# Securing IoT Apps with Fine-grained Control of Information Flows

**Davino Mauro Junior**[1], **Kiev Gama**[1], **Atul Prakash**[2]

[1]Centro de Informática (CIn) – Universidade Federal de Pernambuco (UFPE)
Recife, PE, Brazil

[2]Department of Electrical Engineering and Computer Science – University of Michigan
Ann Arbor, Michigan, U.S.

{dmtsj,kiev}@cin.ufpe.br, aprakash@umich.edu}

***Abstract.*** *Internet of Things is growing rapidly, with many connected devices now available to consumers. With this growth, the IoT apps that manage the devices from smartphones raise significant security concerns. Typically, these apps are secured via sensitive credentials such as email and password that need to be validated through specific servers, thus requiring permissions to access the Internet. Unfortunately, even when developers of these apps are well-intentioned, such apps can be non-trivial to secure so as to guarantee that user's credentials do not leak to unauthorized servers on the Internet. For example, if the app relies on third-party libraries, as many do, those libraries can potentially capture and leak sensitive credentials. Bugs in the applications can also result in exploitable vulnerabilities that leak credentials. This paper presents our work in-progress on a prototype that enables developers to control how information flows within the app from sensitive UI data to specific servers. We extend FlowFence to enforce fine-grained information flow policies on sensitive UI data.*

## 1. Introduction

The Internet of Things (IoT) is growing rapidly, with 8 billion of "things" connected in 2017, and 20 billion expected by 2020. Among these, the consumer segment is the largest, with 63% of installed devices in 2017, in contrast with business-specific domains [Gartner 2017]. Security has been a key concern, with research focusing on both hardware (smart devices) and software (IoT frameworks/platforms) to avoid events like the Mirai Botnet attack, which affected 100,000 IoT devices around the world [Guardian 2016].

Mobile applications targeting IoT devices are commonly developed using IoT frameworks, which provide a set of guiding protocols and standards that simplify the implementation of IoT applications. They enable developers to build apps that compute on data emitted by IoT devices (e.g., camera, heart-rate, and temperature sensors). IoT devices typically rely on cloud services to allow users to monitor and control the devices remotely from their smartphones, requiring users to authenticate to the cloud services or, in some cases, to the device and, under typical permission models used for apps, to grant the app full access to the Internet.

This permission model is too permissive from a security standpoint as it controls *what* sources and sinks the app can access, but not *how* information flows between sources and sinks. For example, a permission which grants the permission to send camera data

(source) to the network (sink) can leak data to arbitrary malicious servers, as the permission does not state which flows of this kind are allowed. In the case of sensitive user-interface (UI) data, such as userid/passwords for cloud services or devices, existing platforms do not even provide a mechanism to tag the UI data as sensitive and thus the sensitive UI data can be arbitrarily leaked to any authorized sink.

Fernandes et al. [Fernandes et al. 2016b] proposed in prior work FlowFence, a framework based on taint analysis which forces an app developer to declare–through flow policies–the intended flows of sources and sinks allowed for that app. These policies are dynamically enforced by FlowFence, which checks whether a flow from source to sink is allowed. While FlowFence enables developers to secure data flows between sources and sinks, it did not provide (1) ability to declare User-Interface (UI) fields as sensitive sources and (2) ability to constrain the network addresses to be used as sinks.

To illustrate the importance of these features, consider, for example, a sensitive UI field in the app, say a password field, that is used to authenticate into a legitimate server. Unfortunately, FlowFence cannot prevent an unwanted flow of this field to a *NETWORK* sink. This is because (1) the password field could not be declared as a sensitive source; and (2) even if the password field was declared as sensitive source, FlowFence would allow the flow anyway, as it cannot enforce flow policies to *NETWORK* sinks in a fine-grained manner, i.e., a declared flow of *SOURCE->NETWORK* permit a flow of *SOURCE* to every server on the *NETWORK*, not just some specific server.

