FlowFence: Practical Data Protection for Emerging IoT Application Frameworks

Earlence Fernandes[1], Justin Paupore[1], Amir Rahmati[1], Daniel Simionato[2]
Mauro Conti[2], Atul Prakash[1]
[1]*University of Michigan* [2]*University of Padova*

Abstract

Emerging IoT programming frameworks enable building apps that compute on sensitive data produced by smart homes and wearables. However, these frameworks only support permission-based access control on sensitive data, which is ineffective at controlling *how* apps use data once they gain access. To address this limitation, we present FlowFence, a system that requires consumers of sensitive data to declare their intended data flow patterns, which it enforces with low overhead, while blocking all other undeclared flows. FlowFence achieves this by explicitly embedding data flows and the related control flows within app structure. Developers use FlowFence support to split their apps into two components: (1) A set of Quarantined Modules that operate on sensitive data in sandboxes, and (2) Code that does not operate on sensitive data but orchestrates execution by chaining Quarantined Modules together via taint-tracked *opaque handles*—references to data that can only be dereferenced inside sandboxes. We studied three existing IoT frameworks to derive key functionality goals for FlowFence, and we then ported three existing IoT apps. Securing these apps using FlowFence resulted in an average increase in size from 232 lines to 332 lines of source code. Performance results on ported apps indicate that FlowFence is practical: A face-recognition based door-controller app incurred a 4.9% latency overhead to recognize a face and unlock a door.

1 Introduction

The Internet of Things (IoT) consists of several data-producing devices (*e.g.,* activity trackers, presence detectors, door state sensors), and data-consuming apps that optionally actuate physical devices. Much of this data is privacy sensitive, such as heart rates and home occupancy patterns. More importantly, we are seeing an emergence of application frameworks that enable third party developers to build apps that compute on such data—Samsung SmartThings [55], Google Brillo/Weave [30], Vera [5], and Apple HomeKit [8] are a few examples.

Consider a smart home app that allows unlocking a door via face recognition using a camera at the door. Home owners may also want to check the state of the door from a secure Internet site (thus, the app requires Internet access). Additionally, the user also wants to ensure that the app does not leak camera data to the Internet. Although this app is useful, it also has the potential to steal camera data. Therefore, enabling apps to compute on sensitive data the IoT generates, while preventing data abuse, is an important problem that we address.

Current approaches to data security in emerging IoT frameworks are modeled after existing smartphone frameworks (§2). In particular, IoT frameworks use permission-based access control for data sources and sinks, but they do not control *flows* between the authorized sources and sinks. This method has already proved to be inadequate, as is evident from the growing reports of data-stealing malware in the smartphone [73] and browser extension spaces [36, 14]. The fundamental problem is that users have no choice but to take it on faith that an app will not abuse its permissions. Instead, we need a solution that forces apps to make their data use patterns explicit, and then enforce the declared information flows, while preventing all other flows.

Techniques like the recognizer OS abstraction [39] could enable privacy-respecting apps by reducing the fidelity of data released to apps so that non-essential but privacy violating data is removed. However, these techniques fundamentally depend on the characteristics of a particular class of applications (§7). For example, image processing apps may not need HD camera streams and, thus, removing detail from those streams to improve privacy is feasible. However, this may not be an option in the general case for apps operating on other types of sensitive data.

Dynamic or static taint analysis has been suggested

as a method to address the limitations of the above permission-based systems [60, 53]. Unfortunately, current dynamic taint analysis techniques have difficulty in dealing with implicit flows and concurrency [59], may require specialized hardware [70, 54, 65], or tend to have significant overhead [48]. Static taint analysis techniques [9, 21, 66, 45] alleviate run-time performance overhead issues, but they still have difficulty in handling implicit flows. Furthermore, some flow-control techniques require developers to use special-purpose languages, for example, JFlow [45].

We present FlowFence, a system that enables robust and efficient flow control between sources and sinks in IoT applications. FlowFence addresses several challenges including not requiring the use of special-purpose languages, avoiding implicit flows, not requiring instruction-level information flow control, supporting flow policy rules for IoT apps, as well as IoT-specific challenges like supporting diverse app flows involving a variety of device data sources.

A key idea behind FlowFence is its new information flow model, that we refer to as *Opacified Computation*. A data-publishing app (or sensitive source) tags its data with a taint label. Developers write data-consuming apps so that sensitive data is only processed within designated functions that run in FlowFence-provided sandboxes for which taints are automatically tracked. Therefore, an app consists of a set of designated functions that compute on sensitive data, and code that does not compute on sensitive data. FlowFence only makes sensitive data available to apps via functions that they submit for execution in FlowFence-provided sandboxes.

When such a function completes execution, FlowFence converts the function's return data into an *opaque handle* before returning control to the non-sensitive code of the app. An opaque handle has a hidden reference to raw sensitive data, is associated with a taint set that represents the taint labels corresponding to sensitive data accessed in generating the handle, and can only be dereferenced within a sandbox. Outside a sandbox, the opaque handle does not reveal any information about the data type, size, taint label, any uncaught exception in the function, or contents. When a opaque handle is passed as a parameter into another function to be executed in a sandbox, the opaque handle is dereferenced before executing the function, and its taint set added to that sandbox. When a function wants to declassify data to a sink, it makes use of FlowFence-provided Trusted APIs that check <source, sink> flow policies before declassifying data. The functions operating on sensitive data can communicate with other functions, and developers can chain functions together to achieve useful computations but only through well-defined FlowFence-controlled channels and only through the use of opaque handles.

Therefore, at a high level, FlowFence creates a data flow graph at runtime, whose nodes are functions, and whose edges are either raw data inputs or data flows formed by passing opaque handles between functions. Since FlowFence explicitly controls the channels to share handles as well as declassification of handles (via Trusted API), it is in a position to act as a secure and powerful reference monitor on data flows. Since the handles are opaque, untrusted code cannot predicate on the handles outside a sandbox to create implicit flows. Apps can predicate on handles within a sandbox, but the return value of a function will always be tainted with the taint labels of any data consumed, preventing apps from stripping taint. An app can access multiple sources and sinks, and it can support multiple flows among them, subject to a stated flow policy.

Since sensitive data is accessible only to functions executing within sandboxes, developers must identify such functions to FlowFence—they encapsulate functions operating on sensitive data in Java classes and then register those classes with FlowFence infrastructure. Furthermore, FlowFence treats a function in a sandbox as a blackbox, scrutinizing only communications into and out of it, making taint-tracking efficient.

FlowFence builds on concepts from systems for enforcing flow policies at the component level, for example, COWL for JavaScript [63] and Hails for web frameworks [28, 52]. FlowFence is specifically tailored for supporting IoT application development. Specifically, motivated by our study of three existing IoT application frameworks, FlowFence includes a flexible Key-Value store and event mechanism that supports common IoT app programming paradigms. It also supports the notion of a discretionary flow policy for consumer apps that enables apps to declare their flow policies in their manifest (and thus the policy is visible prior to an app's deployment). FlowFence ensures that the IoT app is restricted to its stated flow policy.

Our work focuses on tailoring FlowFence to IoT domains because they are still emerging, giving us the opportunity to build a flow control primitive directly into application structure. Flow-based protections could, in principle, be applied to other domains, but challenges are often domain-specific. This work solves IoT-specific challenges. We discuss the applicability of Opacified Computation to other domains in §6.

Our Contributions:

- We conduct a study of three major existing IoT frameworks that span the domains of smart homes, and wearables (i.e. Samsung SmartThings, Google Fit, and Android Sensor API) to analyze IoT-specific challenges and security design issues, and to inform the functionality goals for an IoT application framework (§2).

- Based on our findings we design the Opacified Computation model, which enables robust and efficient source to sink flow control (§3).

- We realize the Opacified Computation model through the design of FlowFence for IoT platforms. Our prototype runs on a Nexus 4 with Android that acts as our "IoT Hub" (§4). FlowFence only requires process isolation and IPC services from the underlying OS, thus minimizing the requirements placed on the hardware/OS.

- We perform a thorough evaluation of FlowFence framework (§5). We find that each sandbox requires 2.7*MB* of memory on average. Average latency for calls to functions across a sandbox boundary in our tests was 92*ms* or less. To understand the impact of these overheads on end-to-end performance, we ported three existing IoT apps to FlowFence (§5.2). Adapting these apps to use FlowFence resulted in average size of apps going up from 232 lines to 332 lines of source code. A single developer with no prior knowledge of the FlowFence API took five days total to port all these apps. Macro-benchmarks on these apps (latency and throughput) indicate that FlowFence performance overhead is acceptable: we found a 4.9% increase in latency for an app that performs face recognition, and we found a negligible reduction in throughput for a wearable heart beat calculator app. In terms of security, we found that the flow policies correctly enforce flow control over these three apps (§5.2). Based on this evaluation, we find FlowFence to be a practical, secure, and efficient framework for IoT applications.

2 IoT Framework Study: Platforms and Threats

We performed an analysis of existing IoT application programming frameworks, apps, and their security models to inform FlowFence design, distill key functionality requirements, and discover security design shortcomings. Our study involved analyzing three popular programming frameworks covering three classes of IoT apps: (1) Samsung SmartThings for the smart home, (2) Google Fit for wearables, and (3) Android Sensor API for quantified-self apps.[1] We manually inspected API documentation, and mapped it to design patterns. We found that across the three frameworks, access to IoT sensor data falls in one of the following design patterns: (1) The *polling pattern* involving apps polling an IoT device's current state; and (2) The *callback pattern* involving apps registering callback functions that are invoked whenever an IoT device's state changes.[2]

We also found that it is desirable for publishers and consumers to operate in a device-agnostic way, without being explicitly connected to each other, *e.g.,* a heart rate monitor may go offline when a wearable is out of Bluetooth range; the consumer should not have to listen to lifecycle events of the heart rate monitor—it only needs the heart beat data whenever that is available. Ideally, the consumer should only need to specify the type of data it requires, and the IoT framework should provide this data, while abstracting away the details. Furthermore, this is desirable because there are many types of individual devices that ultimately provide the same kind of data, *e.g.,* there are many kinds of heart rate monitors eventually providing heart rate data.

