Floodgate: An Information Flow Control System to Prevent Data Leaks from Distributed Mobile Apps

Sérgio Moura
Técnico Lisboa, University of Lisbon

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.

1 Introduction

Increasingly people use mobile-cloud applications today's era. Data travels between mobile devices to the cloud back and forth. However, it is difficult to keep track of how this data is going to be used. First, policies are verbose and difficult to understand and to enforce in practice. Second, people commit mistakes when building such apps, which can lead to data loss or data breaches in the cloud backend. Third, policies change and there is currently no way to let users be aware that their data is no longer being handled according to the original terms.

In this paper, we present Floodgate, a platform that employs Information Flow Control (IFC) [8] techniques for building mobile-cloud apps in which the data can be confined to operate between both endpoints only. Our goals are the following: (1) compatibility with current Android mobile platform, not requiring changes in the mobile OS; (2) simple policy specification methods, in a way easy to express by the developers and understandable by the users; and (3) efficient use of resources, specially on the mobile-side, minimizing the performance overhead imposed by the system.

In Floodgate, applications' programmers mark a few data sources on the mobile cloud endpoint, defines for each of such sources whether the data source is to be private / protected / public, and the system ensures that the data is confined within the mobile-cloud premises as specified by the policy. The system tracks illegal accesses that violate the policy, e.g., sending the data outside the cloud, and throws an exception. This way of indicating the policy is very simple to understand. Once the data is in the cloud, the system keeps track of the tainted data and ensures that it can only be released for future application versions that are compatible.

We faced three main challenges in building Floodgate. The first challenge is about tracking granularity and compatibility. In order to keep track of how information flows from the mobile endpoint source until it is manipulated in the cloud backend, we need some mechanism to enforce IFC policies in the mobile, in the cloud, and propagate the taints between endpoints. In the cloud, systems like Nexus [9], HiStar [12], SIF [2], or variants of these systems can be used to enforce fine-grained IFC policies. Some of them, like Nexus and HiStar, require changes to the server-side operating system and can be easily deployed by the cloud provider. To transmit the taint between endpoints, we can do this without changing protocols by attaching taint information to HTTP headers and export a REST API to the client side. However, taint tracking on the client-side is harder to achieve. Implementing fine-grained taint tracking mechanisms requires changing the OS. Either by deploying a modified Dalvik VM like TaintDroid [3] or by instrumenting Androids classes to implement dynamic taint tracking using source

code instrumentation. This would require us to change the Android platform, which would hinder the deployment of our system. Instead, we look for a more practical solution that does not require deep changes to the Android, even if we have to trade off some tracking granularity in favor of portability. Changes could also have to do with changes in the programming environment, making it hard for developers to build apps.

The second challenge has to do with complexity of policy specification to both the programmer and to the user. Traditionally, IFC systems require programmers to specify IFC policies deep in the code making it hard to separate code from policy, and therefore hindering the maintainability of the code. In addition, it is often the case where policies are extremely cumbersome to specify by the programmers. Look at SIF, for example in Listing 1. This makes this very complex to maintain and hard for the users to grasp. Ironically, could increase the chance of introducing bugs to the policies themselves.

Finally, the third challenge involves performance overheads. As it is widely known, IFC incurs performance overheads. We aim at an IFC system which at least incur less overheads to the client side given that this is a more resource-constrained device that in the cloud.

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

Listing 1: SIF security policy example

To address these challenges, Floodgate implements the following techniques. First, taint tracking is based on API accesses. The sources and sinks correspond to methods of the client- and cloud-side APIs that are marked as sources and sinks. Each of these sources is then annotated with three simple labels: *private*, *protected* and *public*. Private means that a given item never leaves the mobile endpoint. Protected means that it never leaves the mobile-cloud endpoints. And public means that it can be sent away outside the cloud. Naturally more complicated things could be envisioned, e.g., exporting data somewhere outside, etc. However, that would complicate the policy specification and interpretation. By this, we make it very simple.