The fundamental problem is that current solutions cannot control information flows between sources and sinks in a fine-grained manner. Also, they do not treat UI values as sensitive sources, as to avoid leaking these values to unwanted sinks (e.g., *SMS, NETWORK*).

The main contributions of this work are: (1) we enable developers to tag UI fields holding sensitive data as sensitive sources, enforcing flow policies upon those sources to sinks; (2) we provide mechanisms for declaring flow policies that secure network requests to custom endpoints; and (3) though our work was motivated by (in)security of IoT apps, our prototype is Android-based and broadly applicable for helping developers protect sensitive credentials in other Android apps as well.

## 2. Motivating Example

To better understand the importance of securing UI values and network requests in apps, consider the following scenario. Figure 1 illustrates a screen that may be displayed when a user wishes to control an IoT device remotely. After looking at 32 consumer best-selling IoT brands including Nest, Ring, August Home and TP-Link, we found that all of them presented a login screen. The user is prompted to enter his credentials and authenticate to the device manufacturer's cloud server so he can access IoT devices that were registered on the user's account. These credentials usually include an email and password, consisting of sensitive data that, if leaked, could compromise the IoT device being accessed.

On Android, the app needs permission to access *NETWORK*, as the login process involves validating the user credentials to a server. This permission is too coarse-grained, as the data could be sent to any untrusted server if the app was compromised.

Figure 2 shows pseudocode of an IoT app illustrating this scenario. Line 2 shows

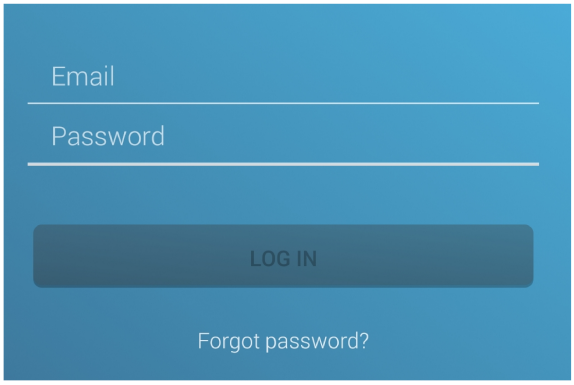


**Figure 1. A commonly seen UI screen in IoT apps. User enter his credentials so he can login and access his IoT device.**

a reference to default Android UI components (`EditText`) used to get the email and password values they hold. For that, one needs to call the `getText` method on both components (lines 3 and 4). Then, the values are used as parameters of another method call that continues the login process (line 5). The process of getting UI values is pretty simple, and as such, could be exploited. For example, taking advantage of the Android UI components, an adversary could inject malicious code that replicates line 3 and 4, but instead of sending the credentials to a cloud server, leaks the data to an untrusted server.

```
1: application SmartApp
2: EditText emailUI, passwordUI;
3: email = emailUI.getText();
4: password = passwordUI.getText();
5: login(email, password);
```

**Figure 2. Pseudocode of a Smart App using default UI components.**

## 3. Related Work

Inherited from the smartphone domain, current IoT frameworks use a permission-based model to restrict which resources mobile apps can access. Previous studies demonstrated how this model is inadequate due to its coarse-grained control of application permissions. Analyzing the permission system of the Android platform, Feng et al. presented several issues and attacks that took advantage of the permission-based model, describing over-claim of permissions as the most severe threat to Android security [Fang et al. 2014]. A study conducted by Felt et al. investigated the requested permissions of 940 apps in the Android platform, demonstrating that 94% of the analyzed apps had 4 or more extra permissions that were not used by the app [Felt et al. 2011].

Previous studies demonstrated how third-party libraries were being used as attack vectors on mobile apps, taking advantage of the app broad's access permissions [Sun and Tan 2014, Backes et al. 2016]. For instance, Backes et al. conducted a study on the top-downloaded Android apps of Google Play, concluding that 296 of these apps, which had a cumulative install base of 3.7 billion devices, were using advertisement third-party libraries with well-known security vulnerabilities [Backes et al. 2016].

