5.5% and 18% increased time for address book create and read operations, respectively, 10% overhead in making a phone call and 29% overhead on taking a picture with the device camera. Also, an IPC benchmark revealed that for a set of 10.000 messages exchanged between local applications, TaintDroid is 27% slower and uses 3.5% more memory.

In conclusion, TaintDroid presents an efficient approach to track information flows on Android systems. The dynamic taint analysis method offers effectiveness on tracking how information coming from taint sources are used throughout the application, relieving the need to trust the application itself. However, TaintDroid is not effective in preventing the exfiltration of sensitive data, since it only builds a report of situations when it occurs, instead of blocking network access in such cases, which is included in Floodgate's goals. In addition, it requires the modification of the underlying Android mobile operating system, which we do not want in Floodgate, in order to facilitate the system's adoption.

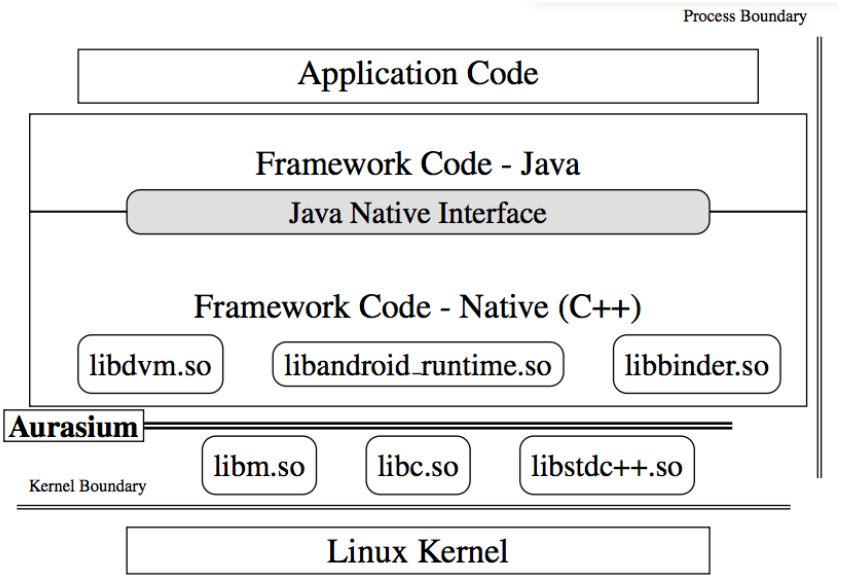

Figure 2.4: Aurasium action context, from Aurasium [2]

2.2.2 Aurasium

The second mobile IFC system we present is Aurasium [2], which may be described as an application-hardening service, presented in 2012. It is intended to turn untrusted Android applications, downloaded from third-party providers, into secure applications enforcing privacy policies regarding the user's sensitive data. In order to achieve that, the system relies on a repackaging process of the application package, and the output is a hardened version of the same inputted application to be directly installed on a device.

In practice, Aurasium takes advantage of Android's application design of mixed Java and native code execution to introduce *libc* interposition code to the target application, wrapping around the Dalvik VM under which the application's Java code runs. This way, Aurasium is able to intercept almost all interactions between the application and the operating system, since most of them trigger the execution of native methods, which is often transparent to developers, due to the available APIs. Also, code modification is used in order to place the interposition code whenever the application starts. Figure 2.4 illustrates the space where the system acts in the Android OS context.

In Aurasium, the security policies are enforced, for example, whenever an application tries to communicate with a remote endpoint through the network. In such case, Aurasium first checks it against an well-known IP blacklist, frequently updated. Also, whenever an application

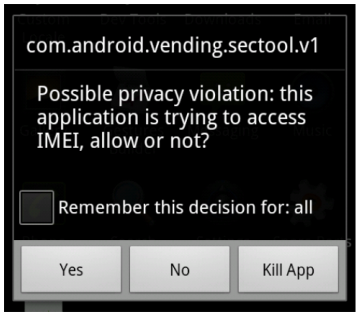

Figure 2.5: Aurasium policy example, from Aurasium [?]

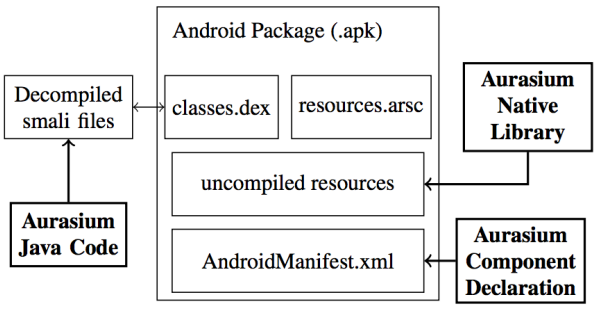

Figure 2.6: Aurasium repackaging process, from Aurasium [2]

attempts to access the device's IMEI, as illustrated in Figure 2.5, a policy check is performed to allow or disallow the access. These policies are included in the self-contained final application package and may save its state and check it in future occasions. Aurasium's interposition code shows a dialog to the user, asking whether permission to perform an action which might violate the security policies should be granted.

In order to produce the hardened version of an application to be run in the same devices as its non-secure version, Aurasium first needs to decompile the application's compiled bytecode, which is stored in a single file called `classes.dex` inside the application's `APK` package. After that, the code is inserted in normal Java classes and the application entry point is set to the Aurasium `Application` class, which guarantees that Aurasium becomes the application's entry point and its sandbox is established before the application can perform any other action. Figure 2.6 shows the components of a common `apk` package and the Aurasium components introduced during the repackaging process.

In terms of performance, Aurasium shows worse results when applications perform a large number of requests to the underlying Android APIs, with an overhead of 14% to 35% on three use-cases, specifically developed to perform a lot of API invocations. In terms of size of the hardened application package, Aurasium showed a constant increasing of about 52 Kb, which the authors considered a minor overhead.

In conclusion, Aurasium aims to achieve practical security enforcement and contributes with its novel repackaging process and a set of policies regarding sensitive mobile information. This system shares with Floodgate the goal of not requiring changes on the mobile operating system in change for protection, avoiding problems regarding the system's adoption and cross-version

compatibility. However, it can not achieve tracking of a given resource from when it is accessed to when it is exported through any of the defined sinks. In Aurasium, these two events are evaluated as independent ones. For example, if an application tries to export the device's IMEI to a remote endpoint, Aurasium presents two dialogs: (1) asking the user whether he wants to let the application read the IMEI or not; and (2) whether he wants to let the application connect to the remote endpoint, with no information of the content being sent. This way, we conclude that Aurasium does not perform real tracking of information flow, as we intend in Floodgate.

2.2.3 FlowDroid

The third mobile IFC system we present is FlowDroid [3]. FlowDroid was presented in 2013, and focuses on the static taint analysis approach to secure in-house developed Android applications as well as performing a triage of Android *malware*. In practice, FlowDroid is a system capable of analyzing applications' bytecode while covering multiple granularities, such as context, flow, field and object-level. Also, it is specifically modeled to fit the lifecycle of Android applications, their *callback* handling and user-defined widgets within the applications.

Android applications may be characterized by having multiple entry-points, instead of a single `main` method from which a call graph may be deducted, as in legacy Java programs. In order to precisely model the Android lifecycle, FlowDroid analyzes the application package and builds a *dummy* `main` method which considers all possible simultaneous execution of components. These components may comprise activites (application screens), services (perform *background* tasks), content providers (provide storage functionality) or broadcast receivers (listen to events). Since these components may be registered in XML configuration files or in Java code, FlowDroid first performs an analysis of both these components, to properly be aware of every entity within the application, as illustrated on Figure 2.7.

In order to precisely perform taint analysis during the application flow, FlowDroid combines a forward taint analysis with an on-demand backward-alias analysis, both illustrated in Figure 2.8, to detect whether some object must be tainted. It features a pre-compiled list of well-known taint sources and sinks to infer where the taint tracking process should start and end. In step 1, variable `w` is tainted, since it is coming from a taint source, and when its value is assigned to `x.f`, FlowDroid applies forward propagation to its taint. In step 2, the taint propagation for both `x.f` and `w` variables continues. Also, step 3 shows that whenever forward propagation

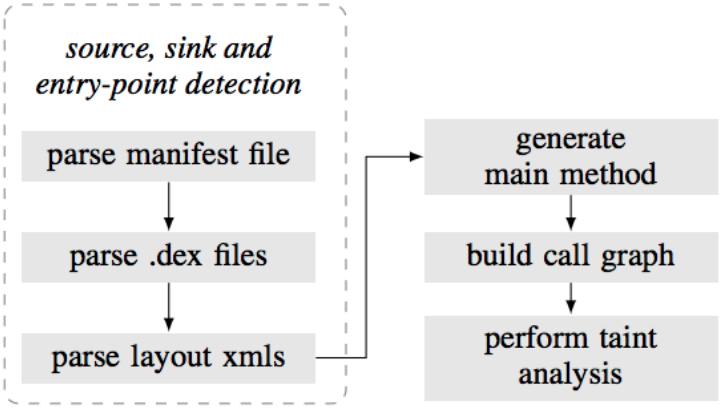

Figure 2.7: FlowDroid overview, from FlowDroid [3]

occurs, the backward-alias mechanism also searches for aliases of the respective object (`x` in this case). These concepts continue being applied during steps 4,5 and 6. Finally, on step 7 the taint analysis mechanism will detect the program is trying to export a variable which is an alias of a previously tainted one (`b` is an alias of `a.g`, which becomes tainted after the `foo(a)` call). After repeating this process for the whole application, FlowDroid reports all discovered flows from sources to sinks, include full path information on each situation.

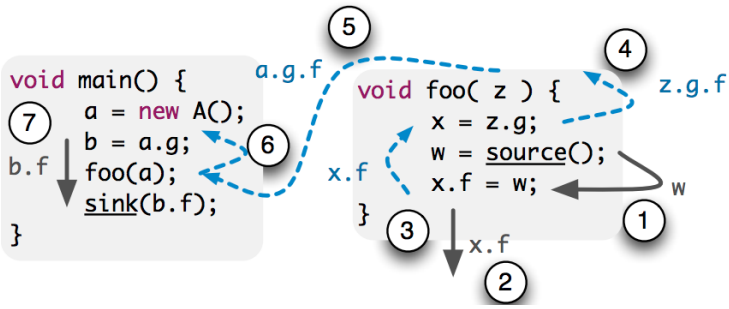

Figure 2.8: FlowDroid taint propagation, from FlowDroid [3]

FlowDroid's evaluation was made by measuring the system's precision and recall on detecting application leaks and avoiding false negative and false positive situations. FlowDroid showed a precision of about 86%, 93% recall and 0.89 F-measure, which must be considered as good results when comparing to the ones achieved by other similar platforms as AppScan[1] and Fortify[2].

[1]http://www-01.ibm.com/ software/de/rational/appscan/
[2]http://www8.hp.com/us/en/software-solutions/enterprise-security.html?#.UW6CVKuAtfQ

2.3 Server-side IFC Systems

Regarding the IFC systems operating exclusively at the server-side, we present *Servlet* Information Flow (SIF) and RESIN [5], two of the most representative, since their server architecture is indeed suchlike the one we propose in Floodgate.

2.3.1 Servlet Information Flow (SIF)

SIF is a framework for developing web applications and was presented in 2007. It targets applications deployed using conventional Linux distributions and containers like Apache Tomcat, requiring some configuration information to be included in the deployment's descriptor (web.xml file). SIF allows web applications to be developed with respect for explicit confidentiality and integrity security policies, ensuring they satisfy users' security requirements. SIF's goal is to perform the tracking of information flows within web applications and information sent to and received from the clients, exchanging the trust that must be placed in the applications for trust in the framework itself. In SIF, it is assumed that web application clients are potentially malicious and that web application implementations are not explicitly malicious but might contain bugs. These security concerns prevent that design or implementation vulnerabilities lead to sensitive data leaks that may rise financial, legal or ethical implications. SIF is designed to support only web applications written in Jif 3.0, which extends the original Jif [13] security-typed language with integrity annotations over data types.

Like its originator Java Servlet framework, SIF architecture is based on a conventional *servlet* container architecture for web applications deployment. SIF allows web applications developers to define handlers to web requests, which intercept the requests and perform label checks even before the request is mapped into the desired action. Figure 2.9 provides a step-by-step overview of how these handlers work when receiving incoming requests from web clients..

1. **Step 1** Web applications must extend the Servlet class and client establish *sessions* with the *servlet* that are tracked by the container. Data inputted by the client on the web application generally generates a request (i.e., submitting a form). SIF ensures that any data coming from a client during a session is annotated as being influenced by the client that owns the session, comprising sensitive information that must be protected according to the defined security policies.