Second, we use aspect oriented programming (AOP) [6] at the mobile endpoint for taint tracking. Aspects are used on the mobile client endpoint and control accesses at the source and at the sink. Because of this, we dont have to change the OS, preserving portability, and we dont have to enforce fine-grained IFC at the mobile side, thereby incurring small performance overheads. What we trade here is a coarser grained information flow control because once the data is accessed it is as if the entire application is tainted. To alleviate this limitation, we implement a few optimizations which take advantage of Androids specific programming model.

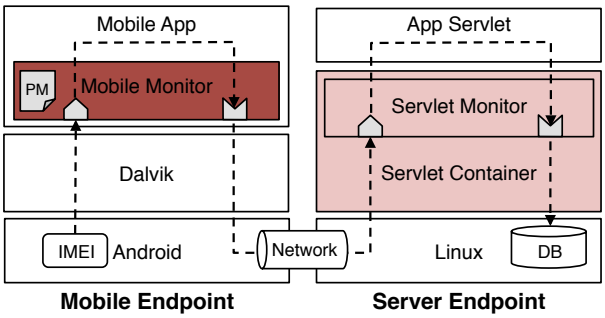

Figure 1: Architecture of Floodgate illustrating how the IMEI is tracked from the mobile- to the server-side of an application. Floodgate's components are shaded.

Third, on the server side, we employ dynamic taint tracking using source code instrumentation. This allows us to increase the granularity of tracking (meaning we gain precision) at the expense of requiring changes to the runtime environment and increased performance penalty. Increased granularity is necessary because tainting the entire backend would be too restrictive and unnecessary.

To implement our architecture, we rely on three different tools: (1) AspectJ [1] on Android, a *framework* for implementing the AOP concerns on the mobile-side; (2) Phosphor [1], a static analysis and code instrumentation tool to achieve the dynamic taint tracking on the server-side; and (3) Dropwizard [2], a *framework* which serves as our *servlet* basis and offers developers tools for implementing web applications based on REST APIs.

We have built a prototype of the system and evaluated it. As it is typical in IFC systems, we observed a performance penalty for the applications. The performance penalty is comparable with other IFC systems. To reduce the performance impact, we propose a design pattern for certain operations that are more CPU intensive.

2 Design

2.1 Overview

We present Floodgate, a system that enforces information flow control policies in distributed mobile applications. Figure 1 illustrates the architecture of our system. Floodgate operates at middleware-level, tying together two endpoints of an application: the *mobile app*, and the

[1] https://eclipse.org/aspectj/
[2] http://dropwizard.io

app servlet. The mobile app is a regular Android application that can read privacy-sensitive information from local resources (e.g., the device's IMEI). This information can then be sent to the app servlet, which is a RESTful service running on a remote server. The application programmer writes both these components along with a *privacy manifest* (PM) that controls how the sensitive data can be accessed by both parts.

2.2 Programming Model

Floodgate offers a programming and deployment model specifically targeted to *backend*-supported mobile applications. On the mobile endpoint, the programming model doesn't suffer any major modifications, since developers only need to add the Floodgate mobile library as a dependency for the mobile application and the permissions manifest as an application's resource. The only reservation is that 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 doesn't suffer any modifications when comparing to the current model.

A simplified example of a permissions manifest file is represented on Listing 2. 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>`.

```
<?xml version="1.0"?>
<privacy>
    <resource>
        <id>IMEI</id>
        <access>protected</access>
        <trusted>
            <endpoint>http://acme.com</endpoint>
        </trusted>
    </resource>
</privacy>
```

Listing 2: Privacy manifest specifying the privacy access modifier for the IMEI resource.

On the server endpoint, Floodgate's programming model is essentially based on the definition of a REST API to be deployed on the secure container.

```
@Path("/imeiSvc")
public class ImeiServlet extends
    AbstractServlet<Imei> {

    @POST
    @Path("/push")
    public void srvImeiPush(Imei imei) {
      persist(imei);
      return;
    }
    /* method for pulling the IMEI from the DB */
}
```

Listing 3: Skeleton of the servlet that implements the IMEI service example

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.