The fundamental problem of the permission model resides on information flows not being controlled once permission to access different sources of data are granted on the apps. To mitigate this problem, previous studies used information flow techniques to identify potential threats. Some of those techniques used static and dynamic analysis to tackle the problem [Gordon et al. 2015, Enck et al. 2014, Arzt et al. 2014, Bell and Kaiser 2014]. For instance, Gordon et al. presented DroidSafe, a static analyzer that used taint tracking to identify malicious flows between sources and sinks upon installation of the app. Albeit effective, techniques like DroidSafe have problems dealing with dynamic code injection and implicit flows [Gordon et al. 2015]. Other techniques proposed were based on flow policies to control information flow on the program [Viet et al. 2010, Fernandes et al. 2016b].

Building on the idea of using flow policies to control data flows within the app, Fernandes et al. presented FlowFence, a framework that enables the development of secure apps under the Android platform [Fernandes et al. 2016b]. Its main concept relies on developers making the app's use patterns explicit through flow policies. These policies are then enforced by FlowFence to control flows of sensitive sources to sinks within the app [Fernandes et al. 2016b]. Albeit proving to be an effective solution, FlowFence did not cover UI as sensitive sources. Also, it did not provide mechanisms to control network requests in a fine-grained manner. In the following, we briefly discuss FlowFence's design in Section 4 and how we extend FlowFence to cover such scenarios in Section 5.

## 4. Background of FlowFence

FlowFence is a framework built for mobile development that enables definition and enforcement of information flow policies under the Android platform. Designed with the IoT architecture in mind, Fernandes et al. introduced a new information flow model referred as *Opacified Computation*. This model enables developers to tag sensitive data with taint labels within the app structure, declaring information flow policies bound to these labels. The rationale is that, once sensitive data is tagged, any computation using this data must run in a sandbox, which are separated processes managed by FlowFence. On these sandboxes, taints are automatically tracked following the defined flow policies. We briefly describe the architecture of FlowFence and how its main components are used for enforcement of information flow policies below.

### 4.1. FlowFence Architecture

FlowFence consists of two major components: (1) Developer-written functions (Quarantine Modules) that operate on sensitive data inside FlowFence-created *sandboxes* and (2) A Service (*Trusted Service*) that manages these *sandboxes* and mediate data flows between sources and sinks, also enforcing statically defined policies and providing APIs for accessing sensitive sinks, e.g., *NETWORK, SMS*.

**Quarantine Modules.** Quarantine Modules (QM) consist of developer-written functions that operate on sensitive data and execute only in FlowFence-created *sandboxes*, i.e., processes created and managed by the *Trusted Service*. These QM functions take serializable data as parameters and return *opaque handles*, which are references to sensitive data that can only be declassified by other QMs or via trusted service.

**Trusted Service/API.** A service responsible for: (1) creation and management of *sandboxes*; (2) control of all data flows between QMs tagged by taint labels, enforcing

policy rules linked to these labels; and (3) providing APIs to sensitive sinks that can only be used within QMs.

**Flow Policies.** Flow policies are declared in the app manifest. For every sensitive data, a taint label is defined in the form *(appID, name)*, where *appID* is the unique identifier of the app (represented by package name in the FlowFence implementation) and the *name* being the taint label itself. Within the declared label, a sensitive source of data is defined with its intended flows in the form *TaintLabel->Sink*. For example, one could define a taint label *(com.package.camera_app, TaintCamera)* and declare its intended flows, e.g., *TaintCamera->Network*.

## 5. Our Approach

One of the key challenges of developing secure apps is that there is no official security guidelines to obtain UI data from the user. Developers often need to send this data through the network, so securing data flows between sensitive UI sources and network sinks is crucial. Broadly speaking, our approach is developer-driven in the sense that it enables developers to secure data flows involving sensitive UI fields and network requests against dynamic code execution, e.g., malicious third-party libraries. To describe our approach, we use the scenario presented in Section 2 throughout the rest of this section.