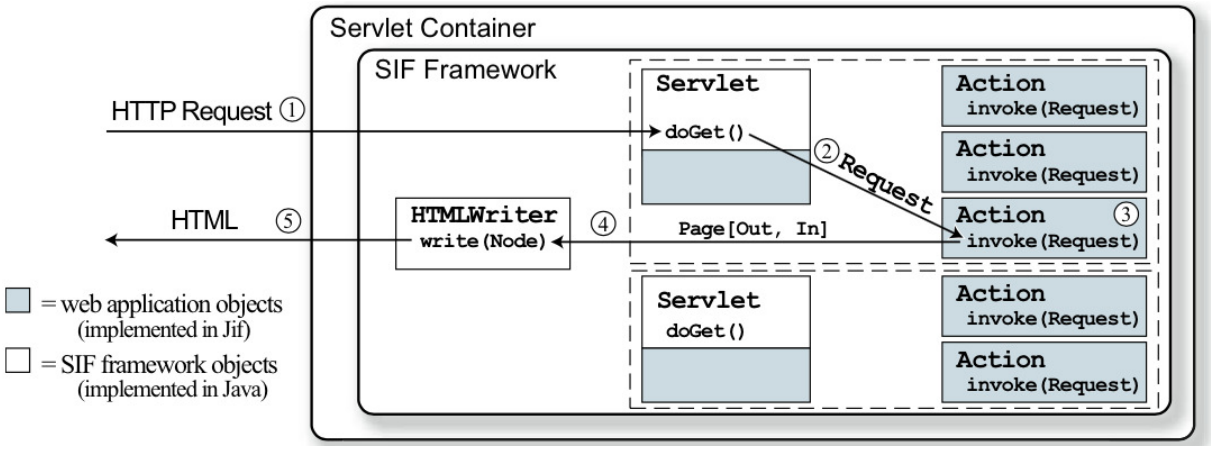

Figure 2.9: Overview of SIF requests handling, from SIF [4]

2. **Step 2** SIF provides a Request class that acts as a wrapper class for an HTTP request. Data contained in a Request is only accessed by principals which the annotation assigned to data in previous step explicitly allows to.

3. **Step 3** Web applications implement functionality through actions that are sub-classes of the SIF Action class. Actions are targets for forms or hyperlinks in the web application and are responsible for receiving and processing the submitted information. When the *servlet* receives an HTTP request, this request usually carries the identifier of the appropriate Action to use. Once the appropriate action object is found, its invoke method is called with a Request object as an argument. The *framework* ensures that tainted data (or low-integrity) coming from requests cannot be used through the application code as if it were untainted or high-integrity.

4. **Step 4** An object of the SIF Page class represents a well-formed HTML page, with elements being instances of the Node class and having associated security labels. By requiring the application to produce Page objects instead of byte sequences, SIF can ensure that each input field on a page has an appropriate security policy associated with it, and that the web application serves only well-formed HTML that does not contain possibly malicious JavaScript.

5. **Step 5** Finally, SIF converts the Page into HTML that may contain other forms or hyperlinks with associated action identifiers, and returns it to the client.

In the end, SIF offers a framework to develop Jif 3.0 web applications, with respect to

security policies defined by the application developers. These security policies are expressed in terms of DIFC concepts like principals and labels. SIF trusts the Jif compiler to dynamically track taint propagation across the web application, making sure that no sensitive information belonging to some user can be used in a way that defined policies do not allow. By using Jif language and compiler, SIF is automatically immune to implicit flows or storage covert channels, two well-known IFC threats, and also allows developers to use a programming model based on Java language, not requiring them to learn a new programming language. In terms of evaluation, SIF reported an overhead of about 25% in the server throughput [4].

Despite SIF's effective taint tracking, it has several limitations considering the goals of our work. In first place, and like most language-based IFC systems, Jif's applications have its security policies specified alongside the code. That means that every variable and method along the code is annotated with Jif labels, which makes it hard for the programmer to modify security requirements on the application, requiring the whole code auditing even for minor changes. Second, SIF's functionality is completely web-page-driven. It is supposed to rely on a cycle where the framework receives HTTP GET/POST requests from a client, processes the incoming data from those requests, and finally, builds an well-formed HTML page using Java classes that represent existing HTML nodes (i.e., Body, Div, Form, or the Page itself) to be returned to the client. In a mobile-server architecture as the one Floodgate implements, the method of returning an HTML page is not the one that suits best, since the mobile side of the system already has its own methods to build interfaces to present information to the user.

2.3.2 RESIN

Presented in 2009, RESIN [5] is a language-level IFC system which operates at a language runtime, like PHP or Python, and allows programmers to correct their program flow explicitly using data flow assertions. An example assertion may be "only a user can read or write its own password". RESIN's goals is to prevent security vulnerabilities like SQL injection, XSS [16], server-side *script* injection or password disclosures in web applications. Thus, just like in Floodgate, application code is not assumed to be explicitly malicious but may contain bugs which cause the referred vulnerabilities. Another assumption is that the language runtime it operates over is not compromised and is also part of the TCB. RESIN addresses three main technical challenges to solve:

(1) Knowing when to verify a data flow assertion. There are many different ways that an adversary might violate the password assertion described above. The adversary might trick the application into emailing the password; might use a SQL injection attack to access the passwords database; or might fetch the password from a file located in the server through a directory traversal attack.

(2) Design a generic mechanism that makes it easy to express data flow assertions, including common assertions like XSS avoidance, as well as other application-specific assertions.

(3) Make data flow assertions coexist with each other and with the application code. A single application may have many different data flow assertions and it must be easy to add an additional assertion if a new data flow rule arises, without changing existing assertions. Also, an application may be designed by multiple programmers which could not be aware of all the defined assertions and RESIN should be able to enforce them all independently.

To address these challenges, RESIN bases itself on three key ideas: (1) policy objects, written in the same language as the application, which encapsulate assertions' functionality specific to data it refers to; (2) data tracking, performed by the RESIN *runtime* which tracks the policy objects as the data propagates through the application; (3) filter objects, which act as data flow boundaries when data is about to leave the control of RESIN, such as being sent over the network. In such cases, they check data flow assertions with the assistance from the data's policy objects. Figure 2.10 illustrates these components and their interaction with a simple example of an e-mail client application. Also, RESIN features a mechanism of persistent storage of security policies along with data they are assigned to. This way, filter objects are also placed on the applications database connectors and ensure that when, for example, a password object is persistently stored in an SQL database, its associated policy object is also stored along with the data, as represented on Figure 2.11.

RESIN performance evaluation comprised the measurement of its server throughput in serving client requests and also on the application of *microbenchmarks* to test CPU-intensive operations. When comparing to a non-version of the same web server, RESIN served 25% less requests per second. On the *benchmarking* tests, RESIN showed an overhead of about 47% on tests with String operations, 71% when data was assigned with more than one policy and also a performance overhead of about 4-5 times on operations involving read and write operations on SQL databases. This way, we consider that Floodgate can outperform RESIN's performance

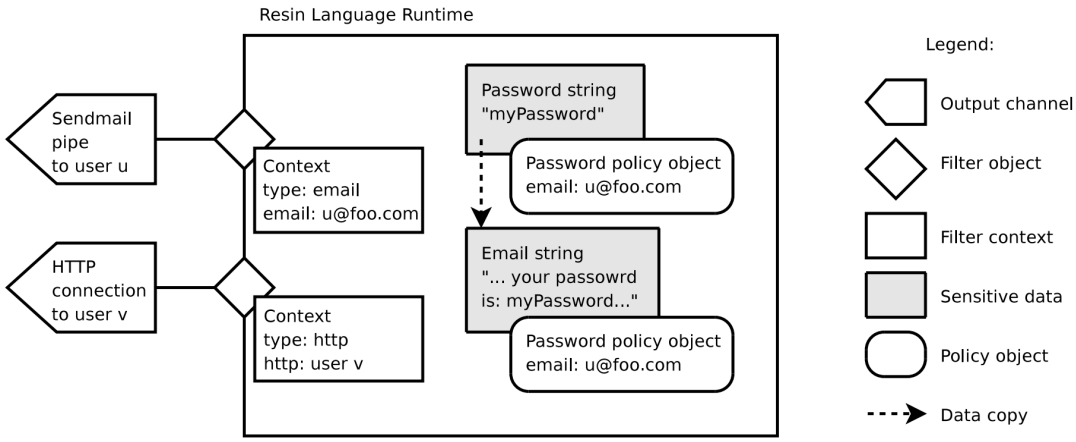

Figure 2.10: RESIN data flow assertions operation, from RESIN [5]

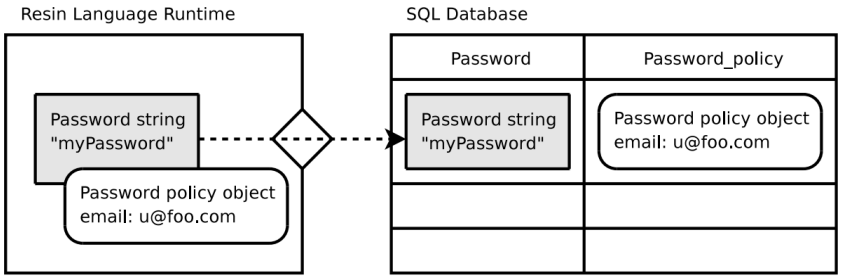

Figure 2.11: Example of data persistent storage in RESIN, from RESIN [5]

results when submitted to similar tests. Also, RESIN's primary target is to protect users and application providers against attacks like XSS and SQL injection, which are not part of Floodgate's primary goals.

2.4 Distributed IFC Systems

In addition to operating on the mobile or on the server side only, there are IFC systems which, like Floodgate, operate in an end-to-end fashion, distributing its operation between two different endpoints, which exchange information and apply common IFC policies over the exchanged data. Next, we present the most representative ones, due to their similarity with Floodgate in terms of design and end-to-end characteristics: SAFE WEB [6], Hails [7] and πBox [8].

2.4.1 SAFE WEB

Presented in 2011, SAFE WEB [6] can be defined as a *middleware* that acts as a "safety net" for event-based web applications. It ensures protection by associating data with security labels and tracking their propagation across different granularity and domains, entities responsible for a part of the application functionality. This tracking is made in order to ensure the confidentiality and integrity of sensitive data included in the web requests. By providing these security assurances while the application is running, SAFE WEB avoids expensive security code reviews, external security consultations and allows the reuse of *middleware* for other applications running over the same hardware. SAFE WEB relies on Ruby language because of its dynamic features in order to achieve label propagation and enforcement of data flow security policies.

SAFE WEB takes two assumptions. On one hand, the application code is not explicitly malicious, so it focus on protection against unintentional vulnerabilities like software bugs. Also, stakeholders are willing to accept some performance overhead in exchange for the proposed functionality.

To achieve an event-based operation as described above, where data comes from the client's web browsers to the system's web *frontend* component, SAFE WEB's design is comprised of two clearly identifiable parts, as represented in Figure 2.12:

1. **Event processing backend:** Running at the server-side, consists in the application logic, comprised by an **Event Processing Engine** that controls the execution and assigns privileges to **Event Processing Units**, which can act as generators, filters or processors of events and exchange labelled events through an IFC-aware **Event Broker**. **Events** are created from confidential data retrieved from the **Main DB** and labelled appropriately. Each unit has declassification privileges over a certain set of labels specified in its **data flow policy** and its **IFC Jail** is responsible for preserving labels during communications with the environment. The result events are stored with appropriate labels in an **Application DB** after processing, becoming available to the web frontend. Events carry key-value attribute pairs, confidentiality and integrity labels and an optional data body. The event broker matches subscriptions with published events, using a topic-based subscription language with optional content filtering on event attributes within a topic. It filters events according to their security labels, which means that only subscribers with

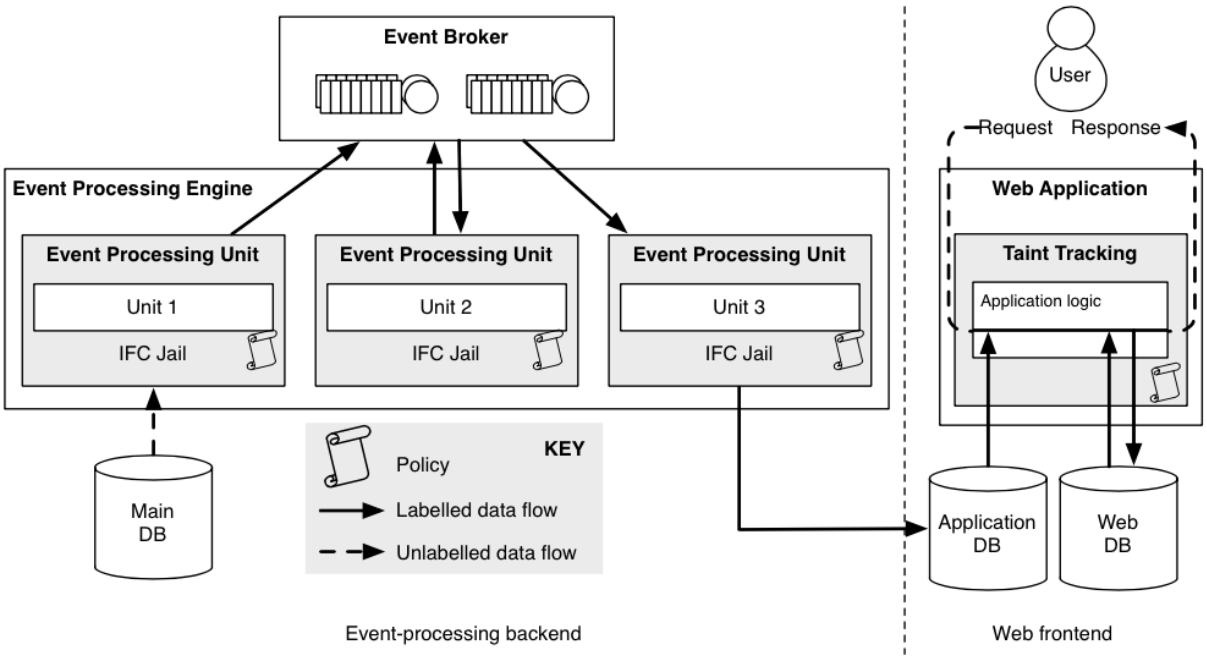

Figure 2.12: SAFE WEB middleware architecture, from SAFE WEB [6]

clearance privileges for all labels on an event can subscribe to that event.