In order to compile and deploy a Floodgate application, developers have to compile Floodgate's mobile library which implements our mobile monitor component as a dependency of the mobile application. After that, developers must define a privacy manifest like the one presented in Listing 2, specifying data they are willing to protect and the corresponding privacy keys. After that, the deploying model doesn't differ from the traditional model of packaging, signing and launching the application in application stores. On the server-side, firstly developers have to produce the application data model and interfaces to communicate with the mobile client. After the application logic is complete, the service provider has to run Floodgate's static instrumentation tool in order to grant taint tracking capabilities for the web application. In the end, the server-side application is a single self-contained package ready to be deployed on any web-powered server.

2.3 End-to-end taint tracking

Step 1: When a mobile application 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.

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 2.2, 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.

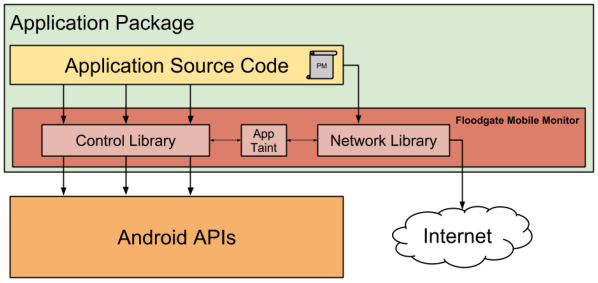

Figure 2: Floodgate mobile component design

2.4 The mobile endpoint

On the mobile-side, Floodgate extends common mobile applications with two components, in order to achieve the application-level taint tracking: The *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 2.

The Floodgate's mobile monitor, by its turn, is comprised by two modules: The control library, which contains the behavior of controlling the access to the resources specified in the permissions manifest and their propagation, and 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 library relies on aspect-oriented programming (AOP) concepts.

AOP relies on partitioning 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.

Our library 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.

#	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 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 using the device's browser.

Apart from that, our mobile library also 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 library features some "concerns" regarding the access to other network libraries instead of ours, blocking its execution.

2.5 The server endpoint

On the **server-side**, Floodgate provides a secure container where to deploy the *backend* of mobile applications. As referred in Section 2.4, the Floodgate network library deals with the propagation of application taints to the server-side.

In order to continue the enforcement of such security policies on the server-side, we apply concepts of static analysis, 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, graphically presented on Figure 3. This process consists in identifying the taint sources, sinks, and points where data interact with each other. After the analysis process, Floodgate instruments the application, applying code injection in order to propagate taints wherever the static analysis process dictates to. Finally, we apply dynamic taint tracking in order to keep track of when tainted data are set to leave the system, acting accordingly. 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.

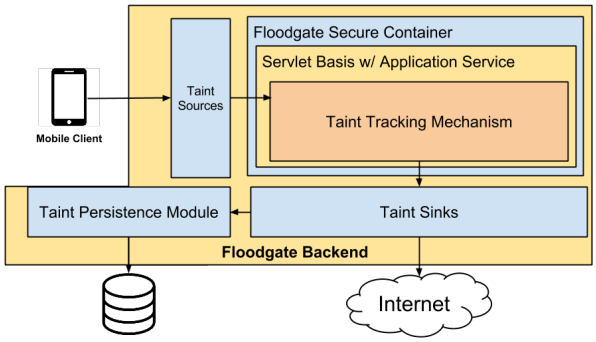

Figure 3: Floodgate server component design

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 con-

crete data, before it can be manipulated by the application or returned to the client.

3 Implementation

3.1 Mobile-side components

In order to implement the mobile architecture described in Section 2.4, we rely on three different components: (1) our privacy manifest; (2) our control library and (3) network library. Following the Android standard for resources, we implemented the privacy manifest as a XML file which might be placed under the `resources/raw` folder on the mobile application project. After our control library performs the parsing of this file, each sensitive permission is stored, along with its privacy value, in a Java `HashMap` data structure for faster accesses.

In order to intercept accesses to sensitive resources, our control library is implemented as an application library which must be added as a project dependency. Concretely, it is a set of Java classes, each one representing an *aspect*, annotated with the `@Aspect` annotation. Every *aspect* is implemented using the AspectJ *framework* and can be defined as two separate parts which together define where in the code they operate: (1) a pointcut, defining which method(s) to intercept; and (2) the advice, which defines the code that should be executed whenever the execution of the pointcut method is intercepted.