A practical IoT programming framework should support the two data sharing patterns described above in a device-agnostic manner. In terms of security, we found that all three frameworks offer permission-based access control, but they do not provide any methods to control data use once apps gain access to a resource. We provide brief detail on each of these frameworks below.

1) Samsung SmartThings. SmartThings is a smart home app programming framework [4] with support for 132 device types ranging from wall plugs to ZWave door locks. SmartThings provides two types of APIs to access device data: `subscribe` and `poll`. The `subscribe` API is the callback design pattern. For instance, to obtain a ZWave door lock's current state, an app would issue a call of the form `subscribe(lockDevice, "lock.state", callback)`. The `subscribe` API abstracts away details of retrieving data from a device, and directly presents the data to consumers, allowing them to operate in a disconnected manner. The `poll` API is the polling pattern. For example, an app can invoke `lockDevice.currentState` to retrieve the state of the lock at that point in time.

For permission control, the end-user is prompted to authorize an app's access request to a device [57], based on a matching of *SmartThings capabilities* (a set of operations) that the app wishes to perform, and the set of capabilities that a device supports. Once an app is granted access to a device, it can access all of its data and features. SmartThings does not offer any data flow control primitives.

2) Google Fit. Google Fit enables apps to interface with wearables like smartwatches [32]. The core abstraction in Google Fit is the Fitness Data Type, which provides a

[1] Quantified Self refers to data acquisition and processing on aspects of a person's daily life, *e.g.,* calories consumed.

[2] We also found an orthogonal *virtual sensor* design pattern: An intermediate app computing on sensor data and re-publishing the derived data as a separate virtual sensor. For instance, an app reads in heart rate at beats-per-minute, derives beats-per-hour, and re-publishes this data as a separate sensor.

device-agnostic abstraction for apps to access them in either instantaneous or aggregated form. The API provides raw access to both data types using only the callback pattern; the polling pattern is not supported. For instance, to obtain expended calories, an app registers a data point listener for the `com.google.calories.expended` instantaneous fitness type. A noteworthy aspect is that apps using the Fit API can pre-process data and publish *secondary* data sources, essentially providing a virtual sensor.

Google Fit API defines *scopes* that govern access to fitness data. For instance, the `FITNESS_BODY_READ` scope controls access to heart rate. Apps must request read or write access to a particular scope, and the user must approve or deny the request. Once an app gains access to a scope, it can access all fitness related data in that scope. Google Fit does not offer any data flow control primitives.

3) Android Sensor API. Android provides API access to three categories of smartphone sensor data: Motion, Environment, and Position. Apps must register a class implementing the `SensorEventListener` interface to receive callbacks that provide realtime sensor state. There is no API to poll sensor state, except for the Location API. Android treats the Location API differently but, for our purposes, we consider it to be within the general umbrella of the sensor API. The Location API supports both the polling and callback design patterns. The callback pattern supports consumers operating in a device-agnostic manner since the consumer only specifies the type of data it is interested in.

Surprisingly, the Android sensor API does not provide any access control mechanism protecting sensor data. Any app can register a callback and receive sensor data. The Location API and heart rate sensor API, however, do use Android permissions [22, 31]. Similar to the previous two frameworks, Android does not offer any data flow control primitives.

IoT Architectures. We observe two categories of IoT software architectures: (1) Hub, and (2) Cloud. The hub model is centralized and executes the majority of software on a hub that exists in proximity to various physical devices, which connect to it. The hub has significantly more computational power than individual IoT devices, has access to a power supply, provides network connectivity to physical devices, and executes IoT apps. In contrast, a cloud architecture executes apps in remote servers and may use a minimal hub that only serves as a proxy for relaying commands to physical devices. The hub model is less prone to reliability issues, such as functionality degradation due to network connectivity losses that plague cloud architectures [58]. Furthermore, we observe a general trend toward adoption of the hub model by industry in systems such as Android Auto [1] and Wear [2], Samsung SmartThings [55][3], and Logitech Harmony [3]. Our work targets the popular hub model, making it widely applicable to these hub-based IoT systems.

Threat Model. IoT apps are exposed to a slew of sensitive data from sensors, devices connected to the hub, and other hub-based apps. This opens up the possibility of sensitive data leaks leading to privacy invasion. For instance, Denning *et al.* outlined emergent threats to smart homes, including misuse of sensitive data for extortion and for blackmail [17]. Fernandes *et al.* recently demonstrated that such threats exist in real apps on an existing IoT platform [26] where they were able to steal and misuse door lock pincodes.

We assume that the adversary controls IoT apps running on a hub whose platform software is trusted. The adversary can program the apps to attempt to leak sensitive data. Our security goal is to force apps to declare their intended data use patterns, and then enforce those flows, while preventing all other flows. This enables the design of more privacy-respecting apps. For instance, if an app on FlowFence declares it will sink camera data to a door lock, then the system will ensure that the app cannot leak that data to the Internet. We assume that side channels and covert channels are outside the scope of this work. We discuss implications of side channels, and possible defense strategies in §6.

3 Opacified Computation Model

Consider the example smart home app from §1, where it unlocks the front door based on people's faces. It uses the bitmap to extract features, checks the current state of the door, unlocks the door, and sends a notification to the home owner using the Internet. This app uses sensitive camera data, and accesses the Internet for the notification (in addition to ads and crash reporting). An end user wishes to reap the benefits of such a scenario but also wants to ensure that the door control app does not leak camera data to the Internet.

FlowFence supports such scenarios through the use of Opacified Computation, which consists of two main components: (1) Quarantined Modules ("functions"), and (2) opaque handles. A Quarantined Module (QM) is a developer-written code module that computes on sensitive data (which is assigned a taint label at the data source), and runs in a system-provided sandbox. A developer is free to write many such Quarantined Modules. Therefore, each app on FlowFence is split into two parts: (1) some non-sensitive code that does not compute on sensitive data, and (2) a set of QMs that compute on sensitive data. Developers can chain multiple QMs together

[3] Recent v2 hubs have local processing.

to achieve useful work, with the unit of transfer between QMs being opaque handles—immutable, labeled opaque references to data that can only be dereferenced by QMs when running inside a sandbox. QMs and opaque handles are associated with a taint set, *i.e.,* a set of taint labels that indicates the provenance of data and helps track information flows (we explain label design later in this section).

An opaque handle does not reveal any information about the data value, data type, data size, taint set, or exceptions that may have occurred to non-sensitive code. Although such opaqueness can make debugging potentially difficult, our implementation does support a development-time debugging flag that lifts these opaqueness restrictions (§4).

Listings 1 and 2 shows pseudo-code of example smart home apps. The `CamPub` app defines `QM_bmp` that publishes the bitmap data. FlowFence ensures that whenever a QM returns to the caller, its results are converted to an opaque handle.

Line 10 of Listing 1 shows the publisher app calling the QM (a blocking call), supplying the function name and a taint label. FlowFence allocates a clean sandbox, and runs the QM. The result of `QM_bmp` running is the opaque handle `hCam`, which refers to the return data, and is associated with the taint label `Taint_CAMERA`. `hCam` is immutable—it will always refer to the data that was used while creating it (immutability helps us reduce overtainting; we discuss it later in this section). Line 11 shows `CamPub` sending the resultant handle to a consumer.

We also have a second publisher of data `QM_status` that publishes the door state (Line 16 of Listing 1), along with a door identifier, and provides an IPC function for consumers to call (Line 20).

The `DoorCon` app defines `QM_recog`, which expects a bitmap, and door state (Lines 6-9 of Listing 2). It computes feature vectors from the bitmap, checks if the face is authorized, checks the door state, and unlocks the door. Lines 18, 19 of Listing 2 show this consumer app receiving opaque handles from the publishers. As discussed, non-sensitive code only sees opaque handles. In this case, `hCam` refers to camera-tainted data, and `hStatus` refers to door-state-tainted data, but the consumer app cannot read the data unless it passes the data to a QM. Moreover, for this same reason, non-sensitive code cannot test the value of a handle to create an implicit flow.

Line 20 calls a QM, passing the handles as parameters. FlowFence automatically and transparently dereferences opaque handle arguments into raw data before invoking a QM. Transparent dereferencing of opaque handles offers developers the ability to write QMs normally with standard types even though some parameters may be passed as opaque handles. During this process, FlowFence allocates a clean sandbox for the QM to run, and propagates the taint labels of the opaque handles to that sandbox. Finally, `QM_recog` receives the raw data and opens the door.

The consumer app uses `QM_report` to send out the state of the door to a remote monitoring website. It also attempts to use `QM_mal` to leak the bitmap data. FlowFence prevents such a leak by enforcing flow policies, which we discuss next.

Flow Policy. A publisher app, which is associated with a sensor (or sensors), can add taint labels to its data that are tuples of the form $(appID, name)$, where *appID* is the identifier of the publisher app and *name* is the name of the taint label. This name denotes a standardized type that publishers and consumers can agree upon, for example, `Taint_CAMERA`. We require labels to be statically declared in the app's manifest. *appID* is unique to an app and is used to avoid name collisions across apps.[4] Additionally, in its manifest, the publisher can specify a set of flow rules for each of its taint labels, with the set of flow rules constituting the publisher policy. The publisher policy defines the permissible flows that govern the publisher's data. A flow rule is of the form `TaintLabel` $\rightarrow$ `Sink`, where a sink can be a user interface, actuators, Internet, etc. `CamPub`'s flow policy is described on Line 3 of Listing 1. The policy states that consumer apps can sink camera data to the sink labeled UI (which is a standard label corresponding to a user's display at the hub).