2. **Web Frontend:** Running at the client-side, serves synchronous web requests from users by accessing the Application DB, or **Web DB** for state specific to given web session, isolating it from the rest of the application data. Labels from the Application DB are propagated in the web application by SAFE WEB's **Taint Tracking library** and checked when generating responses. As a result, security labels are associated with data throughout the processing pipeline and checked at boundaries between components with respect to the application's security policies. Data is unlabelled by the taint tracking library when it is included in a web response, to be returned to the user.

SAFE WEB's operation may be explained with the following scenario. On a healthcare service, a given unit may be responsible for compiling a daily list of patient with the correspondent reports. To achieve this, the unit must issue a subscription request to the engine, that then reads the set of labels from the unit's policy file for which the unit has declassification privileges. After that, the engine forwards the subscription request from the unit to the event broker, with all the verified labels. This way, the event broker is able to deliver every event that matches the referred labels to the unit, that will be able to process it in order to compile the lists.

SAFE WEB is estimated to add a performance overhead of about 14% in the web frontend. For the backend, an overhead of about 15% was estimated, measured by the average latency of 1000 individual events from the time they are initiated at the data producer until they are stored in the Main DB. In general, these are acceptable overheads for web applications with strong security requirements.

When comparing SAFE WEB's design and operation with the ones we target on Floodgate, they both share the same end-to-end fashion, goals and tracking of information which arrive from clients through requests to the *backend* in which both information and labels are tracked persistently stored for future access. SAFE WEB is primarily targeted to event-based applications in which clients subscribe to events generated by the server. Floodgate, in contrary, is more targeted at applications based on a request-response model, in which is the client who triggers the action. In some cases the SAFE WEB's model might fit in Floodgate's applications, but it lacks in flexibility regarding the application model and functionality.

2.4.2 Hails

In 2012, one year after SAFE WEB, our second representative system was presented. Hails [7] is a *framework* to develop web applications based in the Model-View-Controller (MVC) model designed to develop platforms comprised by mutually-distrustful components written by various entities. It relies on two principles to achieve protection of sensitive data. In first place, just like Floodgate, it specifies that security policies shouldn't be spread across the code. Instead, they should be defined alongside the application's data model. Second, access policies are mandatory after a given application gained access to some sensitive data, forcing the application to give up on communication capabilities in exchange for access to sensitive data. Hails can be described as a fine-grained IFC system, since it achieves protection at field-level, allowing programmers to specify different security policies for each field of an entity defined in the model component, as represented in Figure 2.13 where the *user*, *email* and *friends* fields all have protection at different granularity. When comparing to SAFE WEB's event-level taint tracking, Hails performs a more fine-grained field-level taint tracking, which might provide higher effectiveness but also increase performance overhead.

Based on the described assumptions, Hails implements an architecture described as MPVC, which means model-policy-view-controller. This architecture extends the MVC model by associ-

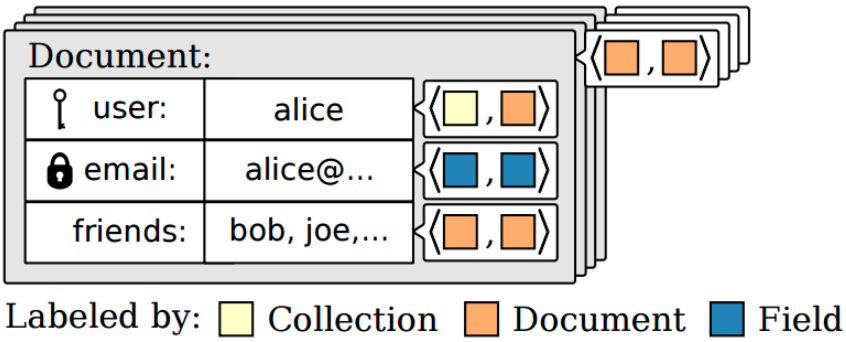

Figure 2.13: Hails field-level confinement, from Hails [7]

ating every model with a policy regarding how that type of data may accessed. In Hails, policies are specified as confidentiality and integrity labels which define principals allowed to read and write data, respectively. Then, these policies are enforced during the interactions between the architecture components. For example, if the model-policy component states that only a user's friends may access its e-mail, then views and controllers or even other model-policy components will lose the ability to communicate over the network after reading this data.

This control of network communication is made by associating to each thread a "current label" to keep track of communication restrictions, which may change during the application's flow. Threads are allowed to read and write objects if the objects' confidentiality and integrity labels, respectively, are less restrictive than the thread's current labels. Also, control over database writes is achieved the same way, checking if the thread trying to update the database is allowed to write to both the desired object and the database itself.

Since Hails is a web *framework*, the main component of every view-controller component is an HTTP server, assumed as part of the TCB. On receiving a request from an authenticated user, the HTTP server calls the necessary model-policy components to interact with persistent state and returns an HTTP response to the user's browser, containing only data the user is permitted to observe. Also, if a view-controller relies on a third-party service to perform a given task, Hails applications have access to an HTTP client that checks the invoking thread's current label against the remote principal, which may only happen if the invoking thread have not accessed data that the remote principal is not allowed to read.

To implement this, Hails applies a mix of three techniques: (1) language-level, (2) OS-level and (3) browser-level confinement. At the language-level, Hails applications are written in Haskell, a strongly-typed, memory-safe language that can identify operations involving side-

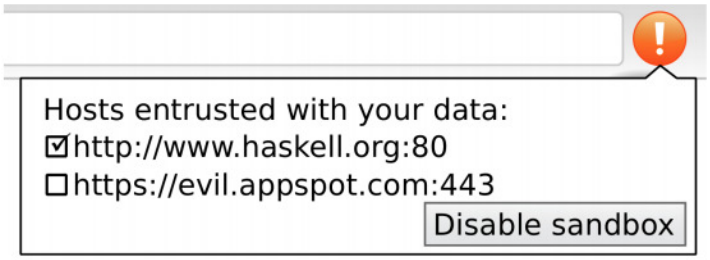

Figure 2.14: Hails browser-level confinement, from Hails [7]

effects (i.e., network/database access) and block them. Also, the OS-level confinement comprises legacy Linux isolation techniques such as only allowing access to the *loopback* interface to processes which accessed data which can not be exported. Finally, browser-level confinement is implemented by a browser extension that intercepts network communication and allows only hosts guaranteed to not leak info. On accessing potentially untrusted hosts, the browser extension explains the risks and requires the user to confirm the access, as showed on Figure 2.14.

To evaluate Hails, several experiments were performed, and compared to other web *frameworks* as Java in Jetty, Apache with PHP and Ruby on Sinatra *framework*. A server throughput test was made to evaluate the *framework* overhead. Results showed that Hails is capable of performing 1.7 times fewer requests/second than Jetty, but 28% and 47 times higher than Apache+PHP and Sinatra, respectively. Also, database read/write *benchmarks* were executed and showed that Hails performs much worse that all the other compared benchmarks.

When comparing Hails with what Floodgate intends to achieve, it has the disadvantage of not being targeted at native mobile applications. Instead, its protection in the client-side is performed at a browser-level, component which we do not want to include in Floodgate's design. Apart from that, all the other requirements such as simple policy specification and acceptable performance overhead re very close to what we want in our system.

2.4.3 πBox

Presented in 2013, πBox [8] is a platform for preserving privacy in web applications, getting a useful balance between users' privacy and applications' functional needs. πBox's goals are to protect users from misbehaving applications and their own lack of privacy knowledge. This platform intends to take the responsibility for enforcing privacy policies over users' data from

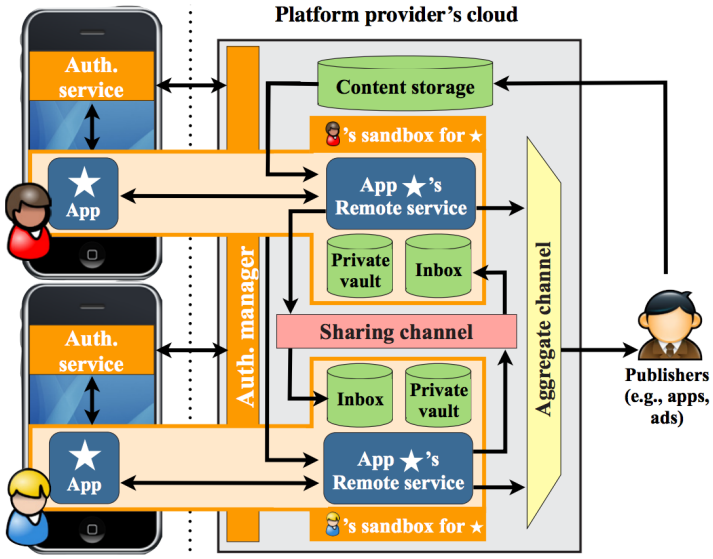

Figure 2.15: πBox architecture, from πBox [8]

the application itself (which might be malicious) or the users, which might not be able to take fine-grained privacy decisions. Instead, πBox trusts the platform to enforce the required privacy.

To achieve these goals, πBox isolates each user's instance of an application from application instances that serve other users, allowing communication only through a well-defined set of channels whose functionality is controlled by πBox. The platform's architecture relies on three different mechanisms that we describe next and illustrate on Figure 2.15:

(1) An extended *sandbox* that provides the abstraction that a slice of the *cloud* is part of the user's device. Every application running on the client-side performs its computations and *storage* within this "distributed" device which is otherwise isolated to protect user's privacy. The local half of the device can only communicate with the remote half intended for the same application and user through an authentication service running on the device. On success, the remote half opens a secure channel between the local and remote halves.

(2) Five specialized communication and storage channels which perform part of the applications *workflow* while πBox preserves the user's privacy: (2.1) A private vault which provides per-sandbox storage for a particular user and application (i.e., profile or location) and app-related content. Each and only each sandbox has read/write access to its private vault; (2.2) a content vault for per-provider storage that applications' instances need (i.e., maps, databases). Only publishers have write access to content vaults and concede read access to applications, which may access multiple vaults to fetch different data (i.e., news from news vault, *ads* from *ads*

vault); (2.3) per-app aggregate channels for publishers to collect statistics of user's collective behavior, while protecting the privacy of individual users (i.e. how many videos were watched but not who watched them). Publishers have read access and applications have write access to these aggregate channels. (2.4) per-sandbox *inbox* storage for publishers to send messages targeted at individual application users, with write access for publishers and read access for users; (2.5) per-sandbox sharing channels for sharing content with users of the same application (i.e., *newsletters*). Here, the permissions are the same as on the *inbox* storage.

(3) An adaptation and implementation of *differential privacy under continual observation* that improves the trade-off between accuracy and privacy of statistics released through the aggregate channel (i.e., ad impression counts). To support free ad-supported applications, πBox authors state that: (1) the ad network must store its ads in content storage on the πBox cloud; (2) the number of impressions (times an ad is showed) must be released through the aggregate channel; (3) the logic for selecting and fetching an ad from the content storage, based on the user's profile, and its output to the aggregate channel must be implemented inside the *sandbox*. Thus, πBox does not allow third-party services as content distribution networks from outside of the platform to communicate with its applications.

To evaluate πBox, the server throughput revealed an overhead of about 50% when comparing to a non-πBox version of the same system. This non-πBox version was comprised by an out-of-the-box Android client communicating with a Jetty-powered Java web service running on Linux. In comparison, Floodgate shows much better results, since it reveals an average overhead of 10.3% against πBox's 50% under similar conditions, with 2000 clients issuing requests for the web server.

When evaluating πBox in Floodgate's context, we find several limitations. Despite the fact that it is targeted at *backend*-supported mobile applications, it requires a modified version of the Android mobile operating system, which we avoid. Also, it acts completely in a *sandboxing* model, and Floodgate's approach reveals itself more relaxed, allowing communication with *endpoints* that are not part of the system whenever the user's privacy may not be at risk.

2.5 Summary

In this chapter, we presented Information Flow Control and its variants, providing some context with respect to the techniques we apply in Floodgate. Also, we reviewed the most representative IFC systems, which means the ones with higher similarity with our proposal. We conclude that, in spite of the multiple proposed systems to address data privacy problems on web applications, these do not really address the same challenges as Floodgate. Concretely, to the best of our knowledge, there are no systems which intend to achieve data protection using IFC techniques in an end-to-end way between native mobile applications and their corresponding *backend*. Most of the studied systems offer data protection on either mobile or server sides only, and there are also some of them which operate in an end-to-end way, but more targeted at web applications, whose clients are browsers instead of mobile devices (i.e. *smartphones*, *tablets*). In the next chapter, we present the details of Floodgate's architecture and how its multiple components integrate to achieve end-to-end privacy protection.

3 Architecture

In this chapter we describe the architecture of Floodgate, our solution, in detail. In first place, on Section 3.1, we present a brief overview of our system. After that, we present the programming and deployment model which developers must follow in order to produce Floodgate-ready applications, on Section 3.2. Then, we provide a detailed explanation of how we achieve the end-to-end taint tracking within the whole system, on Section 3.3. Finally, in Sections 3.4 and 3.5, we present and justify the building blocks in which our mobile-server architecture is based, talking about the IFC models implemented on each end.