Finally, our network library is responsible for providing network capabilities to Floodgate applications (e.g., perform HTTP requests). This library abstracts the OkHttp [3] network library, and adds a functionality regarding our end-to-end taint propagation mechanism. For instance, since Floodgates 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. in order to obligate developers to use our network library (due to taint propagation features), we also rely on our control library to block the execution of other third-party or native communication libraries which do not enforce taint propagation.

3.2 Server-side components

On the server-side, we had to implement two main components that comprise Floodgate's *backend* design: (1) the servlet basis and (2) the taint tracking and propagation mechanism. For the servlet basis, we used the Dropwizard framework [4]. This tool targets simple implementation and deployment of RESTful services, offering interfaces for developers to focus exclusively on the application behavior instead of other side configurations usually required (e.g., web server configurations, logging tools, performance metrics).

In order to enable Floodgates 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. To achieve that, we used the filtering library provided by Dropwizard to implement filters that check the "privacy" header of each HTTP request. 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.

In order to perform taint tracking and propagation on the server-side's application, we used Phosphor [1] 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. Floodgate 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.

4 Evaluation

4.1 Methodology

We evaluate Floodgate performance in three ways: (4.2) The end-to-end performance (4.4); the mobile performance and (4.5) 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 Ora-

[3] `http://square.github.io/okhttp/`
[4] `http://www.dropwizard.io/`

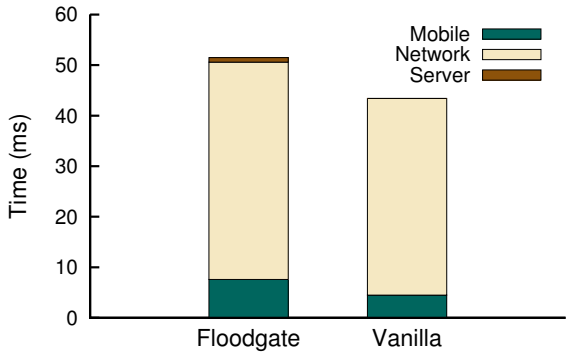

Figure 4: End-to-end performance

cle's "HotSpot" JVM [5], version 1.7.0-79 to support our server applications.

4.2 End-to-end performance

To evaluate the minimum request latency imposed by Floodgate, we developed a simple application which 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, 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. The results for all the components' execution times are represented in Fig. 4.

The results showed that Floodgate introduced an impact of about 69% 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 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 4.4 with multiple mobile resources. 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 4.4 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 4.5, 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.

4.3 Server throughput

We also measure the maximum throughput of Floodgate's server side. 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 4.2 running on the same Floodgate server referred in Section 4.1 and on a non-Floodgate version of the same server. We used 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

[5] On Section 4.5, 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.

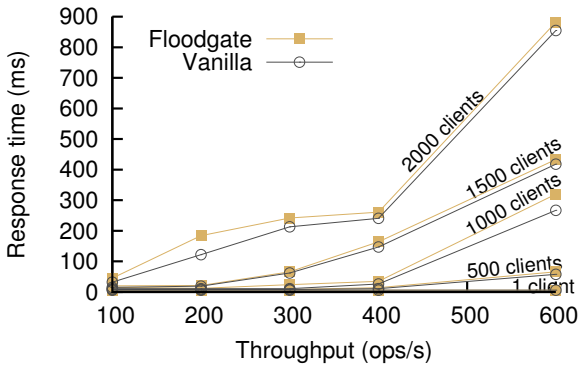

Figure 5: Server throughput

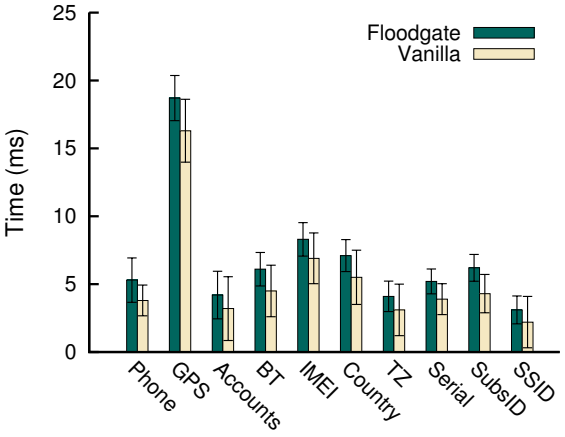

Figure 6: Mobile-side performance impact

server application. We measured the response time for the client-generated requests, while limiting the throughput at the referred values. We present the results of this test on Fig. 5.