Since other possible sinks for camera data are not necessarily known to the publisher, new flow policies are added as follows. A consumer app must request approval for flow policies if it wants to access sensitive data. Consumer flow policies can be more restrictive than publisher policies, supporting the least privilege principle. They can also request new flows, in which case the hub user must approve them. `DoorCon`'s policy requests are described in Lines 2-4 of Listing 2. It requests the flows: `Taint_CAMERA` $\rightarrow$ `Door.Open`, `Taint_DOORSTATE` $\rightarrow$ `Door.Open`, `Taint_DOORSTATE` $\rightarrow$ `Internet`. At app install time, a consumer app will be bound to a publisher that provides data sources with labels `Taint_CAMERA`, `Taint_DOORSTATE`.

To compute the final policy for a given consumer app FlowFence performs two steps. First, it computes the intersection between the publisher policy and the consumer policy flow rules. In our example, the intersection is the null set. If it were not null, FlowFence would authorize the intersecting flows for the consumer app in question. Second, it computes the set difference between the consumer policy and publisher policy. This difference reflects the flows the consumer has requested but the publisher policy has not covered. At this point, FlowFence

[4] An app cannot forge its ID since our implementation uses Android package name as the ID. See §4 for details.

delegates approval to the IoT hub owner to make the final decision about whether to approve the flows or not. If the hub owner decides to approve a flow that a publisher policy does not cover, that exception is added for subsequent runs of that consumer app. Such a approval does not apply to other apps that may also use the data.

If a QM were to attempt to declassify the camera data to the Internet (*e.g.,* `QM_mal`) directly without requesting a flow policy, the attempt would be denied as none of the flow policies allow it. An exception is thrown to the calling QM whenever it tries to perform an unauthorized declassification. Similar to exception processing in languages like Java, if a QM does not catch an exception, any output handle of this QM is moved into the exception state. Non-QM code cannot view this exception state. If an app uses such a handle in a subsequent QM as a parameter, then that QM will silently fail, with all of its output handles also in the exception state. App developers can avoid this by ensuring that a QM handles all possible exceptions before returning and, if necessary, encodes any errors into the return object, which can then be examined in a subsequent QM that receives the returned handle.

FlowFence is in a position to make security decisions because the publisher assigns taint labels while creating the handles, and when `DoorCon` reads in the handles, it results in the taint labels propagating to the sandbox running `QM_mal`. FlowFence simply reads the taint labels of the sandbox at the time of declassification.

All declassification of sensitive data can only occur through well-known trusted APIs that FlowFence defines. Although our prototype provides a fixed set of trusted APIs that execute in a separate trusted process, we envision a plug-in architecture that supports community built and vetted APIs (§4). FlowFence sets up sandbox isolation such that attempts at declassifying data using non-trusted APIs, such as arbitrary OS system calls, are denied.

Table 1 summarizes the taint logic. When a clean sandbox loads a QM, it has no taint. A taint label, belonging to the app, may be added to a handle at creation, or to a sandbox at any time, allowing data providers to label themselves as needed. A call from QM executing in S_0 to another QM that is launched in sandbox S_1 results in the taint labels of S_0 being copied to S_1. When a called QM returns, FlowFence copies the taint of the sandbox into the automatically created opaque handle. At that point, the QM no longer exists. The caller is not tainted by the returned handle, unless the caller (which must be a QM) dereferences the handle. These taint arithmetic rules, combined with QMs, opaque handles, and sandboxes conceptually correspond to a directed data flow graph from sources to sinks, as we illustrate with the example below.

```
1  application CamPub
2  taint_label Taint_CAMERA;
3  allow { Taint_CAMERA -> UI }
4
5  Bitmap QM_bmp():
6    Bitmap face = camDevice.snapshot();
7    return face;
8
9  if (motion at FrontDoor)
10   hCam = QM.call(QM_bmp, Taint_CAMERA);
11   send hCam to DoorCon;
12 -----------------------------------------
13 application DoorStatePub
14 taint_label Taint_DOORSTATE;
15
16 Status QM_status():
17   return (door[0].state(), 0); //state,idx
18
19 /* IPC */ Handle getDoorState():
20   return QM.call(QM_status,
       Taint_DOORSTATE);
```

Listing 1: Pseudocode for two publishers—camera data, and door state. Quarantined Modules are shown in light gray.

```
1  application DoorCon
2  request { Taint_CAMERA -> Door.Open,
3            Taint_DOORSTATE -> Door.Open,
4            Taint_DOORSTATE -> Internet }
5
6  void QM_recog(faceBmp, status):
7    Features f = extractFeatures(faceBmp);
8    if(status != unlocked AND isAuth(f))
9      TrustedAPI.door[0].open();
10
11 void QM_report(status):
12   TrustedAPI.network.send(status);
13
14 void QM_mal(faceBmp):
15   /* this is denied */
16   TrustedAPI.network.send(faceBmp);
17
18 receive hCam from CamPub;
19 Handle hStatus =
     DoorStatePub.getDoorState();
20 QM.call(QM_recog, hCam, hStatus);
21 QM.call(QM_mal, hCam);
22 QM.call(QM_report, hStatus);
```

Listing 2: Consumer app pseudocode that reads camera and door state data, and controls a door. Quarantined Modules are shown in light gray.

FlowFence Data Flow Graph. We now discuss the taint flow logic of FlowFence in more detail, and show how it creates and tracks, at runtime, a directed data flow graph that enables it to make security decisions on flows. Figure 1 shows two publishers of sensitive data that generate $OH_{T_1}(d_1)$—an opaque handle that refers to camera bitmap data d_1, and $OH_{T_2}(d_2)$—an opaque handle that refers to door state data d_2, using QM_{bmp} and QM_{status}

Operation	Taint Action
Sandbox S loads a QM	$T[S] := \varnothing$
QM inside S reads opaque handle $d = OH^{-1}(h)$	$T[S] += T[h]$
QM inside S returns $h = OH(d)$	$T[h] := T[S]$
QM manually adds taints $\{t\}$ to its sandbox	$T[S] += \{t\}$
QM_0 inside S_0 calls QM_1 inside S_1	$T[S_1] = T[S_0]$

Table 1: Taint Arithmetic in FlowFence. $T[S]$ denotes taint labels of a sandbox running a QM. $T[h]$ denotes taint label of a handle h.

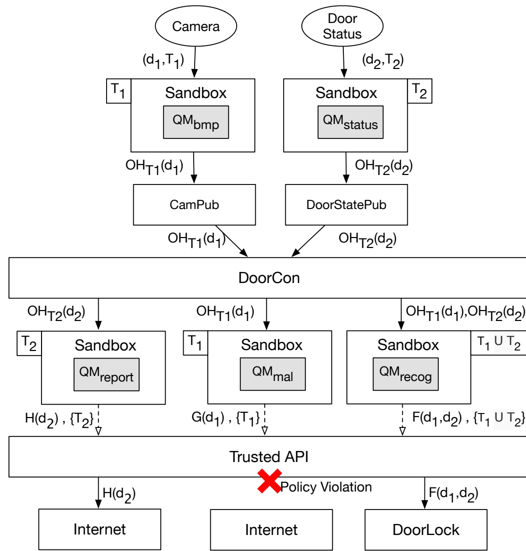

Figure 1: Data flow graph for our face recognition example. FlowFence tracks taint labels as they propagate from sources, to handles, to QMs, to sinks. The dotted lines represent a declassification attempt. The trusted API uses labels on the sandboxes to match a flow policy.

respectively. T_1 and T_2 are taint labels for data d_1 and d_2. The user wants to ensure that camera data does not flow to the internet.

The consumer app (DoorCon) consists of non-sensitive code that reads the above opaque handles from the publishers, and invokes three QMs. QM_{recog} operates on both $OH_{T_1}(d_1)$ and $OH_{T_2}(d_2)$. When the non-sensitive code requests execution of QM_{recog}, FlowFence will allocate a clean sandbox, dereference the handles into raw values, and invoke the module. The sandbox inherits the taint label $T_1 \cup T_2$. Later on, when QM_{recog} tries to declassify its results by invoking the trusted API, FlowFence will read the taint labels (dotted line in Figure 1)—$T_1 \cup T_2$. That is, FlowFence taint arithmetic defines that the taint label of the result is the combination of input data taint labels. In our example, declassifying camera and door state tainted data to the door lock is permitted, since the user authorized the flow earlier.

If the consumer app tries to declassify sensitive data d_1 by invoking a trusted API using QM_{mal}, the API reads the taint labels on the handle being declassified, determines that there is no policy that allows $d_1 \rightarrow$ `Internet`, and denies the declassification.

Immutable opaque handles are key to realizing this directed data flow graph. Consider Figure 1. If handles were mutable, and if QM_{mal} read in some data with taint label T_3, then we would have to assume that $OH_{T_1}(d_1)$ is tainted with T_3, leading to overtainting. Later on, when QM_{recog} executes, its sandbox would inherit the taint label T_3 due to the overtainting. If there was a policy that prevented T_3 from flowing to the door lock, FlowFence would prevent QM_{recog} from executing the declassification. FlowFence avoids these overtainting issues by having immutable handles, which enable better precision when reasoning about flows. There are other sources of overtainting related to how a programmer structures the computation and IoT-specific mechanisms that FlowFence introduces. We discuss their implications and how to manage them in §4 and §6.