3.1 Floodgate Overview

Floodgate is a system that relies on a mobile-server architecture to enforce end-to-end security policies. Figure 3.1 exemplifies how Floodgate controls the access to a sensitive resource, in this case the device's IMEI. Floodgate works at *middleware-level*, enforcing security policies in an end-to-end way between mobile-side applications and the server-side *backend* supporting those applications. Security policies are specified by the application developer and accepted by the user upon the installation of a mobile application supported by our system. In Floodgate, policies are specified on a privacy manifest (PM). The privacy manifest defines two privacy parameters: (1) a set of resources and values which define the privacy of the application resources and (2) a set of *trusted endpoints* which comprise the application's *backend* or other *endpoints* it trusts. Each of the defined resources must be assigned with one of these privacy keys: (1) *public*, if it can be sent over the network with no restrictions; (2) *private* if it can only be shared with *endpoints* that are part of the trusted endpoints section of the privacy manifest or (3) *protected* if the resource must not be exported outside of the mobile device in any condition. On one side, the mobile application running in a Dalvik VM features a *mobile monitor* component responsible for performing an application-level tracking of sensitive data during the application flow, adding policy checks that prevent the undesired exportation of private data, accordingly with specified

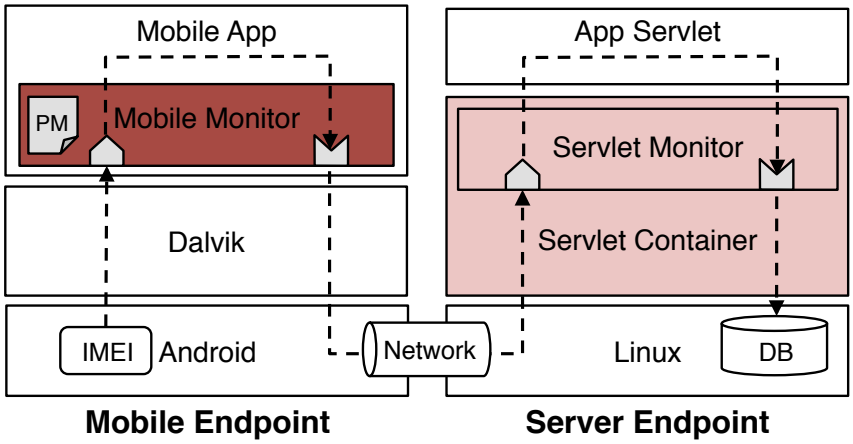

Figure 3.1: Floodgate proposed architecture

security policies. To address the trade-off between resource efficiency and tracking granularity while preventing modifications to the OS, we opted by implementing an application-level tracking model. This means that, the mobile monitor tracks the access to sensitive resources and updates a global application taint whenever on of these resources is accessed by the application code. By application-level tracking we mean that we follow a more coarse-grained tainting approach, applying taints to the whole application instead of data variables present in the application's data flow. When the application is marked as tainted and data is sent to remote *endpoints* through the taint sinks (e.g. network interfaces), this data has to keep the application's taint marks and security policies assigned, in order to achieve the enforcement of those policies on the application server-side. The server-side operates inside a *servlet container* which offers all the tools and libraries to develop and deploy a the server-side applications. Also, it features a *servlet monitor* responsible for the policies' enforcement and which provides data declassification mechanisms. On the server-side, since we don't have the same constrains regarding resources consumption and compatibility with mobile platforms, we can alter the execution environment in order to implement taint-tracking with higher granularity.

To implement taint tracking we rely on two different techniques, each one applied in one of the mobile- and server-side components (represented in Figure 3.1). On the mobile-side, we use aspect-oriented programming (AOP) to efficiently intercept accesses to sensitive resources, check them against the defined security policies, perform our application-level tainting and propagate tainted data to the *backend*. On the server-side, we use dynamic taint tracking through code instrumentation in order to continue the taint propagation and enforcement.

3.2 Programming Model

Floodgate offers a programming and deployment model specifically targeted to distributed mobile applications. By this term, we mean that applications comprise two components: one client running in the mobile endpoint, and a service running on the server endpoint. Both components communicate in an RPC fashion.

On the application client, the programming model does not suffer any major modifications when comparing with the traditional mobile applications programming model. For instance, developers need to add the Floodgate mobile library as a dependency for the mobile application. Also, they need to define the privacy manifest as an application's resource. After that, developers do not need to write specific code for taint tracking issues, for example, to label variables, or to define sinks or sources. Developers must use the network methods offered by Floodgate's mobile library to communicate with remote *endpoints*, since they are a key part on the end-to-end taint tracking process. Also, the deployment process for a Floodgate mobile application does not suffer any modifications when comparing to the current model of packaging, signing and publishing the application in application stores.

To implement the application service, Floodgate's programming model is essentially based on the implementation of a *servlet* and the definition of a REST API to be deployed on the secure container and communicate with the application client. This API must offers methods representing CRUD operations to be consumed by the mobile application, which performs HTTP requests (GET, POST, PUT, DELETE) and receives the corresponding responses, with data to be presented at the mobile *endpoint*. Floodgate's *backend* offers an interface to easily define the available API, and the behavior to produce when each API method is called from the mobile *endpoint*. Also, Floodgate offers methods for easily persist inputted data, according to a previously defined database model. Apart from the API-definition interface, the *backend* provides configuration files where developers must define the application data model (e.g. the entities it will handle), the database model and also configuration parameters regarding the *servlet* itself (e.g. the database credentials, ports to be deployed to, security certificates, etc.).

3.3 End-to-end Taint Tracking

In this section, we detail the end-to-end taint tracking process in Floodgate's applications, from the time when the sensitive data is accessed by the application client at the mobile device, to when it arrives at the application service, and on subsequent accesses from there.

Step 1: When a mobile application running on the application client accesses a sensitive data source (e.g. calling the API native method to retrieve the device's phone number), Floodgate's mobile monitor intercepts this method's execution. On this interception, Floodgate verifies which of the sensitive methods is executing and then maps this method to its corresponding privacy manifest resource. This way, Floodgate is able to check the privacy of the accessed data (*public*, *private* or *protected*). If the accessed resource is marked as private or protected for the application in question, then Floodgate updates the application's global taint accordingly, initializing it if necessary. In the end, the application taint contains the information of all the private resources accessed by the application.

Step 2: When the application client calls a network method, this method's execution is also intercepted by Floodgate's *aspects*. During this interception, Floodgate firstly verifies the application's taint. If it is not empty, meaning that a sensitive resource was accessed, our system will then verify the destination endpoint for the produced request. This verification includes checking the endpoint's address against the trusted endpoints list present in the application's privacy manifest. If the destination endpoint is not part of this list or if the resource is marked as protected, then the request will be blocked. Otherwise, if the application is not tainted, the request will proceed with no further checks or restrictions.

Step 3: If the permission to proceed is granted to the request, the mobile monitor will include the application taint as part of the request, in the form of a header with the name *"privacy"*. This is the way how our system guarantees that the taint tracking is actually performed in an end-to-end fashion. After the HTTP request is composed and the privacy header is appended, the request will be finally sent.

Step 4: On the server-side, Floodgate receives the incoming request, taint data if needed, and enforces the taint tracking during the execution flow of the application service. To achieve this, our system comprises an incoming filter for HTTP requests, which examines each request's headers in search for the *"privacy"* one. If the *privacy* header is not empty, meaning that the

request contains potentially sensitive data, Floodgate will use its server-side tainting API to assign the *privacy* header taint to the request body.

Step 5: The request triggers a REST API method on the server, using the now tainted request body in the method's behavior. Since Floodgate's server side features dynamic taint tracking, whenever any piece of tainted data interacts with other non-tainted data in the server, Floodgate takes care of taint propagation, in order to guarantee the enforcement of the privacy policy on the server-side.

Step 6: Also, Floodgate features outgoing filters for HTTP requests, which work the same way as the incoming filters but in the opposite direction. So, whenever the *backend* performs an HTTP request, either to respond to the mobile application request or to communicate with other *servlets* (in the same container or third-party ones), the filters also take care of checking taints assigned to data. It is the programmers who must define the security policies regarding situations when applications try to export tainted data, based on multiple available factors such as the data taint or the destination endpoint.

Apart from taint tracking and enforcement, Floodgate also features taint persistence on the server-side. We achieve this by extending the database model referred in Section 3.2, in which we include a table for persistent taints. Also, we extend the programming model in a way such that when programmers persist some data, accordingly to the database model, Floodgate automatically persists its taint. Also, when developers query the database model, the returned data is automatically assigned with the corresponding taints, if existing. This allows security policies enforcement even when data is not accessed during long periods of time, and thus not kept in memory. Next, we describe Floodgate's mobile- and server-side building blocks in more detail.

3.4 Mobile-side Blocks

On the mobile-side, Floodgate extends common mobile applications with two components, in order to achieve the application-level taint tracking: A text file *privacy manifest* that is produced by the developer and included in the application's resources; and the mobile monitor, which takes the form of a library archive, to be specified as a target application's dependency. Both components and the way they interact with the mobile application are represented in

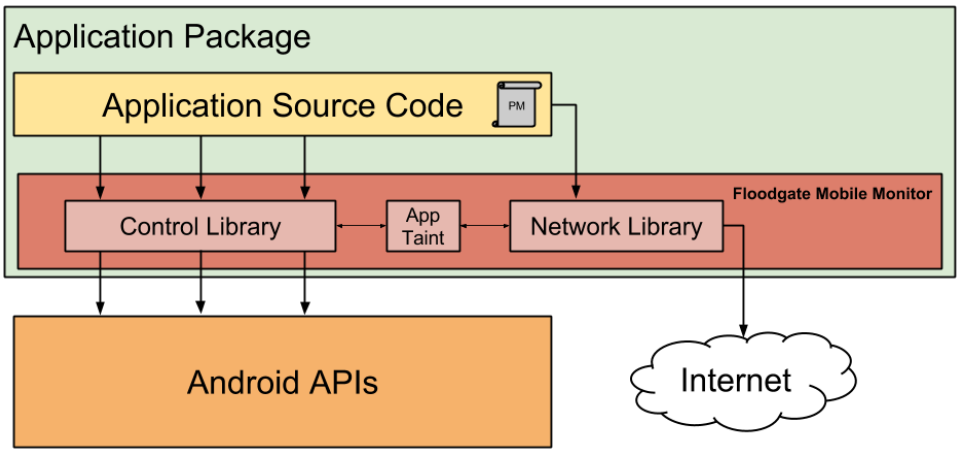

Figure 3.2: Floodgate mobile component design

Figure 3.2.

The privacy manifest (PM in Figs. 3.1 and 3.2) is a data structure which specifies multiple sensitive mobile resources (the device's IMEI, GPS/GSM location, phone number, etc.) along with a key defining the privacy of each resource (public/private). Also, it specifies a list of trusted network *endpoints* with whom the application is allowed to share private resources with. In Floodgate, a public resource is allowed to be sent through the network to any *endpoint*, while a private resource can only be sent to *endpoints* listed as trusted. A simple example of a permissions manifest file is represented in Figure 3.1.

```
<?xml version="1.0"?>
<permissions>
   <!-- Telephony permissions -->
   <permission>
      <id>IMEI</id>
      <access>public</access>
   </permission>
   <!-- Trusted Endpoints -->
   <trusted_endpoint>
      <endpoint>http://172.168.1.1:8080</endpoint>
   </trusted_endpoint>
</permissions>
```

Listing 3.1: Permissions manifest example

The mobile monitor, by its turn, is comprised of two modules: The control library, which controls the access to the resources specified in the privacy manifest and their propagation, and a network library, which enables communication over the network. Since Floodgate actually performs an application-level taint tracking on the mobile-side, this library initializes a data structure containing the application taint. This taint is updated when accesses to sensitive resources, specified in the permissions manifest, are detected.

In order to detect when the sensitive resources are accessed, the control library relies on aspect-oriented programming (AOP) concepts. AOP partitions the program logic into multiple "concerns" (cohesive areas of functionality). With AOP, it's possible to define source code blocks ("cross-cutting concerns") to be executed before/after/instead another piece of code without explicitly changing it. This way, the "cross-cutting concerns" can be implemented once and injected wherever it is needed. The combination of the particular point in a program that might be the target of code injection and the code to be injected is called an *aspect*. Then, the process of injecting code is called *weaving*. Our mobile monitor features multiple concerns to be injected when the execution of methods that access a sensitive resource is detected. On such situations, the application taint is updated, always reflecting the resources accessed by the application's source code. Also, the network library provides methods to communicate with remote *endpoints* over the network. These methods ensure that the application taint is always sent along with data, in a transparent way for the developers. In order to enforce this taint propagation through the network, the monitor features some "concerns" regarding the access to other network libraries instead of ours, blocking their execution.

The method of tainting the whole application whenever a private resource is accessed, raises an issue of high false-positive rates in terms of policy violations. For example, if the developer accesses the device's IMEI and after that sends some text inputted by the user to the *backend*, the application taint will incorrectly report that the device's IMEI is being sent over the network. To leverage these issues, we adopt a strategy of taint cleaning based on some heuristics. An example of these heuristics is the resource lifetime. The user's location may become inaccurate after a short period of time, so it must be sent almost immediately. So we define thresholds for the location tags to be removed from the application taint.

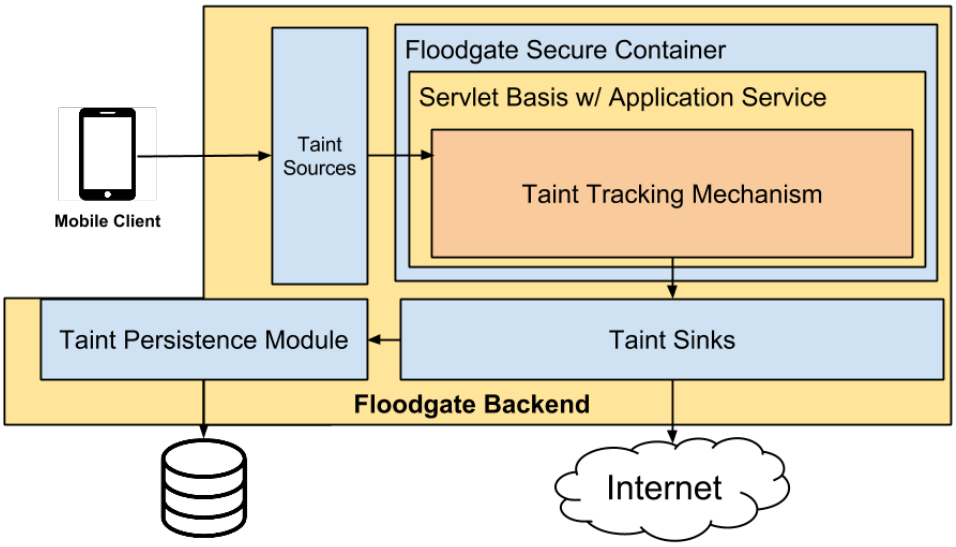

Figure 3.3: Floodgate server component design

3.5 Server-side Blocks

On the server-side, Floodgate provides a secure container where to deploy the *backend* of mobile applications. This secure container is comprised of two main components, as represented on Figure 3.3: (1) a *servlet basis* which, as the name implies, offers programmers tools and interfaces that simplify the development of the *backend* application which supports the mobile-side component; and (2) the *taint tracking and propagation mechanism*, which is responsible for receiving potentially tainted data from the mobile application and continue the enforcement of the defined security policies during the server-side application flow.