### 5.1. Sensitive UI.

To secure sensitive UI data, we extend FlowFence so that UI components can be defined as sensitive sources. For that, we developed new UI components (SensitiveUI) that extend the Android SDK ones such that default behavior remains intact, but accessing the value of the components can only be done in a secure manner. Consider an Android UI component like `EditText`, for example. To *get* or *set* its value programmatically, one needs to call the `getText/setText` method in a reference object to that component. As the created SensitiveUI components extend the Android-based ones, they include both methods. However, accessing these methods is considered sensitive code and needs to be executed within a FlowFence QM. For instance, trying to access the `getText` without using FlowFence would return an empty value.

Inspired by FlowFence, we use a Key-Value mechanism to store UI values. Each SensitiveUI component has its value associated with an ID and taint label in the form (`<id, sensitive_value, taint_label>`), with `taint_label` being declared in a developer-written QM and serving the purpose of tagging the sensitive data to enforce information flow policies.

Figure 3 shows pseudo-code of the example app, only this time using the SensitiveUI component instead of the default Android one. Because we build on FlowFence, we need to split sensitive code into QMs as described in Section 4. Lines 4-6 show a QM that calls the Trusted API to obtain UI values. FlowFence's infrastructure ensures that whenever a QM is called, the return value is converted to an opaque handle. The value that this opaque handle holds can only be accessed by declassifying the handle within a FlowFence-created sandbox, thus running in a secure environment. Line 8 shows another QM responsible for the login process, receiving as parameter opaque handles holding a reference to email and password values. Lines 11 and 12 show how the UI values are recovered, with the QM being called. Finally, line 13 shows both email and password

opaque handles being used as parameters to the QM responsible for the login. Notice that calling the `getText` method upon any SensitiveUI reference is pointless (line 14), as the value can only be accessed within a QM. By using this component, the developer can specify which flows are permitted using the sensitive UI values as sources with flow policies. FlowFence would then proceed to enforce these policies while blocking all undeclared flows that use sensitive UI values.

```
 1: application SmartApp
 2: SensitiveUI emailUI, passwordUI;
 3:
 4: String QM_getUIValue(id):
 5:         value = TrustedAPI.sensitiveUI.getText(id);
 6:         return value;
 7:
 8: void QM_login(email, password):
 9:         // Continue the log in process
10:
11: email = QM.call (QM_getUIValue, emailUI.id);
12: password = QM.call (QM_getUIValue, passwordUI.id);
13: QM.call (QM_login, email, password);
14: password = passwordUI.getText(); // This call returns an empty value
```

**Figure 3. Pseudo-code of a Smart App using secure UI components in a login screen.**

### 5.2. Secure Network Requests.

To make secure network requests possible using FlowFence infrastructure, we start by extending the Trusted API so it can execute network requests to custom URLs. For that, we extend FlowFence's flow policy language to allow filtering of custom URLs through the definition of flow policies in a fine-grained manner, which we discuss next.

With FlowFence original implementation, the developer could only specify a flow of *SOURCE* to *NETWORK*, therefore permitting flows to any URL. Now, the developer can also specify which URL it wants the source data to sink to, declaring the rule in the form (*TaintLabel -> NETWORK, URL*). FlowFence policy checker would then compute the rule, also checking the URL of the request, either granting or denying the flow.

Figure 4 shows pseudocode of the example app described earlier. Line 3 shows the definition of a policy that specifies a flow between UI sources and network sink, but only to a specific cloud server's URL. Line 11 shows how the Trusted API is used to make a network request responsible for logging in. After obtaining opaque handles referencing the user's credentials (line 13 and 14), the QM responsible for login is called passing the credentials, server's URL, and taint label as parameters. With the Trusted API, the URL and taint labels are not considered when checking which network flows are permitted on the app. Line 16 shows how an unauthorized request to an untrusted server would be denied, as FlowFence policy checker would validate the requested URL to the declared flow policy.