As discussed above, taint flows transitively from data sources, to opaque handles, to sandboxes, back to opaque handles, and eventually to sinks via the trusted API, where FlowFence can enforce security policies. This design allows taint flow to be observed in a black-box manner, simply by tracking the inputs and outputs. This allows QMs to internally use any language, without the overhead of native taint tracking, only by using sandbox processes to enforce isolation as described in §4.

FlowFence Security Guarantees. FlowFence uses its taint arithmetic rules to maintain the invariant that the taint set of a QM executing in a sandbox at any time represents the union of the taints of sensitive data used by the QM through opaque handles or through calls from another QM. Furthermore, FlowFence avoids propagating taint on QM returns with the help of opaque handles. Since these handles are opaque outside a QM, non-sensitive code must pass them into QMs to dereference them, allowing FlowFence to track taints. If the non-sensitive code of a consumer app transmits an opaque handle to another app via an OS-provided IPC mechanism, FlowFence still tracks that flow since the receiving app also has to use a QM to make use of the handle.

To prevent flow policy violations, a sandbox must be designed such that writes from a QM to a sink go through a trusted API that enforces specified flow policies. We discuss how we achieve this sandbox design in §4.

4 FlowFence Architecture

FlowFence supports executing untrusted IoT apps using two major components (Figure 2): (1) A series of *sandboxes* that execute untrusted, app-provided QMs in an isolated environment that prevents unwanted communication, and (2) A *Trusted Service* that maintains handles and the data they represent; converting data to opaque handles and dereferencing opaque handles back; mediating data flow between sources, QMs, and sinks, including taint propagation and policy enforcement; and creating, destroying, scheduling, and managing lifetime of sandboxes.

We discuss the design of these components in the context of an IoT hub with Android OS running on top. We selected Android because of the availability of source code. Google's recently announced IoT-specific OS—Brillo [29], is also an Android variant.[5] Furthermore, with the introduction of Google Weave [30], we expect to see Android apps adding IoT capabilities in the future.

Untrusted IoT Apps & QMs. Developers write apps for FlowFence in Java and can optionally load native code into QMs. As shown in Figure 2, each app consists of code that does not use sensitive data inputs, and a set of QMs that use sensitive data inputs. Although abstractly, QMs are functions, we designed them as methods operating on serializable objects. Each method takes some number of parameters, each of which can either be (1) raw, serialized data, or (2) opaque handles returned from previous method calls on this or another QM. A developer can write a method to return raw data, but returning raw data would allow leakage. Thus, FlowFence converts that raw data to an opaque handle prior to returning to the untrusted app.[6]

Trusted Service & APIs. This service manages all sensitive data flowing to and from QMs that are executing in sandboxes. It schedules QMs for execution inside sandboxes, dereferencing any opaque handle parameters, and assigning the appropriate taint labels to the sandboxes. The Trusted Service also ensures that a sandbox is correctly tainted whenever a QM reads in sensitive data (Tainter component of Figure 2), as per the taint arithmetic rules in FlowFence (Table 1). Once it taints a sandbox, the Trusted Service maintains the current taint labels securely in its process memory.

FlowFence does not track or update taints for variables inside a QM. Instead, it treats a QM as a blackbox for the

[5]Brillo OS is only a limited release at the time of writing. Therefore, we selected the more mature codebase for design, since core services are the same on Android and Brillo.

[6]A QM can theoretically leak sensitive data through side channels (*e.g.,* by varying the execution time of the method prior to returning). We assume side channels to be out of scope of our system and thus we do not address them in our current threat model. If such leaks were to be a concern, we discuss potential defense strategies in §6.

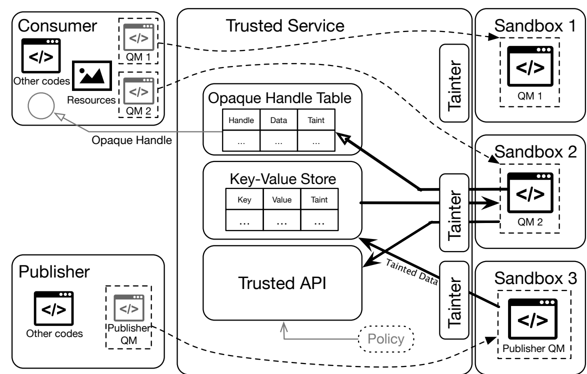

Figure 2: FlowFence Architecture. Developers split apps into Quarantined Modules, that run in sandbox processes. Data leaving a sandbox is converted to an opaque handle tainted with the sandbox taint set.

purpose of taint analysis and it only needs to examine sensitive inputs being accessed or handles provided to a method as inputs. We expect QMs to be limited to the subset of code that actually processes sensitive data, with non-sensitive code running without change. Although this does reduce performance overhead and avoids implicit flow leaks by forcing apps to only use controlled and well-defined data transfer mechanisms, it does require programmers to properly split their app into least-privilege QMs, which if done incorrectly, could lead to overtainting.

When a QM Q running inside a sandbox S returns, the Trusted Service creates a new opaque handle h corresponding to the return data d, and then creates an entry $< h, < d, T[S] >>$ in its opaque handle Table (Figure 2), and returns h to the caller.

The Trusted Service provides APIs for QMs allowing them to access various sinks. Our current prototype has well-known APIs for access to network, ZWave switches, ZWave locks, camera streams, camera pictures, and location. As an example of bridging FlowFence with such cyber-physical devices, we built an API for Samsung SmartThings. This API makes remote calls to a web services SmartThings app that proxies device commands from FlowFence to devices like ZWave locks. The Trusted API also serves as a policy enforcement point, and makes decisions whether to allow or deny flows based on the specific policy set for the consumer app.

We envision a plug-in architecture that enables community-built and vetted Trusted APIs to integrate with our framework. The plugin API should ideally be in a separate address space. The Trusted Service will send already declassified data to this plugin API via secure IPC. This limits risk by separating the handle table from external code.

Sandboxes. The Trusted Service uses operating system support to create sandbox processes that FlowFence uses to execute QMs. When a QM arrives for execution, FlowFence reserves a sandbox for exclusive use by that QM, until execution completes. Once a QM finishes executing, FlowFence sanitizes sandboxes to prevent data leaks. It does this by destroying and recreating the process.

For efficiency reasons, the Trusted Service maintains a pool of clean spare sandboxes, and will sanitize idle sandboxes in the background to keep that pool full. In addition, the Trusted Service can reassign sandboxes without needing to sanitize them, if the starting taint (based on the input parameters) of the new QM is a superset of or equal to the ending taint of the previous occupant of that sandbox. This is true in many common cases, including passing the return value of one QM directly into another QM. In practice, sandbox restarts only happen on a small minority of calls.

FlowFence creates the sandboxes with the `isolatedProcess` flag set, which causes Android to activate a combination of restrictive user IDs, IPC limitations, and strict SELinux policies. These restrictions have the net effect of preventing the isolated process from communicating with the outside world, except via an IPC interface connected to the Trusted Service.

As shown in Figure 2, this IPC interface belongs to the Trusted API discussed earlier. When the sandboxes communicate with the Trusted Service over an IPC interface, the IPC request is matched to the sandbox it originated from as well as to the QM that initiated the call. As discussed, the Trusted Service maintains information about each sandbox, including its taint labels and running QM, in a lookup table in its own memory, safely out of reach of, possibly malicious, QMs.

Debugging. Code outside QMs cannot dereference opaque handles to inspect corresponding data or exceptions, complicating debugging during development. To alleviate this, FlowFence supports a development time debugging option that allows code outside a QM to dereference handles and inspect their data and any exception traces. However, a deployment of FlowFence has this debugging flag removed. Also, as discussed previously, use of a opaque handle in exception state as a parameter to a QM results in the QM returning a new opaque handle that is also in the exception state. Providing a mechanism for exception handling in the called QM without increasing programmer burden is challenging and a work-in-progress. Currently, we use the idiom of a QM handling all exceptions it can and encoding any error as part of the returned value. This allows any subsequent QM that is called with the handle as a parameter to examine the value and handle the error.

Key-Value Store. This is one of the primary data-sharing mechanisms in FlowFence between publishers of tainted sensitive data and consumer apps that use the data. This design was inspired by our framework study in §2, and it supports publishers and consumers operating in a device-agnostic manner, with consumers only having to know the type of data (taint label) they are interested in processing. Each app receives its own KV store (Figure 2) into which it can update the value associated with a key by storing a $< key, sensitive_value, taint_label >$ while executing a QM. For instance, a camera image publisher may create a key such as `CAM_BITMAP`, with an image byte array as the value, and a taint label `Taint_CAMERA` to denote the type of published data (declared in the app manifest). A key is public information—non-sensitive code outside a QM must create a key before it can write a corresponding value. This ensures that a publisher cannot use creation of keys as a signaling mechanism. An app on FlowFence can only write to its own KV store. Taints propagate as usual when a consumer app keys from the KV store. Finally, the publishing QM associated with a sensor usually would not read other sensitive information sources, and thus would not have any additional taint. In the case this QM has read other sources of information, then the existing taint is applied to any published data automatically.

If a QM reads a key's value, the value's taint label will be added to that QM's sandbox. All key accesses are pass-by-value, and any subsequent change in a value's taint label does not affect the taint labels of QMs that accessed that value in prior executions. Consider an example value V with taint label T_1. Assuming a QM Q_1 accessed this value, it would inherit the taint. Later on, if the publisher changes the taint label of V to $T_1 \cup T_2$, this would not affect the taint label of Q_1, until it reads V again.

The polling design pattern is easy to implement using a Key-Value Store. A consumer app's QM can periodically access the value of a given key until it finds a new value or a non-null value. Publicly accessible keys simplify making sensitive data available to third-party apps, subject to flow policies.