In detail, the *servlet basis* offers interfaces for programmers to focus on fast application development instead of worrying about setting up the whole environment needed for a web server to run (i.e. databases and network configuration). It compiles these required settings in a single configuration file and offers programmers a project structure where they must implement the application logic (i.e. data model, interfaces). Also, it comes with multiple included tools that help in data exchange (JSON/XML parsing tools), database querying and management or even application security concerns (SSL configuration and certificates management.)

Focusing on the interfaces to communicate with the mobile component of the application, the *servlet* basis allows the definition of APIs on the application service in an URL-driven way, as represented in Listing 3.2. By analysing the code, we can identify a class that implements a

servlet named `ItemResource`. This *servlet* provides a set of methods through its API (in this case just howing method `getAllItems`), define a set of resource methods to deal with `Item` objects, as long as this data type is defined on the application model. The `@Path` annotation means that every request sent to the specified path, in this case a relative one, will be handled by this class. After that, the annotations `GET` or `POST` specify that the triggered method depends not only on the destination URL included in the request but also on the HTTP request type issued. In the figure's case, an HTTP GET request to the specified URL will trigger the `getAllItems()` method.

```
@Path("/items")
public class ItemResource {
   @GET
   public List<Item> getAllItems() {
     return findAllItems();
   }
```

Listing 3.2: Jersey syntax example

As referred in Section 3.4, the Floodgate network library deals with the propagation of application taints to the server-side. In order to continue the enforcement of privacy policies on the server-side, we apply concepts of static analysis, code instrumentation and dynamic taint tracking.

Every *servlet* running on the Floodgate's secure container has to pass through an *a priori* process of static analysis. This process consists in identifying the points in the application where tainted data enters the system, named taint sources (e.g. interfaces to receive network requests coming from the respective mobile application), where it is passed (e.g. variables assignment, methods return) and where it may leave the system (e.g. output network interfaces, database connectors), which we refer to as taint sinks.

After the analysis process, Floodgate instruments the application, applying code injection in order to propagate taints wherever the static analysis process dictates to. This instrumentation process follows a set of rules to define the points in program where injection is needed, by the way of adding lines of code that perform the taint propagation. Every variable is annotated with a *tag*, which can be another variable with the corresponding *tag* name or it can be a field belonging to the variable itself if possible (i.e. a data structure). Listing 3.3 provide a simple

example of how Floodgate achieves taint propagation with code instrumentation. This example shows two variable assignments. For instance, in the first instruction (present in the original version of the program), variable `i` is declared. Since it is a primitive type variable and can not have any other fields belonging to the same variable, Floodgate achieves taint propagation by injecting the second instruction, which defines `i_tag` as the tag of variable `i`, initialized with value 0. Also, on the original version of the program, the third instruction declared and initialized an instance of `CustomObject`. Since it is a data structure which can handle local fields, Floodgate automatically defines a `tag` field inside the `CustomObject` instance to handle the variable passing.

After that, whenever occurs variables assignment, as in instructions 5 and 7, Floodgate automatically propagates the tags, dealing with the differences in the object types, as it can be checked in instructions 6 and 8. This way, every variable gets its taint tag propagated along the application flow, avoiding that tainted data flows to other not tainted data in an unnoticeable way.

```
int i = 1;
int i_tag = 0;
CustomObject obj = new CustomObject();
obj.tag = 0;

int b = i;
int b_tag = i_tag;

CustomObject xyz = obj;
xyz.tag = obj.tag;
```

Listing 3.3: Floodgate code instrumentation example

Finally, we apply dynamic taint tracking in order to keep track of when tainted data are set to leave the system, acting accordingly. Since every variable has an assigned taint, Floodgate is able to check these taints at application boundaries against the defined security policies to find whether there occur policy violations, possibly blocking the actions. Also, Floodgate features a taint persistence module, ensuring that taints entering the *backend* are persistently stored along with data they refer to, avoiding taint loss due to in-memory storage only.

The taint persistence module is comprised of a database which keeps the mapping of every taint entering the system, along with an identifier of the object to which it belongs. When tainted data enters the system and the application performs a write operation to the database, our *servlet basis* is responsible for also storing its correspondent taint in the persistence module. In the same way, whenever there occurs a read operation, the *servlet basis* is called in order to associate the taint to the concrete data, before it can be manipulated by the application or returned to the client.

3.6 Summary

In this chapter, we presented the Floodgate design, along with a detailed explanation of all the involved components present on both mobile and server sides. Also, we explained the techniques we use to achieve the desired protection of user's sensitive data when manipulated by *backend*-supported mobile applications, aspect-oriented programming and dynamic taint tracking through code instrumentation. Also, we provided a step-by-step overview of the system's operation, showing an example of end-to-end taint tracking in Floodgate applications. Next, we present the implementation details of our system.

4 Implementation

In this chapter we present the implementation details and the tools we used in order to materialize the architecture described in Section 3. On Section 4.1 we focus on the implementation of Floodgate's mobile component and leave the server component for Section 4.2.

4.1 Mobile-side Components

In this section, we outline some of the most important implementation aspects of our system: (1) the *privacy manifest* support; (2) details about both components which comprise the *mobile monitor*; and (3) an insight on the implementation of the *servlet basis* and the *taint tracking and propagation mechanism*.

4.1.1 Privacy Manifest

In order to keep programming paradigm that Android developers are already used to, we implemented our *privacy manifest* as a XML file as well. Android SDK allows developers to define multiple text files containing resources to use within an application's source code. These text files may contain pre-defined *strings*, colors, or values and are implemented using the XML text-based format. Even the `AndroidManifest`, a configuration file where core application parameters are defined, appears on the form of a XML file. This manifest is placed under the `resources/raw` folder and contains two sections: One for defining the privacy of each mobile sensitive resource; and other for defining the application's trusted endpoints. Then, when a Floodgate application starts its execution, this file is parsed and its results are stored under Java `HashMap` data structures for faster access.

A simplified example of a privacy manifest file is represented on Listing 3.1 and currently supported privacy-sensitive resources are summarized on Table 4.1. Here, we chose resources of the mobile device (e.g., IMEI, phone number) or the user itself (e.g., saved accounts, location)

#	Resource ID	Method
1.	BLUETOOTH_ADDRESS	android.bluetooth.BluetoothAdapter.getAddress()
2.	BOOKMARKS	android.provider.Browser.getAllBookmarks()
3.	COUNTRY	java.util.Locale.getCountry()
4.	GPS_LOCATION	android.location.Location.getLatitude() / getLongitude()
5.	GSM_LOCATION	android.telephony.gsm.GsmCellLocation.getCid() / getLac()
6.	IMEI	android.telephony.TelephonyManager.getDeviceId()
7.	INSTALLED_APPS	android.content.pm.PackageManager.getInstalledApplications()
8.	INSTALLED_PACKAGES	android.content.pm.PackageManager.getInstalledPackages()
9.	LAST_KNOWN_LOCATION	android.location.LocationManager.getLastKnownLocation()
10.	LINE1_NUMBER	android.telephony.TelephonyManager.getLine1Number()
11.	MAC_ADDRESS	android.net.wifi.WifiInfo.getMacAddress()
12.	SIM_SERIAL	android.telephony.TelephonyManager.getSimSerialNumber()
13.	SSID	android.net.wifi.WifiInfo.getSSID()
14.	SUBSCRIBER_ID	android.telephony.TelephonyManager.getSubscriberId()
15.	TIMEZONE	java.util.Calendar.getTimeZone()
16.	USER_ACCOUNTS	android.accounts.AccountManager.getAccounts()
17.	VISITED_URLS	android.provider.Browser.getAllVisitedUrls()

Table 4.1: Privacy-sensitive resources currently supported by Floodgate: (1) the device's bluetooth address, (2) the user's bookmarked sites on browsers, (3) the user's selected country, (4) fine location from the GPS sensor, (5) device's coarse location based on network information, (6) device IMEI, (7) the set of applications installed on the device, (8) the set of packages installed on the device, (9) the device's last known location, (10) device phone number, (11) device MAC address, (12) SIM card's serial number, (13) the current network's SSID, (14) subscriber ID, (15) current timezone, (16) user accounts saved on the device, (17) visited URLs on the device.

and each method on the table is the way how it can be accessed in the Android *framework*. In order to extend this privacy manifest to other types of sensitive data, developers only have to provide all the methods which can be used to access each resource on the control library, and define a key for each of these resources to be included in the application taint when necessary. In the privacy manifest, the `<permission>` tag is used to define a new resource privacy permission, identified by its `<id>` tag and which privacy value (public/private) is defined with the tag `<access>`. The trusted endpoints are specified with the tag `<trusted_endpoint>`, and its value is represented under the tag `<endpoint>`.

4.1.2 Control Library

In order to intercept the accesses to sensitive resources on the mobile device and provide network communication capabilities, the control library is used. It is implemented as an application library, which can be added as an Android project dependency through Gradle [1], a build automation and dependency management tool widely-used in Android development.

[1] www.http://gradle.org/

Control library can be defined as a set of Java classes, each one representing an *aspect*, annotated with the `@Aspect` annotation. Every *aspect* is implemented using AspectJ, an aspect-oriented extension to the Java programming language, whose library archive is included in the Floodgate library as a dependency. In practice, an AspectJ *aspect*, as illustrated in listing 4.1, can be defined as two separate parts which together define where in the code they operate and its actual behavior: A pointcut, represented with the `@Pointcut` annotation, which defines which method the aspect is intended to intercept and whether the method must be intercepted on every execution or only when an explicit call is made at it by the programmer. For example, the pointcut `call(* android.telephony.TelephonyManager.getDeviceId())` states that the aspect which it refers to aims at intercepting every explicit call to a method named `getDeviceId()` of the `TelephonyManager` Android native class, with any type of return value.

The other part, which defines the *aspect* actual behavior when called, is the advice, which in AspectJ, can be represented by the `@Before`,`@After` or `@Around`, whether it must be executed before, after or instead the pointcut method's code. Here was the place where we implemented the checks regarding the privacy of the permissions being accessed (by parsing the content of permissions manifest), updating the application's taint if necessary.

```
@Aspect
public class ImeiAspect {

   private static final String POINTCUT_METHOD =
      "call(* android.telephony."
      "TelephonyManager.getDeviceId())";

   @Pointcut(POINTCUT_METHOD)
   public void getIMEI() {
   }

   @Around("getIMEI()")
   public Object weaveJoinPoint(
         ProceedingJoinPoint joinPoint) throws Throwable {
      /*
       * check privacy options,
```

```
        * update taint information
        */
        return joinPoint.proceed();
    }
}
```

Listing 4.1: Skeleton of the Floodgate aspect responsible for intercepting IMEI accesses

Also, application developers must add some configurations to the application's build file, for the application to recognize the defined *aspects* during the building process. Concretely, we have to use the AspectJ compiler (`ajc`, an extension of the Java compiler, `javac`) to weave all the classes that are affected by our *aspects*[2].

4.1.3 Network Library

Floodgate's network library, responsible for providing network communication capabilities to Floodgate applications, is also a key-part of our implementation. Abstracting the OkHttp [3] library, it contains methods for developers to perform common HTTP requests, manage cookies and handle network exceptions. Also, since Floodgate's taint propagation from the mobile to the server side is achieved by the addition of a "privacy" header to each HTTP request, the network library is responsible for that addition. For example, when a developer calls the `post(...)` method to perform an HTTP POST request to the *backend*, the library transparently adds the "privacy" header, enforcing the taint propagation. Listing 4.2 illustrates the client code necessary to send the device's IMEI over the network, from which we can conclude that all the security enforcement remains transparent for the developer.

```
object = new JSONObject();
object.put("imei", TelephonyManager.getDeviceId());
resp = NetworkHelper.post(BACKEND_URL, object.toString());
```

Listing 4.2: Fragment of client code for reading the device's IMEI and sending it over to the server.

[2] http://fernandocejas.com/2014/08/03/aspect-oriented-programming-in-android/
[3] http://square.github.io/okhttp/

In order to obligate developers to use our network library (due to the taint propagation features), we also block the execution of other third-party communication libraries which do not enforce taint propagation. This is reached by extending our control library with some "aspects", blocking the access to those third-party libraries. Next, we cover the implementation details of the server-side components.

4.2 Server-side Components

To implement the Floodgate's server-side secure container, where lies most of the system's complexity and functionality, we focused on two main challenges. On the one side, the *servlet basis*, which provides a platform that implements the proposed programming model, offering a simple API to application developers. On the other side, the *taint tracking and propagation mechanism*, based on code instrumentation, which enables the enforcement of defined privacy policies.