Analyzing the results, we can state that Floodgate doesn't have a great impact regarding the server's throughput or saturation 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.

4.4 Mobile-side performance overhead

To evaluate the mobile-side performance impact introduced by 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. We measured the time interval between calling the desired

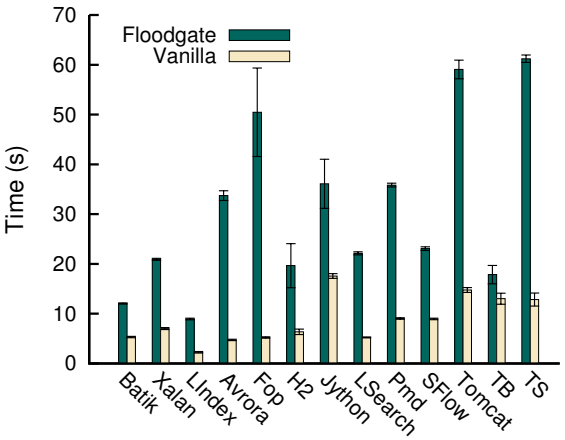

Figure 7: Server-side performance impact

resource until getting the HTTP request ready to be sent. This way, we measure the performance of the whole Floodgate mobile-side operation. Figure 6 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 32% overhead. This increase is 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, without modifying the underlying mobile operating system, and Aurasium, while performing coarse-grained taint tracking, which Aurasium does not achieve.

4.5 Server-side performance overhead

For the operations' latency on the server-side, we measure the execution time and memory usage of server-side programs instrumented for Floodgate. To this end, we leveraged the DaCapo [6] 9.12-bach macro-benchmark suite, which contains 14 benchmarks and simulates real-world applications, manipulating multiple data types. We ran these benchmarks using the *"default"* size workload and measured the execution time and maximum JVM heap usage. The test results are represented on Fig.

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

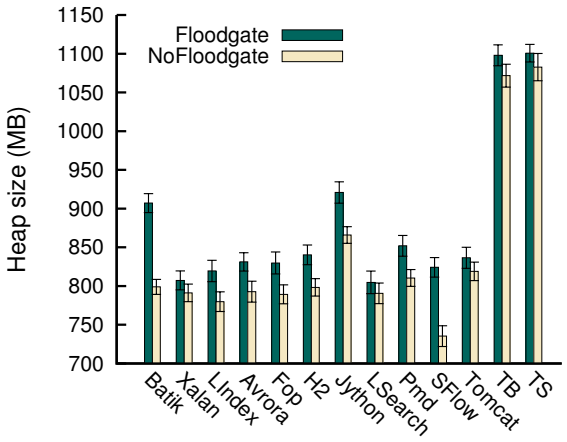

Figure 8: Server-side memory usage impact

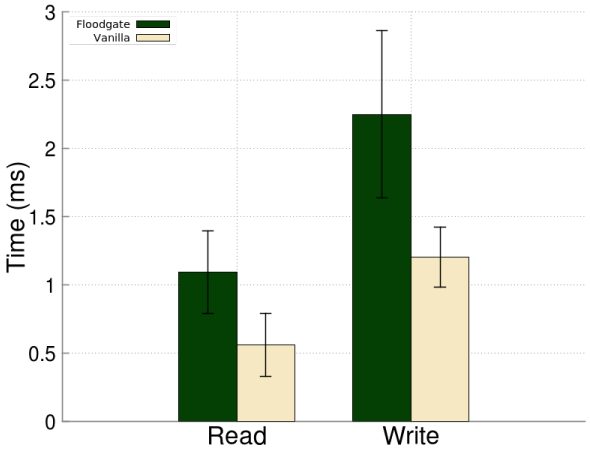

Figure 9: Floodgate database performance

7 and 8, for execution times and JVM heap usage, respectively.