Event Channels. This is the second data-sharing mechanism in FlowFence; it supports the design pattern of registering callbacks for IoT device state changes (*e.g.,* new data being available). The channel mechanism supports all primitive and serializable data types. An app creates channels statically by declaring them in a manifest file at development time (non-sensitive code outside QMs could also create it), making it the owner for all declared channels. Once an app is installed, its channels are available for use—there are no operations to explicitly open or close channels. Other app's QMs can then register to such channels for updates. When a channel-owner's

QM puts data on the channel, FlowFence invokes all registered QMs with that data as a parameter. FlowFence automatically assigns the current set of taint labels of the channel-owner to any data it puts on the channel, so that all QMs that receive the callback will be automatically tainted correctly. If a QM is executed as a callback for a channel update, it does not return any data to the non-sensitive code of the app.

Although the publishers and consumers can share opaque handles using OS-provided sharing mechanisms, we designed the Key-Value store, and Event channels explicitly so that publishers and consumers can operate in a device-agnostic manner by specifying the types of data they are interested in, ignoring lower level details.

As described here, both inter-app communication mechanisms, the KV store and event channels, can potentially lead to poison-pill attacks [37] where a compromised or malicious publisher adds arbitrary taint labels, with the goal of overtainting consumers and preventing them from writing to sinks. See the discussion of overtainting in §6 for a defense strategy.

FlowFence Policies and User Experience. In our prototype, users install the app binary package with associated policies. FlowFence prompts users to approve consumer flow policies that are not covered by publisher policies at install time. This install-time prompting behavior is similar to the existing Android model. FlowFence models its flow request UI after the existing Android runtime permission request screens, in an effort to remain close to existing permission-granting paradigms and to leverage existing user training with permission screens. However, unlike Android, FlowFence users are requested to authorize flows rather than permissions, ensuring their control over how apps use data. If a user approves a set of flows, FlowFence guarantees that only those flows can occur.

Past work has shown that users often do not comprehend or ignore prompts [25], however, existing research does point out interesting directions for future work in improving such systems. Felt *et al.* discuss techniques to better design prompting mechanisms [23], and Roesner *et al.* discuss contextual prompting [50, 51] as possible improvements.

5 Evaluation

We evaluated FlowFence from multiple perspectives. First, we ran a series of microbenchmarks to study call latency, serialization overhead, and memory overhead of FlowFence. We found that FlowFence adds modest computational and memory costs. Running a sandbox takes $2.7MB$ RAM on average, and running multiple such sandboxes will fit easily within current hardware for IoT hubs.[7] We observed a $92ms$ QM call latency with 4 spare sandboxes, which is comparable to the latency of common network calls in IoT apps. FlowFence supports a maximum bandwidth of $31.5MB/s$ for transferring data into sandboxes, which is large enough to accommodate typical IoT apps. Second, we ported three IoT apps to FlowFence to examine developer effort, security, and impact of FlowFence on macro-performance factors. Our results show that developers can use FlowFence with modest changes to their apps and with acceptable performance impact, making FlowFence practical for building secure IoT apps. Porting the three apps required adding 99 lines of code on average per app. We observed a 4.9% latency increase to perform face recognition in a door controller app. More details follow.

5.1 Microbenchmarks

We performed our microbenchmarks on an LG Nexus 4 running FlowFence on Android 5.0. The Nexus 4 serves as our "IoT hub" that runs QMs and enforces flow policies. In our experiments, we evaluated three factors that can affect apps running on FlowFence.

Memory overhead. We evaluated memory overhead of FlowFence using the `MemoryInfo` API. We ran FlowFence with $0-15$ empty sandboxes and recorded the memory consumption. Our results show that the FlowFence core requires $6.35MB$ of memory while each sandbox requires $2.7MB$ of memory on average. To put this in perspective, LG Nexus 4 has 2GB memory and loading a blank page on the Chrome browser on it used $98MB$ of memory, while loading FlowFence with 16 sandboxes used $49.5MB$. Therefore, we argue that the memory overhead of FlowFence is within acceptable limits for the platform.

QM Call Latency. We measured QM call latency for non-tainted and tainted parameters (30 trials each with 100 QM call-reply sequences) to assess performance in scenarios that allowed reuse of a sandbox without sanitizing and those that required sanitizing. For tainted calls, each QM takes a single boolean parameter that is tainted. We also varied the number of clean spare sandboxes that are available for immediate QM scheduling initially before each trial. Regardless of the number of spare sandboxes, untainted calls (which did not taint the sandboxes and thus could reuse them without sanitizing) showed a consistent latency of $2.1ms$ (SD=$0.4ms$). The tainted calls were made so as to always require a previously-tainted sandbox to be sanitized. Figure 3 shows average latency of tainted calls across 30 trials for different number of spare sandboxes. As the number of spare sandboxes increases from 0 to 4, the average call

[7] For example, Samsung SmartThings hub has $512MB$ RAM [56], and Apple TV hub has $1GB$ RAM [7].

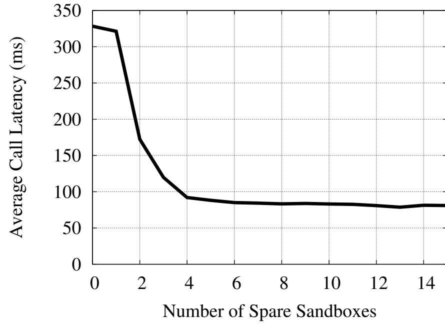

Figure 3: QM Call latency of FlowFence given various number of spare sandboxes, for calls that require previously-used sandboxes to be sanitized before a call. Calls that can reuse sandboxes without sanitizing (untainted calls in our tests) show a consistent latency of $2.1ms$, which is not shown in this graph.

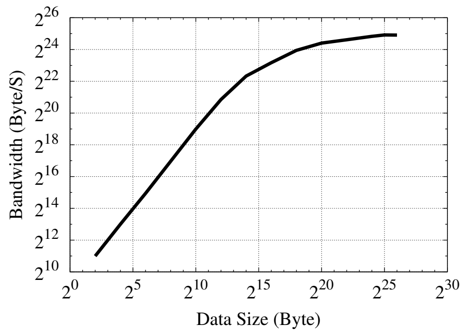

Figure 4: Serialization bandwidth for different data sizes. Bandwidth caps off at $31.5MB/s$.

latency decreases from $328ms$ to $92ms$. Further increase in the number of spare sandboxes does not improve latency of QM calls. At 4 spares, the call latency is less than $100ms$, making it comparable to latencies seen in controlling many IoT devices (*e.g.,* Nest, SmartThings locks) over a wide-area network. This makes QMs especially suitable to run existing IoT apps that already accept latencies in this range.

Serialization Overhead. To understand FlowFence overhead for non-trivial data types, we computed serialization bandwidth for calls on QMs that cross sandbox boundaries with varying parameter sizes. Figure 4 presents the results for data ranging from $4B$ to $16MB$. The bandwidth increases as data size increases and caps off at $31.5MB/s$. This is large enough to support typical IoT apps—for example, the Nest camera uses a maximum bandwidth of $1.2Mbps$ under high activity [33]. A single camera frame used by one of our ported apps (see below), is $37kB$, requiring transferring data at $820kB/s$ to a QM.

5.2 Ported IoT Applications

We ported three existing IoT apps to FlowFence to measure its impact on security, developer effort, end-to-end latency, and throughput on operations relevant to the apps (Table 2). SmartLights is a common smart home app (*e.g.,* available in SmartThings) that computes a predicate based on a location value from a beacon such as a smartphone, or car [47]. If the location value inside the home's geofence, the app turns on lights (and adjusts other devices like thermostats) around the home. When the location value is outside the home's geofence, the app takes the reverse action.

FaceDoor performs face recognition and unlocks a door, if a detected face is authorized [34]. The app uses the camera to take an image of a person at the door, and runs the Qualcomm face recognition SDK (chipset-specific native code, available only as a binary).

HeartRateMonitor accesses a camera to compute heart rate using photoplethysmography [67]. The app uses image processing code on streamed camera frames.

FlowFence provides trusted API to access switches, locks, and camera frames. These three existing apps cover the popular IoT areas of smart homes and quantified self. Furthermore, face recognition and camera-frame-streaming apps are among the more computationally expensive types of IoT apps, and stress test FlowFence performance. We ran all our experiments on Android 5.0 (Nexus 4).

Security. We discuss data security risks that each of the three IoT apps pose when run on existing platforms, and find that FlowFence eliminates those risks successfully under leakage tests.

1) SmartLights: It has the potential to leak location information to attackers via the Internet. The app has Internet access for ads, and crash reporting. On FlowFence, the developer separates code that computes on location in a QM which isolates the flow: `loc` $\rightarrow$ `switch`, while allowing other code to use the Internet freely.

2) FaceDoor: This app can leak camera data to the Internet. We note that this app requires Internet access for core functionality—it sends a notification to the user whenever the door state changes. Therefore, under current IoT frameworks it is very easy for this app to leak camera data. FlowFence isolates the flow of camera and door state data to door locks from the flow of door state data to the Internet using two QMs, eliminating any possibility of cross-flows between the camera and the Internet. This app uses the flows: `cam` $\rightarrow$ `lock`, `doorstate` $\rightarrow$ `lock`, `doorstate` $\rightarrow$ `Internet`.