4.2.1 Servlet Basis

For the *servlet basis*, we used the Dropwizard *framework*[4]. This Java *framework* targets simple implementation and deployment of production-ready and high performance RESTful web services, allowing developers to focus mostly on the application behavior, instead of worrying about web server configuration, application metrics, logging or operational tools. In the next paragraphs we will describe what this *framework* provides to developers.

For the HTTP server, Dropwizard uses Jetty web engine[5] to directly embed a web server into the application package. This way, Floodgate applications become self-contained programs, with no need for deployment on *servlet* containers like Apache Tomcat. All the web-server configuration parameters, like HTTP/HTTPS ports to deploy to, certificates and *keystores* to use, database credentials or logging settings are described in a configuration file which remains in the application package.

To build RESTful services, Dropwizard uses Jersey[6]. This *framework* allows developers to

[4] `www.dropwizard.io`
[5] `www.eclipse.org/jetty`
[6] `www.jersey.java.net`

gracefully map HTTP requests to Java objects, simplifying the task of inputting data into the application. Also, it provides simple mechanisms for developers to define the web application's URL *schema*, mapping the methods to be called when those URLs are reached. For example, Listing 3.2 provides a simple example of how this mapping is made in Jersey. When deployed, the referred code will execute `getAllItems()` method whenever an HTTP GET is sent to the URL `<application_url>/items`. Regarding data formats, Dropwizard uses Jackson, a library for dealing with JSON data in Java. Also, it provides object mapping, making it easy to convert JSON objects to the application's domain model. Also, Dropwizard provides libraries for taking application performance metrics (Metrics), logging (Logback and slf4j), data validation (Hibernate Validator), database interaction (JDBI), time data handling (Joda Time), among others.

Apart from the presented *framework-provided* tools, in order to enable Floodgate's taint propagation for the server-side, and backwards, we had to implement the HTTP filters represented as "taint sources" and "taint-sinks" in Figure 3.3. To achieve that, we used the filtering library provided by Jersey. Then, we created a custom annotation "`@TaintCheckRequired`", with which developers must annotate the *backend* URL-mapped methods that must be checked against taint presence. This annotation triggers a method which provides two different behaviors, considering the request's direction. For incoming requests, it checks the request headers, materially the "privacy" one. If it is not empty, then the filter taints the incoming data accordingly. For outgoing requests, it filters the taint of the data being exported. In case data is indeed tainted, the filter adds the same "privacy" header to the request, with the respective value. Although, the action of annotating data with taint tags or checking the assigned taint tags means we need dynamic taint tracking along the application flow and also a tainting API to provide us those methods. This takes us to the second part of our server-side implementation, the taint tracking and propagation mechanism.

4.2.2 Taint Tracking and Propagation Mechanism

In order to perform taint tracking and propagation on the server-side's application flow, we used Phosphor [11] instrumentation tool and its tainting API. Phosphor applies dynamic taint tracking techniques through *a priori* code instrumentation. This means that it takes as input any archive containing pre-compiled Java binaries (i.e. project folders, `.jar` archives, or

even simple `.class` files) and outputs an instrumented version of the same archive. Within the instrumentation process, Phosphor firstly finds and analyzes the inputted binaries (inputted archives may contain other types of files), and then starts the instrumentation process, applying some code modifications in order to achieve the desired taint tracking and propagation.

Phosphor tracks taint tags for primitive variables declared within the code by adding an additional variable for each primitive variable (or an array for each primitive array) to store those tags. The tag is stored in a memory location adjacent to the original primitive variable. When a given method returns a primitive value, Phosphor changes its return type to return instead a pre-allocated object containing the original return and its taint tag. Taint tags for primitive method arguments are always passed just before the tagged argument, simplifying stack shuffling prior to method invocation. Phosphor modifies almost all *bytecode* operations to be aware of these additional variables. For example, instructions that load primitive values to the operand stack are modified to also load the taint tag to the stack. Phosphor also wraps all reflection operations to propagate tags through these same semantics as well. Unlike multiple other taint tracking systems, which can only deal with Integer tags, Phosphor allows both Integer or other type objects to be used as tags, which, although, introduces some additional runtime overhead. Listings 4.3, 4.4 and 4.5 illustrate an example of modifications which Phosphor applies to store and propagate taint tags in Java code, both with Integer and arbitrary object tags. Despite the examples being shown at source code level, we must be aware that Phosphor works entirely at *bytecode*-level, requiring no access to the application's source code.

```
public int foo(int in){
   int ret = in+val;
   return ret;
}
```

Listing 4.3: Original Java code

```
public TaintedIntWithIntTag foo$$PHOSPHOR(int in_tag,int in){
   int ret = in+val;
   int ret_tag = in_tag | val_tag;
   return TaintedIntWithIntTag.valueOf(ret_tag,ret);
}
```

Listing 4.4: Phosphor intrumentation with Integer return

```
public TaintedIntWithObjTag foo$$PHOSPHOR(Taint in_tag,int in){
   int ret = in+val;
   Taint ret_tag = in_tag.combine(in);
   return TaintedIntWithObjTag.valueOf(ret_tag,ret);
}
```

Listing 4.5: Phosphor intrumentation with Object return

To propagate taint tags, Phosphor can apply one of two different techniques – integer tainting mode or object tainting mode – the one we apply in Floodgate. In Integer tainting mode, tags are 32-bit integers, and Phosphor uses bit-wise OR operations to combine them, allowing only 32 distinct tags, but faster propagation. In Object tainting mode, taint tags are arbitrary objects and also hold a structure which contains all other tags from which that tag was derived, allowing for an arbitrary number of objects and relationships. Like most taint tracking systems, Phosphor propagates taint tags through data flow operations (e.g. variables assignment, arithmetic operations, etc.), but also through control flow. This means that, even when there are no explicit interactions between the input and the output, Phosphor is still able to perform taint propagation. To achieve this, Phosphor modifies each method to pass and accept an additional parameter, representing the control flow dependencies of the program to the point of that method. Within the method execution, Phosphor tracks a stack of dependencies, with one entry for each branch condition that is currently influencing the method's execution. When a given branch no longer controls the execution (e.g. at the point where both sides of the branch merge), the resulting taint tag is popped from the control flow stack. Before any assignment, Phosphor inserts code to generate a new tag for that variable by merging the current control flow tags with any existing tags on the variable. Listing 4.6 shows an example of a method in which taint tags are not propagated through data flow, but that Phosphor's control flow propagation helps solving.

```
public String leakString(String in){
   String ret = "";
   for(int i = 0; i < in.length; i++){
      switch(in.charAt(i)){
         case 'a':
            ret+='a';
```

```
            break;
        ...
        case 'z':
            ret+='z';
            break;
    }
}
return ret;
}
```

Listing 4.6: Phosphor control flow example

Apart from the taint tracking mechanism, Phosphor offers a tainting API for assigning and reading taint tags on variables. In Floodgate's context, we use the tainting API in order to assign and verify taint tags on the *backend*'s incoming and outgoing HTTP filters, respectively. Also, we use it the same way when reading and writing persistent objects, so that we can persistently store or retrieve the corresponding taint tags, using our taint persistence module. Developers must call the `Tainter.taintedXXX(XXX input,int tag)` or `MultiTainter. taintedXXX(XXX input,Object tag)` methods, replacing `XXX` with appropriate type (e.g. `int`, `long`, `Object` etc.), for integer tainted or object tainted values, respectively. The referred methods return a tainted copy of the input value with the desired tag transparently applied. To retrieve tags, developers must call the `Tainter.getTaint(...)` and `MultiTainter.getTaint(...)` functions. Phosphor wraps its Object tags in instances of its Taint class, which contains the variable's tag and its list of all its Taint type dependencies. Since Phosphor applies to pre-compiled *bytecode*, developers must write calls to these methods when writing their applications' code, then compile it, instrument it, and finally run it. in the end, Phosphor will be responsible for detecting calls to methods belonging to its tainting API and produce the expected functionality.

4.3 Summary

In this chapter, we presented the implementation details of our Floodgate prototype. Concretely, we provided an overview of how we implement the techniques we chose for both mobile

and server sides and the tools we rely on to achieve that. On the mobile-side, we presented AspectJ, a framework we applies to Android applications in order to provide the aspect-oriented concepts in which our mobile monitor component relies. On the server side, we presented two different tools that integrate Floodgate: (1) Phosphor, a tool which implements dynamic taint tracking capabilities in Java applications through code instrumentation in compile-time; and (2) Dropwizard, a *framework* that we use to implement our *servlet* basis and which provides tools and mechanisms for developers to build full-fledged web applications that mobile clients can interact with, which completely fits Floodgate's model. The next chapter presents the evaluation we performed to our system, along with concrete results and some critics.

5 Evaluation

In this chapter we present and discuss our evaluation of Floodgate, along with its results. In first place, we explain the evaluation methodology with detailed test plans, and the reasons for its choice. Next, we present the verified results, in the form of graphs, with mean and standard deviation values. Finally, we will discuss these results, presenting possible future optimizations. Our Floodgate evaluation was performed in three different strands: (5.1) performance evaluation; (5.2) qualitative evaluation through the development of a use-case application and (5.3) a security evaluation of our system.

5.1 Performance Evaluation

5.1.1 Methodology

Since Floodgate operates in an end-to-end fashion, in order to enforce data protection both on mobile and *backend* sides, its evaluation must also be performed on both sides. We evaluate Floodgate performance in three ways: (5.1.2) The end-to-end performance (5.1.3); the mobile performance and (5.1.4) the server-side performance. All of our experiments were performed on the following hardware: For the mobile side, an LG Nexus 5 (2014), 2GB RAM, 16 GB storage. For the backend, a single-core 2GHz virtual machine running on an Hewlett-Packard BladeCenter, 2GB RAM, running Debian 7 64-bit and a Ceph-distributed storage of 5GB. Also, we used Oracle's "HotSpot" JVM [1], version 1.7.0-79 to support our server applications.

End-to-end performance To evaluate the minimum request latency imposed by Floodgate, we developed a simple application which works as a *baseline* for test purposes. In the mobile endpoint, the application accesses the device's phone number and sends it through an HTTP request to the *backend*. The *backend*, by its turn, simply returns the same phone number,

[1] On Section 5.1.4, for the *tradesoap* and *tradebeans* benchmarks we used OpenJDK "IcedTea" JVM, version 1.7.0-79, due to some required classes not being present on Oracle's JVM.

concatenated with the string "ok!". Finally, the mobile device prints the `<phone_number> + ok!` message on the screen.

We measured the time spent in the execution of an application request in three main parts: (1) the mobile endpoint, in which Floodgate controls the access to the phone number and generates the network request; (2) the *backend*, responsible for receiving the resource and generating the response and (3) the network, responsible for delivering the network request and returning the correspondent response.

Mobile-side performance impact: Floodgate's operation on the mobile-side consists in controlling the access and propagation of sensitive information concerning the user and the device itself. Due to that, to evaluate the mobile-side performance impact introduced by Floodgate, we compared the performance of accessing multiple sensitive information sources when in the presence and absence of Floodgate. Our test plan consisted in accessing 10 different sensitive resources available through the Android APIs, and subsequently sending them through HTTP requests to the *backend*. Each resource was previously set to "protected" in the privacy manifest. Each access was performed in independent experiments, in order to guarantee no influence between accesses and minimal system load. The 10 resources tested were: the device's GPS location, bluetooth address, IMEI, the SIM card phone number, serial number, IMSI, the network SSID and the user's country, timezone and saved accounts. We measured the time interval between calling the desired resource until getting the HTTP request ready to be sent. This way, we measure the performance of the whole Floodgate mobile-side operation.

Server-side performance impact To evaluate the impact of dealing with instrumented data on the server-side, we defined four metrics to measure that impact: (1) the operations' latency; (2) the overhead in database accesses; (3) the resources usage efficiency and (4) the server throughput. For the operations' latency, we measure the execution time and memory usage of server-side programs instrumented for Floodgate. To this end, we leveraged the DaCapo [2] 9.12-bach macro-benchmark suite, which contains 14 benchmarks and simulates real-world applications within its workloads, manipulating multiple data types. We ran these benchmarks using the *"default"* size workload. We measured the execution time and maximum JVM heap usage.

Server Throughput: We also measure the maximum throughput of Floodgate's server side.

[2] `http://www.dacapobench.org/`

By throughput, we mean the maximum number of requests our server could respond to, before hitting the saturation point (i.e. the point in time when the server can no longer answer incoming requests, within a reasonable amount of time. To measure the throughput, we used the same *baseline* application used in Section 5.1.2 running on the same Floodgate server referred in Section 5.1.1 and on a non-Floodgate version of the same server. To generate the requests, we used a MacBook Pro 2014, featuring an Intel Core i7 quad-core 2,5 GHz, 16GB RAM DDR3L 1600 MHz running the Apache JMeter software, version 2.3 r1665067, with the following test configuration:

- Number of parallel clients (threads): 1, 500, 1000, 1500, 2000
- Target throughput (ops/s): 100, 200, 300, 400, 500, 600
- Ramp-up period: 5 seconds
- Duration: 20 seconds

This configuration means that, during a time period of 20 seconds, we put the referred number of concurrent clients (starting at equally-distributed periods over the test duration, e.g., 500 clients / 5 seconds = 100 clients starting each second) sending HTTP requests to our server application. We measured the response time for the client-generated requests, while limiting the throughput at the referred values.

Database operations (read/write) performance Another meaningful test plan we performed considered the performance of read/write operations of the *backend* database, since all common application *backends* have to persistently store their data. In this test plan, we took the simple application we created for the evaluation on 5.1.2 and created a simple database on our *backend* that could handle the phone numbers arriving there. Also, we implemented a method on the mobile *endpoint* application to query the database for the existing phone numbers. Here we measured the average time spent by Floodgate on two essential database operations: writing a new phone number to the database, and reading the existing phone numbers (by querying the database).