Let's reconsider the scenario in Section 2 and show how the above mechanisms help reduce the attack surface available to untrusted third-party libraries. Malicious code that could be executed either by third-party libraries or through dynamic code injection

```
 1: application SmartApp
 2: taint_label Taint_UI;
 3: allow { Taint_UI -> NETWORK, http://appcloudserver.com }
 4: SensitiveUI emailUI, passwordUI;
 5:
 6: String QM_getUIValue(id):
 7:         value = TrustedAPI.sensitiveUI.getText(id);
 8:         return value;
 9:
10: void QM_login(email, password, url):
11:         TrustedAPI.network.post(email, password, url);
12:
13: email = QM.call (QM_getUIValue, emailUI.id);
14: password = QM.call (QM_getUIValue, passwordUI.id);
15: QM.call (QM_login, email, password, http://appcloudserver.com, Taint_UI);
16: QM.call (QM_login, email, password, http://untrustedserver.com, Taint_UI); // This request would be
    denied, as there is no policy specifying a flow between UI source and this URL as sink.
```

**Figure 4. Pseudo-code of a Smart App using secure UI components to send data through the network.**

outside a QM would not gain read access to sensitive UI data in the FlowFence-protected sandbox due to standard FlowFence mechanisms. Even if injected code manages to execute inside a QM, it would not be able to leak intercepted credentials to arbitrary servers due to fine-grain network policy. A limitation of our work that remains is that we do not prevent phishing attacks that use a fake UI screen generated from outside a QM by injected code. We plan to address the limitation in our future work. A possible approach is to extend the mechanisms for protecting against UI deception described in [Fernandes et al. 2016a, Bianchi et al. 2015] to assist the user in distinguishing the sensitive UI fields that are generated from within a QM and those that are not.

## 6. Conclusions

Controlling how data flows within IoT apps in a fine-grained manner is crucial to avoid data leakage. In this work, we presented our work in progress on a solution for the limitations of the permission-based model which are often used in current IoT and smartphone frameworks. We addressed how developers can secure sensitive sources of UI and control network flows in a fine-grained manner. As future work, we envision the evaluation of our approach with developers, quantifying both effort to port IoT apps as well as performance impact. We also plan to develop further ideas for controlling information flows within mobile apps.

## 7. Acknowledgements

This reserach is supported by NSF under grants No. 1740897 and 1740916, and by RNP under grant No. 002951. The authors thank the anonymous reviewers and also Luís Melo and Harvey Lu as well as professors Darko Marinov and Marcelo d'Amorim for their valuable feedback.

## References

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive

and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA. ACM.

Backes, M., Bugiel, S., and Derr, E. (2016). Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 356–367, New York, NY, USA. ACM.

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

Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., and Vigna, G. (2015). What the app is that? deception and countermeasures in the android user interface. 2015:931–948.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29.

Fang, Z., Han, W., and Li, Y. (2014). Permission based android security: Issues and countermeasures. *Computers and Security*, 43:205 – 218.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011). Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA. ACM.

Fernandes, E., Chen, Q. A., Paupore, J., Essl, G., Halderman, J. A., Mao, Z. M., and Prakash, A. (2016a). Android UI deception revisited: Attacks and defenses. In *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*, pages 41–59.

Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., and Prakash, A. (2016b). Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 531–548, Austin, TX. USENIX Association.

Gartner (2017). Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016.

Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. (2015). Information flow analysis of android applications in droidsafe.

Guardian, T. (2016). https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet.

Sun, M. and Tan, G. (2014). Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, pages 165–176, New York, NY, USA. ACM.

Viet, V., Tong, T., Clark, A., and Mé, L. (2010). Specifying and enforcing a fine-grained information flow policy: Model and experiments. 1.