3) HeartRateMonitor: The app can leak images of people, plus heart rate information derived from the camera stream. However, similar to previous apps, the developer of this app too will use FlowFence support to isolate the

Name	Description	Data Security Risk without FlowFence	LoC original	LoC FlowFence	Flow Request
SmartLights [47]	Reads a location beacon and if the beacon is inside a geofence around the home, automatically turn on the lights	App can leak user location information	118	193	`loc` → `switch`
FaceDoor [34]	Uses a camera to recognize a face; If the face is authorized, unlock a doorlock	App can leak images of people	322	456	`cam` → `lock`, `doorstate` → `lock`, `doorstate` → `net`
HeartRateMonitor [67]	Uses a camera to measure heart rate and display on UI	App can leak images of people, and heart rate information	257	346	`cam` → `ui`

Table 2: Features of the three IoT apps ported to FlowFence. Implementing FlowFence adds 99 lines of code on average to each app (less than 140 lines per app).

flow: `cam` → `ui` into a QM. We note that in all apps, the QMs can return opaque handles to the pieces of code not dealing with sensitive information, where the handle can be leaked, but this is of no value to the attacker since a handle is not sensitive data.

Developer Effort. Porting apps to FlowFence requires converting pieces of code operating on sensitive data to QMs. On average, 99 lines of code were added to each app (Table 2). We note that typical IoT apps today are relatively small in size compared to, say, Android apps. The average size across 499 apps for which we have source code for SmartThings platform is 162 line of source code. Most are event-driven, receiving data from various publishers that they are authorized to at install time and then publish to various sinks, including devices or Internet. Much of the extra code deals with resolving the appropriate QMs, and creating services to communicate with FlowFence. It took a developer with no prior knowledge of the FlowFence API to port the first two apps in two 8-hour (approx.) days each, and the last app in a single day. We envision that with appropriate developer tool support, many boiler plate tasks, including QM resolution, can be automated. We note that the increase in LoC is not co-related to the original LoC of the app. Instead, there is an increase in LoC only for pieces of the original app that deals with sensitive data. Furthermore, it is our experience that refactoring an existing app requires copying logic as-is, and building QMs around it. For instance, we did not have source-code access to the Qualcomm Face Recognition SDK, but we were able to successfully port the app to FlowFence.

Porting FaceDoor. Here, we give an example of the steps involved in porting an app. First, we removed all code from the app related to camera access, because FlowFence provides a camera API that allows QMs to take pictures, and access the corresponding bitmaps. Next, we split out face recognition operations into its own Quarantined Module—QM_{recog}, that loads the native code face recognition SDK. We modified QM_{recog} to use the Trusted API to access a camera image, an operation that causes it to be tainted with camera data. We modified the pieces of code related to manipulating a ZWave lock to instead use FlowFence-provided API for accessing door locks. We also created QM_{report} that reads the door state source and then sends a notification to the user using the Internet. These two QMs isolate the flow from camera and door state to door lock, and the flow from door state to the Internet, effectively preventing any privacy violating flow of camera data to the Internet, which would otherwise be possible with current IoT frameworks.

End-to-End Latency. We quantified the impact of FlowFence on latency for various operations in the apps that are crucial to their functionality. We measured latency as the time it takes for an app to perform one entire computational cycle. In the case of SmartLights, one cycle is the time when the beacon reports a location value, till the time the app issues an operation to manipulate a switch. We observed a latency of 160*ms* (SD=69.9) for SmartLights in the baseline case, and a latency of 270*ms* (SD=96.1) in the FlowFence case. The reason for increased latency is due to QM load time, and cross-process transfers of the location predicate value.

FaceDoor has two operations where latency matters. First, the enroll latency is the time it takes the app to extract features from a provided bitmap of a person's face. Second, recognition latency is the time it takes the app to match a given bitmap of a person's face to an item in the app's database of features. We used images of our team members (6), measuring 612x816 pixels with an average

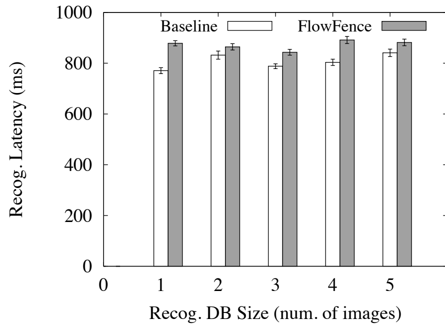

Figure 5: FaceDoor Recognition Latency (ms) on varying DB sizes for Baseline and FlowFence. Using FlowFence causes 5% increase in average latency.

HeartRateMonitor Metric (fps)	Baseline Avg (SD)	FlowFence Avg (SD)
Throughput with no Image Processing	23.0 (0.7)	22.9 (0.7)
Throughput with Image Processing	22.9 (0.7)	22.7 (0.7)

Table 3: Throughput for HeartRateMonitor on Baseline (Stock Android) and FlowFence. FlowFence imposes little overhead on the app.

size of 290.3kB (SD=15.2).

We observed an enroll latency of 811ms (SD=37.1) in the baseline case, and 937ms (SD=60.4) for FlowFence, averaged over 50 trials. The increase in latency (15.5%) is due to QM load time, and marshaling costs for transferring bitmaps over process boundaries. While the increase in latency is well within bounds of network variations, and undetectable by user in both previous cases; it is important to recognize that most of this increase is resulted from setup time and the effect on actual processing time is much more modest. Figure 5 shows latency for face recognition, averaged over 10 trials, for Baseline, and FlowFence. We varied the recognition database size from 1 to 5 images. In each test, the last image enrolled in the database is a specific person's face that we designated as the test face. While invoking the recognition operation, we used another image of the same test person's face. We observe a modest, and expected increase in latency when FaceDoor runs on FlowFence. For instance, it took 882ms to successfully recognize a face in a DB of 5 images and unlock the door on FlowFence, compared to 841ms on baseline—a 4.9% increase. This latency is smaller than 100ms and thus small enough to not cause user-noticeable delays in unlocking a door once a face is recognized [13].

Throughput. Table 3 summarizes the throughput in frames per second (fps) for HeartRateMonitor. We observed a throughput of $23.0 fps$ on Stock Android for an app that read frames at maximum rate from a camera over a period of 120 seconds. We repeated the same experiment with the image processing load of heart rate detection, and observed no change in throughput. These results matched our expectations, given that the additional serialization and call latency is too low to impact the throughput of reading from the camera (camera was the bottleneck). Thus, we observed no change in the app's abilities to derive heart rate.

6 Discussion and Limitations

Overtainting. Overtainting is difficult to avoid in taint propagation systems. FlowFence limits overtainting in two ways: (1) by not propagating taint labels from a QM to its caller—an opaque handle returned as a result of a call to a QM has an associated taint but does not cause the caller to become tainted (unless the caller is a QM that dereferences the handle), limiting the taints to QMs; and (2) a QM (and associated sandbox) is ephemeral. Since FlowFence sanitizes sandboxes if a new occupant's taints differ from the previous occupant, reusing sandboxes does not cause overtainting. Nevertheless, FlowFence does not prevent overtainting due to poor application decomposition into QMs.

A malicious publisher can potentially overtaint a consumer by publishing overtainted data that the consumer subscribes to, leading to poison-pill attacks [37]. A plausible defense strategy is to allow a consumer to inspect an item's taint and not proceed with a read if the item is overtainted [63]. However, this risks introducing a signaling mechanism from a high producer to a low consumer via changes to the item's taint set. To address the attack in the context of our system. We first observe that most publishers will publish their sensor data under a known, fixed taint. The key idea is to simply require publishers to define a *taint bound* TM_c, whenever a channel c is created.[8] If the publisher writes data with a taint set T that is not a subset of TM_c to the channel c, the write operation is denied and results in an exception; else the write is allowed. The consumer, to avoid getting overtainted, can inspect this channel's taint bound (but not the item's taint) before deciding to read an item from the channel. The taint bound cannot be modified, once defined, avoiding the signaling problem. A similar defense mechanism was proposed in label-based IFC systems [63, 62].

Applicability of Opacified Computation to other domains. In this work we only discussed Opacified Com-

[8] Same idea applies when creating keys, with a taint bound defined at that time for any future value associated with the key.

putation in the context of IoT frameworks (*e.g.,* FlowFence Key-Value Store and Event Channels are inspired by our IoT framework study). The basic Opacified Computation model is broadly applicable. For example, there is nothing fundamental preventing our hub from being a mobile smartphone and the app running on it being a mobile app. But, applying FlowFence to existing mobile apps is challenging because of the need to refactor apps and the libraries they use (many of the libraries access sensitive data as well as sinks). As another design point, there is no fundamental limitation that requires IoT hub software to run in a user's home; it could well be cloud-hosted and provided as a trusted cloud-based service for supporting computations on sensitive data. Use of a cloud-based service for executing apps is not unusual—SmartThings runs all apps on its cloud, using a hub to primarily serve as a gateway for connecting devices to the cloud-based apps.

Usability of Flow Prompts. FlowFence suffers from the same limitation as all systems where users need to make security decisions, in that we cannot prevent users from approving flows that they should not. FlowFence does offer additional information during prompts since it presents flow requests with sources and sinks indicating *how* the app intends to use data, possibly leading to more informed decision-making. Flow prompts to request user permissions could be avoided if publisher policies always overrode consumer policies, with no user override allowed. But that just shifts the burden to specifying publisher policies correctly, which still may require user involvement. User education on flow policies and further user studies are likely going to be required to examine usability of flow prompts. In some IoT environments, the right to configure policies or grant overrides could be assigned to specially-trained administrators who manage flow policies on behalf of users and install apps and devices for them.

Measuring flows. Almuhimedi *et al.* performed a user study that suggests that providing metrics on frequency of use of a previously granted permission can nudge users to patch their privacy policy [6]. For example, if a user is told that an app read their location 5,398 times over a day, they may be more inclined to prevent that app from getting full access to the location. Adding support for measuring flows (both permitted and denied) to assist users in evaluating past flow permissions is part of future work.