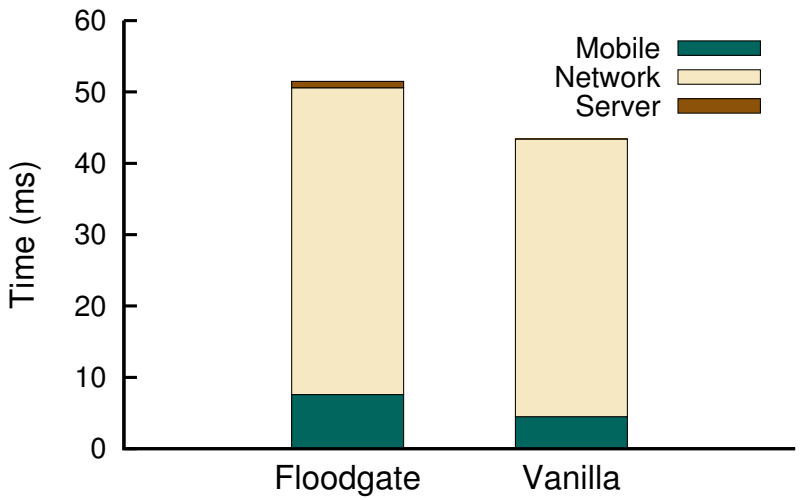

Figure 5.1: Mobile-server latency

5.1.2 End-to-end Performance:

By comparing the Floodgate and the non-Floodgate versions, we expected increases in both the mobile-side and server-side components, due to Floodgate's operation, but similar measures in the network component. This is because all that Floodgate adds to the network requests is a "privacy" header, which we think is not a relevant addition in terms of time to deliver the requests.

The results for all the components' execution times are represented in Figure 5.1. Floodgate introduces an impact of about 40% on the mobile-side performance, 10% on the network component and also an overhead of about 400 times on the server-side performance.

On the mobile-side, the observed results somehow matched with the expected ones. In this test, we accessed the device's phone number and sent it through the network, measuring the execution time on the mobile-side only, just as we did on Section 5.1.3 with multiple mobile resources. On that test, accessing the mobile phone number with Floodgate showed an overhead of about 40% as well.

The network component, with an average overhead of 10%, also makes us assuming it as a good result. We can justify the network overhead with the fact that, with Floodgate, each HTTP request will be received by an instrumented *backend*, which caused that some more overhead was introduced before we could measure the time spent on the network component. Also, each request will carry an additional HTTP header, the "privacy". Still, we consider that this result

does not harm the practical user experience with Floodgate applications.

Finally, the server-side verified an excessive overhead of about 400 times. On Section 5.1.4, we already measured the server-side performance under heavy operations, which showed an overhead of about 300 times. Although, this time our *backend* only executed a simple baseline operation of returning the input sent by the mobile device. Due to that, the execution times revealed themselves really small. The no-Floodgate version of the server spent an average of 0.0022 milliseconds to complete this task. Applying the Floodgate server instrumentation, this time increased to an average of 0.90 milliseconds. This way, we justify the excessive overhead with the fact that it can easily occur when execution times are of these order of magnitude.

5.1.3 Mobile-side Performance Impact

Figure 5.2 presents the test results, by comparing the time to perform the operations described above. The results show a high similarity between the experiments durations, except for the GPS location resource, since it requires an additional overhead caused by the Android framework asking for the location sensor refresh in order to get an updated device position. Also, the results show that Floodgate adds an average of 24% overhead. Concretely, Floodgate adds 1.5 *ms* (29%) to the phone number access time, 2.4 *ms* (13%) to the GPS, 1 *ms* (24%) to the accounts resource, 1.6 *ms* (26%) to the bluetooth address, 1.4 *ms* (17%) to the device's IMEI, 1.6 *ms* (23%) to the user's country, 1 *ms* (24%) to the timezone, 1.3 *ms* (25%) to the SIM serial, 1.9 *ms* (31%) to the subscriber ID and 0.9 *ms* (29%) to the SSID.

These increases are due to a) the interception of the resource call by our AOP module, in order to verify the resource's privacy settings and subsequent application taint according to that; and b) interception of the network requests and the privacy header addition according to the application taint. We consider these results as acceptable, not harming the user experience with applications. Also, these results show that Floodgate imposes roughly the same mobile-side overhead as TaintDroid does 2.2.1, without modifying the underlying mobile operating system, and Aurasium 2.2.2, while performing coarse-grained taint tracking, which Aurasium does not achieve.

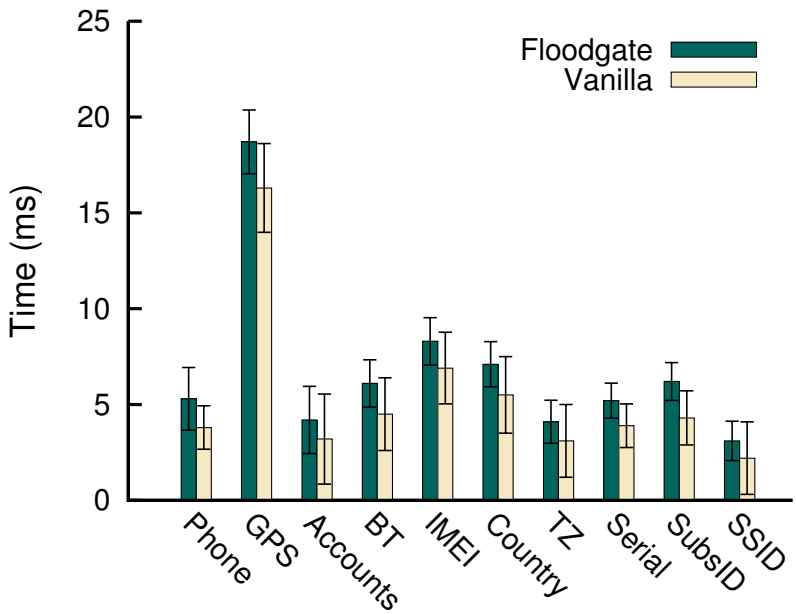

Figure 5.2: Mobile-side performance impact: (BT) Bluetooth Address, (TZ) Timezone, (Serial) SIM Serial, (SubsID) Subscriber ID

5.1.4 Server-side Performance Impact

The test results for the server-side performance evaluation are represented on Figures 5.3 and 5.4, for execution times and JVM heap usage, respectively. Analyzing the results, we spot that Floodgate introduces an average overhead to the execution of the application service of about three times on the *backend* performance. Despite the fact that it is a considerable value, each of the performed tests was specifically engineered to test different and heavy operations. Looking closely to Figure 5.3, we can conclude that Floodgate behaves better in some of the tests than in others. Concretely, Floodgate introduced lower overhead values when executing the Batik or Xalan benchmarks. On the one side, the Batik benchmark consists in converting image files from `.png` to `.svg` format. The Xalan benchmark, on the other side, transforms XML into HTML documents. In opposition, the Fop or Avrora benchmarks introduced higher overhead values. Their execution consist in parsing an XSL-FO file and converting it to PDF, and simulating a number of programs running on a grid of AVR microcontrollers, respectively. Their higher overhead may be explained with the fact that both tests perform multiple operations with arrays of primitive types. In such cases, Floodgate doubles the number of arrays to store the variables' tags. In Batik and Xalan tests most of the workload is performed with operations of variables assignment and loops, which are not so CPU-intensive after instrumentation. The

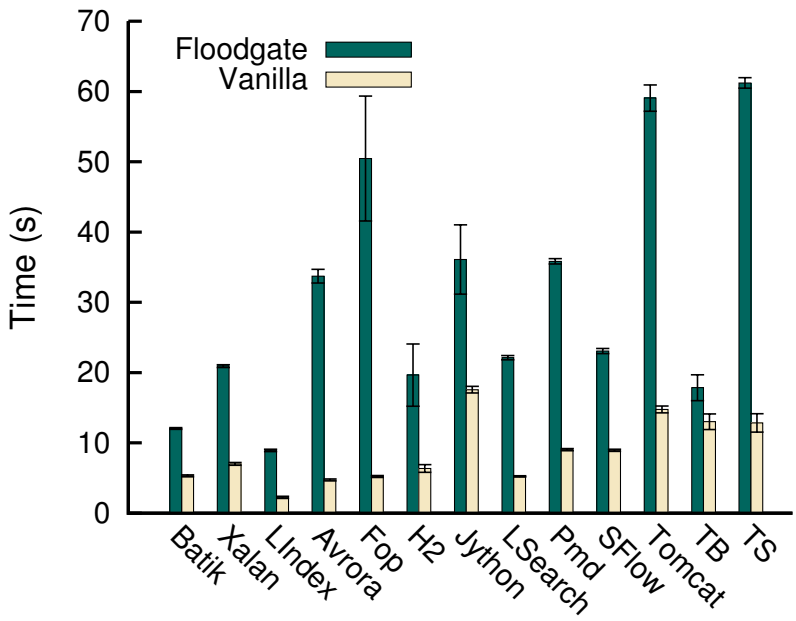

Figure 5.3: Server-side performance impact: (LIndex) LuIndex, (LSearch) LuSearch, (SFlow) SunFlow, (TB) TradeBeans, (TS) TradeSoap

verified results can be explained due to the overhead imposed by Phosphor's instrumentation process, which adds instructions to the code in order to propagate taint tags. In conclusion, Floodgate's *backend* performance can prove itself better or worse, depending on the kind of operations performed by the application at the server-side. Also, Phosphor have suffered some implementation changes from the time when it was evaluated with the DaCapo *framework* [11]. At that time, the *framework* revealed much better results. However, it was evaluated using the Integer tainting mode, instead of the Object tainting mode we use in Floodgate and that the author admits as much slower but not formally *benchmarked*. Due to that, we believe that these results may be improved with some optimizations in the *framework*.

The maximum memory heap usage test showed that Floodgate introduced an overhead of about 5.2%, when measuring the memory heap size when running the presented *benchmarks*. However, we can not compare these results with the ones we studied, due to lack of information.

5.1.5 Server Throughput

We present the results of this test on Figure 5.5. Analyzing the results, we can state that Floodgate does not have a great impact regarding the server's throughput or saturation

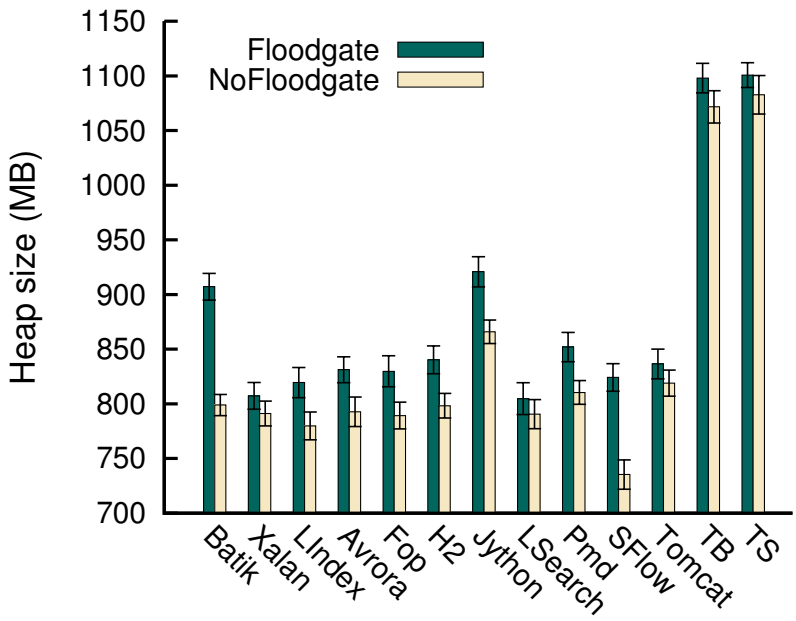

Figure 5.4: Server-side memory usage impact: (LIndex) LuIndex, (LSearch) LuSearch, (SFlow) SunFlow, (TB) TradeBeans, (TS) TradeSoap

point, since in both Floodgate and non-Floodgate cases they both feelingly achieved similar times. Concretely, Floodgate causes an average overhead of 11.8%, 24.5%, 6.5% and 10.3% with 500, 1000, 1500 and 2000 clients, respectively. With only one client sending requests to the server we did not verify any measurable overhead regarding the response time. This overhead can be explained with the fact that Floodgate's instrumentation process introduces additional instructions to perform taint tracking.

5.1.6 Database Read/Write Operations Performance

The results on Figure 5.6 show that Floodgate imposes an overhead of about 88% when writing an object to the database, and an overhead of about 95% when reading from the database. During read operations, the verified results can be classified as extremely positive. This is because every time an object is read from the database on a Floodgate server, another query is automatically executed to our persistence taint module, in order to find the respective taint tag, and assign it to the object. This way, an overhead of about 100% was expected, and the results match this expectation.