The test results for the server-side performance evaluation are represented on Figures 7 and 8, 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 7, 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 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.

4.6 Database performance overhead

Another meaningful test plan we performed considered the performance of read/write operations of the *backend* database. In this test plan, we took the simple application we created for the evaluation on 4.2 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).

The results on Figure 9 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.

4.7 Use-case application

Apart from the quantitative evaluation, we developed AuctionsApp, an use-case application with both mobile and *backend* components, as near as possible to a real-world application which allows its users to perform real-time auctions. To test Floodgate's operation, we developed a permissions manifest in which we declared the device's phone number as a private resource, and 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 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.

5 Related Work

Lately, there have been a lot of investigation in privacy protection on mobile systems. This conducted to the appearance of multiple systems with different goals, like protecting sensitive data or controlling privilege escalation intents by malicious mobile applications. For example, TaintDroid [3], is an extension of the Android mobile operating system which relies on dynamic taint tracking to detect and report situations where applications try to leak private information regarding the device or its user (e.g., IMEI, phone number). Pebbles [10], another mobile IFC system, produces higher-level abstractions for common resources present in mobile devices, named *logical data objects* (LDO), allowing the definition of policies over these abstractions. Although, these system operate exclusively at the mobile-side, losing control whenever data is exported over the network.

Also, there are some systems with similar goals which operate exclusively on the server-side. These employ techniques which vary a lot between them: On the one hand, Flume [7] relies on IFC at the level of Linux processes, allowing the definition of labels and privacy policies over these processes to control the execution of multiple applications on the same server. SIF [2], on the other hand, applies IFC at the language-level, where privacy concerns must be explicitly written along with applications, allowing further deployment of secure applications into traditional containers like Apache Tomcat. Also, there are systems operating at a higher level, like RESIN [11], which operates within a language runtime, like Python or PHP, allowing the definition, of policy objects to express privacy concerns, data flow assertions to detect policy violations and filter objects to define where in applications should the data assertions be verified against the security policies. But, as in mobile systems, these perform control over server-side operations only, which stays short for our purposes.

There are also record of systems which try to achieve data protection in an end-to-end way, just like Floodgate. Hails [4] and SAFE WEB [5] are some examples of *frameworks* to develop web applications with end-to-end security concerns. Although they achieve protection between client and server-sides, the referred systems don't suit Floodgate's model. This is due to the chosen programming models and policy-definition mechanisms, since they are best tailored to fit web applications which programming model is very different from the one behind native mobile applications.

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 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.

References

[1] BELL, J., AND KAISER, G. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proc. of OOPSLA* (2014).

[2] CHONG, S., VIKRAM, K., AND MYERS, A. C. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. of USENIX Security* (2007).

[3] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of OSDI* (2010).

[4] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proc. of OSDI* (2012).

[5] HOSEK, P., MIGLIAVACCA, M., PAPAGIANNIS, I., EYERS, D. M., EVANS, D., SHAND, B., BACON, J., AND PIETZUCH, P. Safeweb: A middleware for securing ruby-based web applications. In *Proc. of Middleware* (2011).

[6] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-Oriented Programming. In *Proc. of ECOOP* (1997).

[7] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard os abstractions. In *Proc. of SOSP* (2007).

[8] MYERS, A. C., AND LISKOV, B. A Decentralized Model for Information Flow Control. In *Proc. of SOSP* (1997).

[9] SIRER, E. G., DE BRUIJN, W., REYNOLDS, P., SHIEH, A., WALSH, K., WILLIAMS, D., AND SCHNEIDER, F. B. Logical Attestation: An Authorization Architecture For Trustworthy Computing. In *SOSP* (2011).

[10] SPAHN, R., BELL, J., LEE, M., BHAMIDIPATI, S., GEAMBASU, R., AND KAISER, G. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *Proc. of OSDI* (2014).

[11] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving Application Security with Data Flow Assertions. In *Proc. of SOSP* (2009).

[12] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proc. of OSDI* (2006).