Side Channels. A limitation of our current design is that attackers can encode sensitive data values in the time it takes for QMs to return. Such side channel techniques are primarily applicable to leaking low-bandwidth data. Nevertheless, we are investigating techniques to restrict this particular channel by making QMs return immediately, and have them execute asynchronously, thus eliminating the availability of fine-grain timing information in the opaque handles (as in LIO [61]). This would involve creating opaque handle dependency graphs that determine how to schedule QMs for later execution. Furthermore, timing channel leakages can be bounded using predictive techniques [72].

7 Related Work

IoT Security. Current research focuses around analyzing the security of devices [35, 27], protocols [44, 11], or platforms [26, 12]. For example, Fernandes *et al.* showed how malicious apps can steal pincodes [26]. Current IoT frameworks only offer access control but not data-flow control primitives (§2). In contrast, our work introduces, to the best of our knowledge, the first security model targeted at controlling data flows in IoT apps.

Permission Models. We observe that IoT framework permissions are modeled after smartphone permissions. There has been a large research effort at analyzing, and improving access control in smartphone frameworks [20, 49, 22, 24, 51, 50, 10, 16, 43, 68]. For instance, Enck *et al.* introduced the idea that dangerous permission combinations are indicative of possibly malicious activity [20]. Roesner *et al.* introduced User-Driven Access control where apps prompt for permissions only when they need it [51, 50]. However, permissions are fundamentally only gate-keepers. The *PlaceRaider* sensory malware abuses granted permissions and uses smartphone sensors (*e.g.,* camera) to reconstruct the 3D environment of the user for reconnaissance [64]. This malware exploits the inability of permission systems to control data usage once access in granted. The IoT fundamentally has a lot more sensitive data than a single smartphone camera, motivating the need for a security model that is capable of strictly controlling data use once apps obtain access. PiBox does offer privacy guarantees using differential-privacy algorithms after apps gain permissions, but it is primarily applicable to apps that gather aggregate statistics [43]. In contrast, FlowFence controls data flows between arbitrary types of publishers and consumers.

Label-based Information Flow Control. FlowFence builds on substantial prior work on information flow control that use labeling architectures [52, 42, 18, 71, 63, 28, 62, 41, 15, 38, 46]. For example, Flume [42] enforces flow control at the level of processes while retaining existing OS abstractions, Hails [28] presents a web framework that uses MAC to confine untrusted web apps, and COWL [63] introduces labeled compartments for JavaScript code in web apps. Although FlowFence is closely related to such systems, it also makes design choices tailored to meet the needs specific to the IoT domain. In terms of similarities, FlowFence shares the design principles of making information flow explicit, con-

trolling information flow at a higher granularity than the instruction-level, and supporting declassification. However, these systems only support producer (source) defined policies whereas FlowFence supports policies defined by both producing and consuming apps. This feature allows for more versatility in environments such as IoT, where a variety of consuming apps could request for a diverse set of flows. Our evaluation shows hows such a mix of flow policies supports real IoT apps (§5.2).

Computation on Opacified Data. Jana *et al.* built the recognizer OS abstraction and Darkly [39, 40]—systems that enable apps to compute on perceptual data while protecting the user's privacy. These systems also use opaque handles, but they only support trusted functions operating on the raw data that handles refer to. In contrast, FlowFence supports untrusted third-party functions executing over raw data while providing flow control guarantees. Furthermore, these systems leverage characteristics of the data they are trying to protect to achieve security guarantees. For example, Darkly depends on camera streams being amenable to privacy transforms, allowing it to substitute low-fidelity data for high-fidelity data, and it depends on apps being able to tolerate the differences. However, in the general case, neither IoT data nor their apps may be amenable to such transforms. FlowFence is explicitly designed to support computation over sensitive IoT data in the general case.

Taint Tracking. Taint tracking systems [69, 19] are popular techniques for enforcing flow control that monitor data flows through programs [60]. Beyond performance issues [48], such techniques suffer from an inability to effectively handle implicit flows, and concurrency [59]. Although there are techniques to reduce computational burden [54, 65], they often require specialized hardware, not necessarily available in IoT environments. These techniques are also difficult to apply to situations where taint labels are not known *a priori* (*e.g.,* manage tainted data that is generated by apps, rather than known sources). Compared to these techniques, FlowFence adds little performance overhead. Furthermore, FlowFence does not require specialized hardware, and does not suffer from implicit flow attacks.

Static Analysis. Another class of systems such as FlowDroid [9], and Amandroid [66] use static taint tracking to enforce flow control. While these techniques do not suffer from performance issues associated with dynamic systems, they still suffer from same shortcomings associated with concurrency and implicit flows [9]. Besides static analysis techniques, there are also language-based techniques, such as JFlow [45], that require the developer to learn and use a single security-typed language. In contrast, FlowFence supports building apps using unmodified existing languages and development tools, enabling developers to quickly port their apps.

8 Conclusions

Emerging IoT programming frameworks only support permission based access control on sensitive data, making it possible for malicious apps to abuse permissions and leak data. In this work, we introduce the Opacified Computation model, and its concrete instantiation, FlowFence, which requires consumers of sensitive data to explicitly declare intended data flows. It enforces the declared flows and prevents all other flows, including implicit flows, efficiently. To achieve this, FlowFence requires developers to split their apps into: (1) A set of communicating Quarantined Modules with the unit of communication being opaque handles—taint tracked, opaque references to data that can only be dereferenced inside sandboxes; (2) Non-sensitive code that does not compute on sensitive data, but it still orchestrates execution of Quarantined Modules that compute on sensitive data. We ported three IoT apps to FlowFence, each requiring less than 140 additional lines of code. Latency and throughput measurements of crucial operations of the ported apps indicate that FlowFence adds little overhead. For instance, we observed a 4.9% latency increase to recognize a face in a door controller app.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Deian Stefan, for their insightful feedback on our work. We thank Kevin Borders, Kevin Eykholt, and Jaeyeon Jung for providing feedback on earlier drafts. This research is supported in part by NSF grant CNS-1318722 and by a generous gift from General Motors. Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). His work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), the Italian MIUR-PRIN TENACE Project (agreement 20103P34XC), and by the projects "Tackling Mobile Malware with Innovative Machine Learning Techniques," "Physical-Layer Security for Wireless Communication," and "Content Centric Networking: Security and Privacy Issues" funded by the University of Padua. Any opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Android auto. `https://www.android.com/auto/`. Accessed: May 2016.

[2] Android wear. `https://www.android.com/wear/`. Accessed: May 2016.

[3] Logitech harmony hub. http://www.logitech.com/en-us/product/harmony-hub. Accessed: May 2016.

[4] Samsung SmartThings Home Automation. http://www.smartthings.com/. Accessed: Oct 2015.

[5] Vera Smart Home Controller. http://getvera.com/controllers/vera3/. Accessed: Oct 2015.

[6] ALMUHIMEDI, H., SCHAUB, F., SADEH, N., ADJERID, I., ACQUISTI, A., GLUCK, J., CRANOR, L. F., AND AGARWAL, Y. Your Location Has Been Shared 5,398 Times!: A Field Study on Mobile App Privacy Nudging. In *ACM Conference on Human Factors in Computing Systems (CHI)* (2015).

[7] APPLE. Apple TV Memory Specifications. https://developer.apple.com/library/tvos/documentation/General/Conceptual/AppleTV_PG/index.html#//apple_ref/doc/uid/TP40015241-CH12-SW1. Accessed: June 2016.

[8] APPLE. HomeKit. http://www.apple.com/ios/homekit/. Accessed: Oct 2015.

[9] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM symposium on Programming Language Design and Implementation (PLDI)* (2014).

[10] BACKES, M., BUGIEL, S., AND GERLING, S. Scippa: System-centric ipc provenance on android. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014).

[11] BEHRANG FOULADI AND SAHAND GHANOUN. Honey, I'm Home!!, Hacking ZWave Home Automation Systems. Black Hat USA, 2013.

[12] BUSOLD, C., HEUSER, S., RIOS, J., SADEGHI, A.-R., AND ASOKAN, N. Smart and secure cross-device apps for the internet of advanced things. In *Financial Cryptography and Data Security (FC)* (2015).

[13] CARD, S. K., ROBERTSON, G. G., AND MACKINLAY, J. D. The information visualizer, an information workspace. In *SIGCHI Conference on Human factors in computing systems* (1991).

[14] CARLINI, N., FELT, A. P., AND WAGNER, D. An evaluation of the google chrome extension security architecture. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (2012).

[15] CHENG, W., PORTS, D. R., SCHULTZ, D., POPIC, V., BLANKSTEIN, A., COWLING, J., CURTIS, D., SHRIRA, L., AND LISKOV, B. Abstractions for usable information flow control in aeolus. In *USENIX ATC* (2012).

[16] CONTI, M., CRISPO, B., FERNANDES, E., AND ZHAUNIAROVICH, Y. Crêpe: A system for enforcing fine-grained context-related policies on android. *TIFS* (2012).

[17] DENNING, T., KOHNO, T., AND LEVY, H. M. Computer security and the modern home. *Communications of ACM* (2013).

[18] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *SOSP* (2005).

[19] 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 *OSDI* (2010).

[20] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2009).

[21] ERNST, M. D., JUST, R., MILLSTEIN, S., DIETL, W., PERNSTEINER, S., ROESNER, F., KOSCHER, K., BARROS, P. B., BHORASKAR, R., HAN, S., VINES, P., AND WU, E. X. Collaborative verification of information flow for a high-assurance app store. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2014).

[22] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2011).

[23] FELT, A. P., EGELMAN, S., FINIFTER, M., AKHAWE, D., AND WAGNER, D. How to ask for permission. In *USENIX Conference on Hot Topics in Security (HotSec)* (2012).

[24] FELT, A. P., EGELMAN, S., AND WAGNER, D. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (2012).