On the other side, write operations reveal a similar overhead (88%). Just like in read oper-

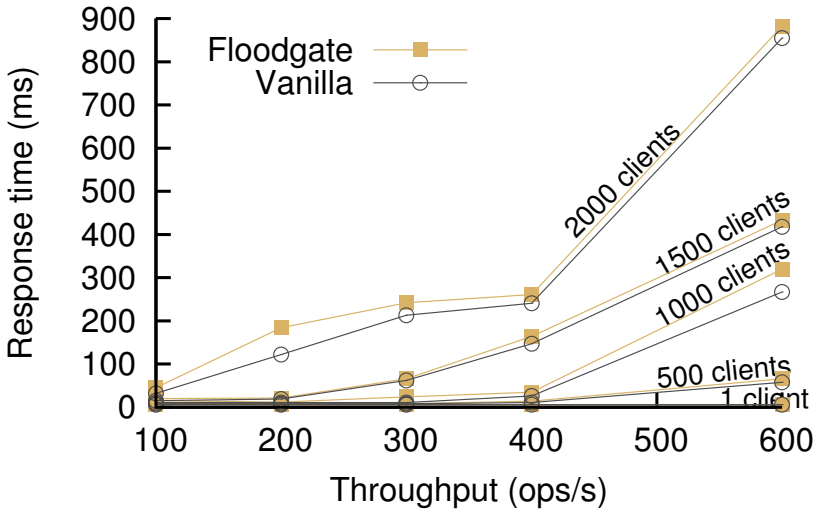

Figure 5.5: Server throughput

ations, when an object is written to the database Floodgate provides, another object containing the first's taint tag must be written to the persistence taint database. So, we can conclude that the results also match the expected ones, for the kind of the performed operations. In order to improve these results, we could perform both queries/writes in parallel, possibly using more memory/CPU of the host system but reducing these times

5.2 Use-case Application

Apart from the quantitative evaluation we presented on above, Floodgate was also qualitatively evaluated. To do that, we developed an use-case application, with both mobile and *backend* components, as near as possible to a real-world application which could at the same time provide a useful service to its users and show all the Floodgate's functionality. This way, we implemented AuctionsApp. As the name implies, it consists in an auctions app that allows users to post items for auctioning and live bidding on any available item except the ones they posted itself. The application data model and available operations are represented in Figure 5.7.

To test Floodgate's operation, we developed a permissions manifest in which we declared the device's phone number as the only private resource and our application *backend* as the only trusted endpoint. Then, we implemented an willful security flaw. For instance, every time a user puts a bid on any item, the mobile application will access the device's phone number, and send it through the network to the *backend*, along with the bid's information. Here, Floodgate

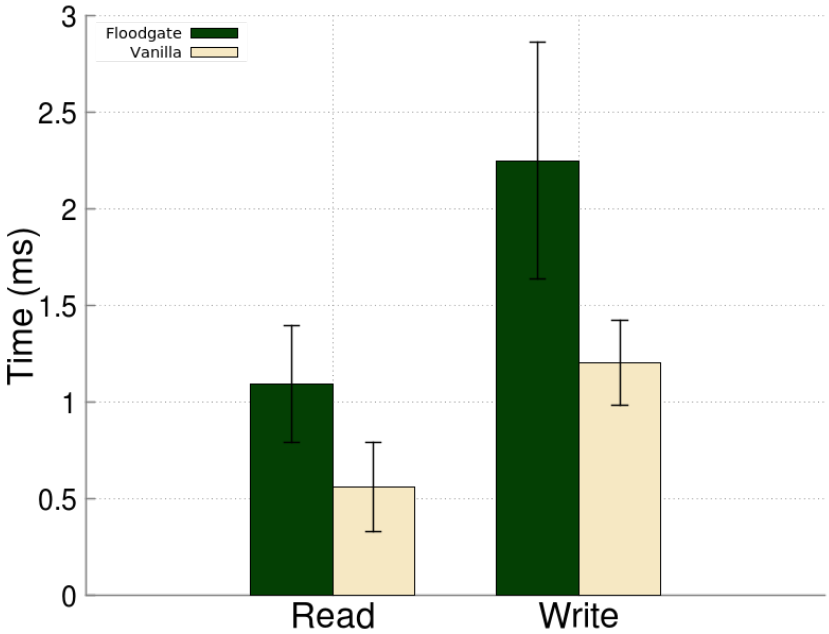

Figure 5.6: Floodgate database performance

ensures that all the requests exchanged between the mobile and server sides will include the "privacy" header to state whether the request's content is tainted or not.

On the server-side, the application receives the request and creates the bid, persistently storing the incoming information and updating the price of the item to which the bid applies. Also, Floodgate filters the request, enforcing incoming taints to be applied to data, right before this data enters the *backend's* application flow and enforces taints to be persistently stored at the same time as their correspondent data.

Every time a user asks for the list of current available items (whose auction periods haven't expired yet), it triggers an action on the Floodgate's server which can be decomposed on multiple parts: First of all, the application queries the persistent database for all the open auctions available. After that, Floodgate checks whether each one of these items have a correspondent persistent taints. If so, those taints are applied before the items leave the database and enter the application flow. After that, the application produces an HTTP response with the information of all available items and calls network methods to dispatch it. Although, this response will be filtered by the outcoming HTTP filters present on the *backend*, in order to check whether the content being exported is, therefore, tainted. If it is, Floodgate is able to block the response or to let it proceed based on certain checks, to be defined by the programmer itself (i.e. destination endpoint, taint type, etc.).

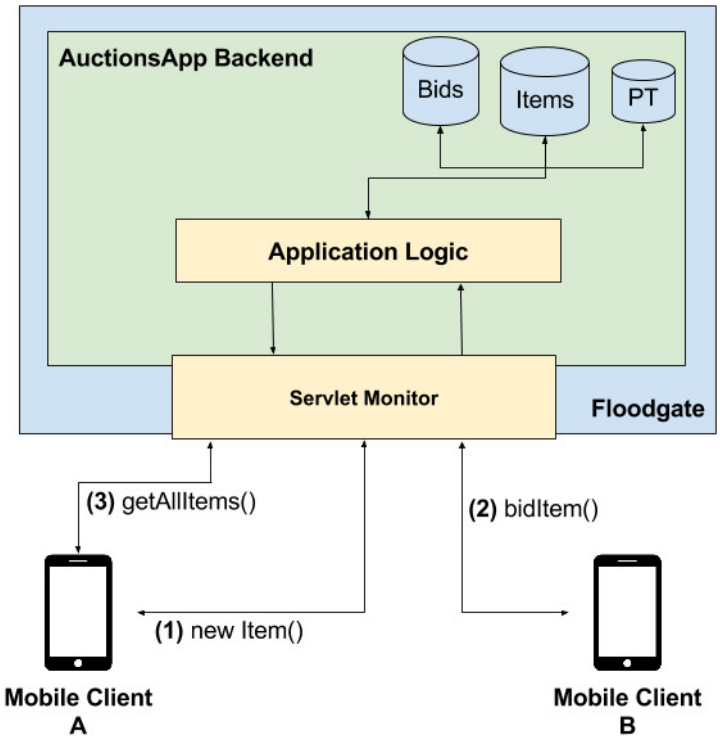

Figure 5.7: Auctions App use-case application

5.3 Security Evaluation

Apart from the quantitative evaluation and the use-case application, we also performed a security evaluation in which we describe our system's security limitations and optimizations performed in order to overcome them. Also, we describe how an attacker could behave against our system, becoming a threat.

Since Floodgate is an end-to-end IFC system, it is necessary to evaluate how well it performs against common IFC threats. Tracking implicit flows is one of the difficulties of IFC systems operating in *runtime*, since these only occur under certain conditions and can be equivalent to potentially dangerous explicit flows, which can be defined as passing data between variables. For example,

$$x := y \tag{5.1}$$

is an explicit flow of data between x and y, since x explicitly acquires the value of y. This explicit flow can be considered equivalent to the implicit flow

if x **then**

$y := true$

else

$y \quad :=$

$false$

end if

where no explicit assignment takes place, but at the end of the execution y has the same value as x, so we say that x implicitly flows to y.

This may have potentially dangerous consequences, since attackers may implicitly pass data between variables in order to surpass the taint tracking and propagation mechanisms featured on most *runtime* IFC systems. Floodgate provides a medium protection against this type of attacks, by tracking implicit flows on its server-side container but not the mobile endpoint. This is achieved using the protection offered by Phosphor. Although, on the mobile-side, Floodgate applications are indeed vulnerable to this type of threats. Implicit flows are a known limitation of IFC which affects almost every one of the studied systems, and despite we do not primarily focus on solving it, when designing Floodgate we tried to leverage its consequences on the server-side.

Another Floodgate limitation, present only on the mobile-side, is the high number of false positives regarding policy violations that our system might deal with. In our model, after an application requests access to a sensitive resource marked as `private` in the privacy manifest, the whole application will be tainted and every subsequent network access will report this taint, even if that specific resource is never sent over the network. Although, we performed an optimization to leverage this limitation. The mobile resources that comprise our *default* policies are very different between them, and due to its characteristics they may have sort of a lifetime. For example, when an application requests the device's location it will likely send it through the network as soon as possible, because otherwise, it may become inaccurate since the location may change quickly. This way, we defined time thresholds for certain taints (e.g. the location ones) to be cleaned, and this way to lower the false positives number. With this, we can conclude that mobile applications feature a very characteristic programming model in which some resources must actually be used (i.e. sent to the server-side) right after they are accessed, and this can be a good starting point for some more optimizations like the ones we implemented.

5.4 Summary

In this chapter, we presented the evaluation performed to our system. We performed an important set of tests in order to measure every important aspect of Floodgate, on mobile and server-sides, apart from end-to-end tests which evaluated the system's operation as a whole. Despite the fact that we couldn't compare every result with studied systems, due to lack of information, we conclude that in most cases Floodgate is able to match the results presented in literature or even overcome them.

6 Conclusions

We have described the architecture, implementation, and experimental evaluation of Floodgate, a system that aims at protecting private sensitive data when accessed by *backend*-supported mobile applications, in and end-to-end way. Our experimental results showed that, despite Floodgate incurs a significant overhead on both the mobile and the server sides of an application, it does not harm the user experience and does not require significant changes in the application programming and deployment model. Also, our results demonstrate that Floodgate introduces less overhead at the mobile-side, commonly the most resource-constrained component of this type of applications. Floodgate has been implemented as open source and is available for experiments and further improvements.

As future work we would like to perform more optimizations. On the mobile-side, we would like to develop a technique which could leverage the number of false-positives in terms of data that is considered as tainted, while keeping all of our requirements. This said, we could apply a static analysis tool like Aurasium [2] or FlowDroid [3] in order to perform the heavy workload of taint tracking on compile time, trying to partition Floodgate's mobile applications into code which has to be controlled by our AOP mechanism and code which is considered as not dangerous, and thus preventing AOP from tainting data when interacting with such parts of the application.

In terms of the definition of privacy policies, we would like to automatically generate *aspects* for the defined security policies, and allow programmers to define those policies in a generic way, allowing every kind of method to be tracked by our AOP module, instead of the current solution where a pre-defined set of the device resources are monitored.

On the server-side, our priority would be to improve the server performance. Despite the fact that Information Flow Control is traditionally heavy in terms of performance, we believe that some tuning both in the tool we use to provide dynamic taint tracking, Phosphor [11], and in our *servlet* basis, could result in more acceptable performance results and could be a step

forward in server-side IFC.

Bibliography

W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pp. 1–6, USENIX Association, 2010.

R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, (Berkeley, CA, USA), USENIX Association, 2012.

S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, pp. 259–269, June 2014.

S. Chong, K. Vikram, and A. C. Myers, "Sif: Enforcing confidentiality and integrity in web applications," in *In Proc. 16th USENIX Security*, 2007.

A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 291–304, 2009.

P. Hosek, M. Migliavacca, I. Papagiannis, D. M. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch, "Safeweb: A middleware for securing ruby-based web applications.," in *Middleware*, vol. 7049 of *Lecture Notes in Computer Science*, pp. 491–511, Springer, 2011.

D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pp. 47–60, 2012.

S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov, "Ïbox: A platform for privacy-preserving apps," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 501–514, USENIX, 2013.

A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *In Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pp. 129–142, 1997.

G. Kiczales, "Aspect-oriented programming," *ACM Comput. Surv.*, vol. 28, Dec 1996.

J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, (New York, NY, USA), pp. 83–101, ACM, 2014.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, (London, UK, UK), pp. 327–353, Springer-Verlag, 2001.

A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, pp. 410–442, October 2000.

J. Bacon, D. Eyers, T.-M. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch, "Information flow control for secure cloud computing," *Network and Service Management, IEEE Transactions on*, vol. 11, pp. 76–89, March 2014.

N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, pp. 6 pp.–263, May 2006.

P. Vogt, F. Nentwich, N. Jovanovic, and E. Kirda, "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.," *NDSS*, 2007.

P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: Being lazy with class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pp. 12–1–12–55, ACM, 2007.

B. D. Payne, R. Sailer, R. Cáceres, R. Perez, and W. Lee, "A layered approach to simplified access control in virtualized systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 12–19, July 2007.

P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," *SIGOPS Oper. Syst. Rev.*, vol. 39, pp. 17–30, October 2005.

M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 321–334, 2007.

N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pp. 19–19, USENIX Association, 2006.

E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider, "Logical attestation: An authorization architecture for trustworthy computing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 249–264, ACM, 2011.

I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pp. 63–74, 2009.

N. Zeldovich, S. Boyd-wickizer, and D. MaziÃ¨res, "Securing distributed systems with information flow control," in *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI*, pp. 263–278, 2006.

R. Spahn, J. Bell, M. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser, "Pebbles: Fine-grained data management abstractions for modern operating systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 113–129, USENIX Association, Oct 2014.

A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pp. 49–54, ACM, 2011.

B. Parno, J. McCune, D. Wendlandt, D. Andersen, and A. Perrig, "Clamp: Practical prevention of large-scale data leaks," in *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 154–169, May 2009.

M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch, "Defcon: High-performance event processing with information security," in *USENIX Annual Technical Conference (USENIX ATC'10)*, USENIX, USENIX, June 2010.

A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pp. 228–241, 1999.