[25] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (2012), Symposium On Usable Privacy and Security (SOUPS).

[26] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security analysis of emerging smart home applications. In *IEEE Symposium on Security and Privacy (S&P)* (2016).

[27] FISHER, D. Pair of Bugs Open Honeywell Home Controllers Up to Easy Hacks. https://threatpost.com/pair-of-bugs-open-honeywell-home-controllers-up-to-easy-hacks/113965/. Accessed: Oct 2015.

[28] 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 *OSDI* (2012).

[29] GOOGLE. Project Brillo. https://developers.google.com/brillo/. Accessed: Oct 2015.

[30] GOOGLE. Project Weave. https://developers.google.com/weave/. Accessed: Oct 2015.

[31] GOOGLE ANDROID. Requesting Permissions at Runtime. http://developer.android.com/training/permissions/requesting.html. Accessed: Feb 2016.

[32] GOOGLE DEVELOPERS. Google Fit Developer Documentation. https://developers.google.com/fit/. Accessed: Feb 2016.

[33] GOOGLE NEST. How much bandwidth will Nest cam use? https://nest.com/support/article/How-much-bandwidth-will-Nest-Cam-use. Accessed: June 2016.

[34] HACHMAN, M. Want to unlock your door with your face? Windows 10 for IoT Core promises to do just that. http://www.pcworld.com/article/2962330/internet-of-things/want-to-unlock-your-door-with-your-face-windows-10-for-iot-core-promises-to-do-just-that.html. Accessed: Feb 2016.

[35] HESSELDAHL, A. A Hacker's-Eye View of the Internet of Things. http://recode.net/2015/04/07/a-hackers-eye-view-of-the-internet-of-things/. Accessed: Oct 2015.

[36] HEULE, S., RIFKIN, D., RUSSO, A., AND STEFAN, D. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015), USENIX Association.

[37] HRITCU, C., GREENBERG, M., KAREL, B., PIERCE, B. C., AND MORRISETT, G. All your ifcexception are belong to us. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), SP '13.

[38] HRITCU, C., GREENBERG, M., KAREL, B., PIERCE, B. C., AND MORRISETT, G. All your ifcexception are belong to us. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE.

[39] JANA, S., MOLNAR, D., MOSHCHUK, A., DUNN, A., LIVSHITS, B., WANG, H. J., AND OFEK, E. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security Symposium* (2013).

[40] JANA, S., NARAYANAN, A., AND SHMATIKOV, V. A Scanner Darkly: Protecting User Privacy from Perceptual Applications. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[41] JIA, L., ALJURAIDAN, J., FRAGKAKI, E., BAUER, L., STROUCKEN, M., FUKUSHIMA, K., KIYOMOTO, S., AND MIYAKE, Y. Run-time enforcement of information-flow properties on android. In *European Symposium on Research in Computer Security* (2013).

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

[43] LEE, S., WONG, E. L., GOEL, D., DAHLIN, M., AND SHMATIKOV, V. box: A platform for privacy-preserving apps. In *NSDI* (2013).

[44] LOMAS, N. Critical Flaw identified In ZigBee Smart Home Devices. `http://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/`. Accessed: Oct 2015.

[45] MYERS, A. C. Jflow: Practical mostly-static information flow control. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (1999).

[46] NADKARNI, A., AND ENCK, W. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM.

[47] PANSARASA, J. Lights-After-Dark SmartThings App. `https://github.com/jpansarasa/SmartThings/blob/master/smartapps/elasticdev/lights-after-dark.src/lights-after-dark.groovy`. Accessed: Feb 2016.

[48] PAUPORE, J., FERNANDES, E., PRAKASH, A., ROY, S., AND OU, X. Practical always-on taint tracking on mobile devices. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (2015).

[49] RAHMATI, A., AND MADHYASTHA, H. V. Context-specific access control: Conforming permissions with user expectations. In *ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)* (2015).

[50] ROESNER, F., AND KOHNO, T. Securing embedded user interfaces: Android and beyond. In *USENIX Security Symposium* (2013).

[51] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE S&P* (2012).

[52] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical fine-grained decentralized information flow control. In *PLDI* (2009).

[53] RUSSELLO, G., CONTI, M., CRISPO, B., AND FERNANDES, E. Moses: Supporting operation modes on smartphones. In *ACM Symposium on Access Control Models and Technologies (SACMAT)* (2012).

[54] RUWASE, O., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., CHEN, S., KOZUCH, M., AND RYAN, M. Parallelizing dynamic information flow tracking. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures* (2008).

[55] SAMSUNG. SmartThings. `http://www.smartthings.com/`. Accessed: Nov 2015.

[56] SAMSUNG SMARTTHINGS. Samsung SmartThings Memory Specifications. `https://community.smartthings.com/t/the-next-generation-of-smartthings-is-here/21521`. Accessed: June 2016.

[57] SAMSUNG SMARTTHINGS. SmartThings Capabilities Reference. `http://docs.smartthings.com/en/latest/capabilities-reference.html`. Accessed: Feb 2016.

[58] SAMSUNG SMARTTHINGS. What happens if the power goes out or I lose my internet connection? `https://support.smartthings.com/hc/en-us/articles/205956960-What-happens-if-the-power-goes-out-or-I-lose-my-internet-connection-`. Accessed: May 2016.

[59] SARWAR, G., MEHANI, O., BORELI, R., AND KAAFAR, M. A. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *International Conference on Security and Cryptography (SECRYPT)* (2013).

[60] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)* (2010).

[61] STEFAN, D., RUSSO, A., BUIRAS, P., LEVY, A., MITCHELL, J. C., AND MAZIÉRES, D. Addressing covert termination and timing channels in concurrent information flow systems. In *ACM SIGPLAN Notices* (2012).

[62] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible dynamic information flow control in Haskell. In *Haskell Symposium* (September 2011), ACM SIGPLAN.

[63] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting users by confining javascript with cowl. In *OSDI* (2014).

[64] TEMPLEMAN, R., RAHMAN, Z., CRANDALL, D., AND KAPADIA, A. PlaceRaider: Virtual theft in physical spaces with smartphones. In *ISOC Network and Distributed System Security Symposium (NDSS)* (2013).

[65] VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., AND AUGUST, D. I. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on* (2004).

[66] WEI, F., ROY, S., OU, X., AND ROBBY. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2014).

[67] WETHERELL, J. Android Heart Rate Monitor App. `https://github.com/phishman3579/android-heart-rate-monitor`. Accessed: Feb 2016.

[68] XU, Y., HUNT, T., KWON, Y., GEORGIEV, M., SHMATIKOV, V., AND WITCHEL, E. Earp: Principled storage, sharing, and protection for mobile apps. In *NSDI* (2016).

[69] XU, Y., AND WITCHEL, E. Maxoid: Transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM.

[70] YOON, M.-K., SALAJEGHEH, N., CHEN, Y., AND CHRISTODORESCU, M. Pift: Predictive information flow tracking. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems* (2016).

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

[72] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Predictive mitigation of timing channels in interactive systems. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2011).

[73] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *IEEE S&P* (2012).

Appendix A: FlowFence API

We summarize the object-oriented FlowFence API for developers in Table 4. There are two kinds of API: QM-management, and Within-QM. Developers use the QM-management API to request loading QMs into sandboxes, making QM calls, and receiving opaque handles as return values. The primary data types are: *QM <T>*, and *Handle*. The former data type represents a reference to a loaded QM. The latter data type represents an opaque handle, that FlowFence creates as a return value of a QM. Developers use `resolveCtor`, or `resolveM` to load a specific QM into a sandbox (FlowFence automatically manages sandboxes), and receive a reference to the loaded QM. Then, developers specify the string name of a QM method to execute.

The Within-QM API is available to QMs while they are executing within a sandbox. Currently, FlowFence has two data types available for QMs. *KVStore* offers ways to get and put values in the Key-Value store. The Trusted API offers facilities like network communication, logging, and smart home control (our prototype has a bridge to SmartThings).

QM-management Data Types and API	**Semantics**
Handle	An opaque handle. Data is stored in the Trusted Service, with its taint labels.
QM <T>	A reference to a QM of type T, on which developers can issue method calls.
QM <T> ctor = **resolveCtor**(T)	Resolve the constructor for QM T, and return a reference to it.
QM <T> m = **resolveM**(retType, T, methStr, [paramTypes])	Resolve an instance/static method of a QM, loading the QM into a sandbox if necessary.
Handle ret = *QM <T>*.**call**([argList])	Call a method on a loaded QM, and return an opaque handle as the result.
subscribeEventChannel(appID, channelName, *QM <T>*)	Subscribe to a channel for updates, and register a QM to be executed automatically whenever new data is placed on the channel.
Within-QM Data Types and API	**Semantics**
KVStore	Provides methods to interact with the Key-Value Store.
KVStore kvs = **getKVStore**(appID, name)	Get a reference to a named KVStore.
kvs.**put**<T>(key, value, taint_label)	Put a (key, value) pair into the KVStore along with a taint label, where T can be a basic type such as Int, Float, or a serializable type. Any existing taint of the calling QM will be automatically associated with the value's final set of taint labels.
T value = kvs.**get**<T>(key)	Get the value of type T corresponding to specified key, and taint the QM with the appropriate set of taint labels.
getTrustedAPI(apiName).**invoke**([params])	Call a Trusted API method to declassify sensitive data.
getChannel(chanName).**fireEvent**(taint_label, [params])	Fire an event with parameters, specifying taint label. Any existing taint labels of the calling QM will be added automatically.

Table 4: FlowFence API Summary. QM-management data types and API is only available to the untrusted portion of an app that does not operate with sensitive data. The Within-QM data types and API is available only to QMs.