Flowverine: Leveraging Dataflow Programming for Building Privacy-Sensitive Android Applications

Eduardo Gomes[†], Igor Zavalyshyn[†‡], Nuno Santos[†], João Silva[†], Axel Legay[‡]
[†]INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
[‡]UCLouvain, ICTEAM, Louvain-la-Neuve, Belgium
Email: {eduardo.l.gomes,igor.zavalyshyn,nuno.m.santos,joao.n.silva}@tecnico.ulisboa.pt, axel.legay@uclouvain.be

***Abstract*—Software security is a fundamental dimension in the development of mobile applications (apps). Since many apps have access to sensitive data (e.g., collected from a smartphone's sensors), the presence of security vulnerabilities may put that data in danger and lead to privacy violations. Unfortunately, existing security solutions for Android are either too cumbersome to use by common app developers, or may require the modification of Android OS. This paper presents Flowverine, a system for building privacy-sensitive mobile apps for unmodified Android platforms. Flowverine exposes an API based on a dataflow programming model which allows for efficient taint tracking of sensitive data flows within each app. By checking such flows against a security policy, Flowverine can then prevent potential privacy violations. We implemented a prototype of our system. Our evaluation shows that Flowverine can be used to implement mobile applications that handle security-sensitive information flows while preserving compatibility with existing Android OS and incurring small performance overheads.**

I. Introduction

An ever-growing number of mobile apps collects highly sensitive user data, e.g., location, photos, or health-related data. As the leakage of personal data can cause serious privacy breaches, app developers face the challenge of making sure such data is handled securely. For instance, a fitness-tracking app that reads the user's heart rate from a Fitbit fitness tracker must guarantee that this information can never be shared with unauthorized parties. However, ensuring the absence of bugs and security vulnerabilities is in itself a difficult task due to the complexity of the Android API. Furthermore, any third-party libraries [1, 2] (e.g., ad libs) included in the app, may have their own vulnerabilities, or, worse, contain malicious code leaking user data. Thus, it is important to have mechanisms in place that allow both app developers and users to control sensitive data flows within their apps, and consequently block those flows that can lead to security or privacy violations.

Unfortunately, despite the number of security improvements featured in the latest Android OS versions, no mechanisms are yet available for enforcing information flow control (IFC) policies. Many proposals from academia [3–10] refine Android's coarse-grained permission system, but fall short at controlling how sensitive data is processed inside the apps. Some other tools [11–13] employ static code analysis, which, however, can result in high false positive rates, fail to track flows performed via the Android API, or may be impractical to adopt in case the source code is not available (e.g., third-party libraries). Other systems overcome these limitations through the use of dynamic taint analysis [14, 15], but require changes to the Android OS.

To complement existing techniques, we propose Flowverine, a system that allows app developers to build secure-by-design privacy-aware Android apps. Flowverine apps run on commodity Android devices and require no changes to the Android OS in order to track sensitive data flows and enforce security policies. The apps are written using a specific programming model and API (possibly including third party libraries) so that all sensitive data flows within an app can be tracked. App developers can specify security policies for white-listing sensitive data flows, e.g., "*heart rate readings from a user's fitness tracker can be sent exclusively to a specific cloud backend and nowhere else*". Users that install the app can verify such policies and employ additional restrictions.

Flowverine implements a taint tracking mechanism based on two techniques. Firstly, to increase the abstraction level and enable efficient static taint tracking, we built upon the concept of *element-based programming* recently proposed for (simple) smart home apps [16], and apply it to complex Android apps. In Flowverine, all sensitive data flows must become explicit by construction, i.e., an app must be written as a graph where the nodes (named *elements*) represent compute units and the edges represent data flows. Flowverine provides a set of native elements – named *trusted elements* – that mediate an app's access to the Android native API. Because trusted elements come with a specification that describes how the data flows through the Android runtime, Flowverine allows for sound static taint tracking to be performed across Android API calls.

Secondly, if an app includes third party code that needs to access the raw Android API, Flowverine uses sandboxes to isolate such code inside *untrusted elements*, and Aspect-Oriented Programming (AOP) to intercept native Android API calls and perform dynamic taint analysis. AOP precludes the need to modify the OS, thus favoring compatibility.

Our performance evaluation shows that Flowverine has a relatively small impact on app execution time and has no noticeable impact to the user experience. We implemented three use case Android apps that showcase the ability of Flowverine to (1) prevent sensitive data flows that are not explicitly indicated in the app graph provided by the app developer, (2) allow for the strict privilege separation of multiple independent flows within any given app, and (3) support the main Android API programming abstractions.

II. Building Privacy-Sensitive Apps

A. Challenges in Building Android Apps

Android provides a popular platform for mobile apps. We highlight three major challenges faced by developers when building apps that manipulate privacy-sensitive data from local sensors or external connected devices. These challenges arise mainly from Android's programming and security models.

Tracking direct sensitive data flows: Tracking information flows between *source* and *sink* Android API calls – i.e., the calls that allow an app to obtain sensitive data and send it to remote parties, respectively – based on the inspection of a data flow graph can be cumbersome and error-prone as a result of the app separation into components (e.g., Activities) and the asynchronous nature of Android programming. Many static analysis tools can help to automate this task, but are seldom used in practice because of high false positive rates.

Tracking indirect sensitive data flows: Sensitive data flows can also be generated indirectly, i.e., outside the data flow paths between source and sink API calls. Some flows can be established through internal Android data structures, e.g., via an app context (akin to a global object store) in which independent app components can store and retrieve data using specific API calls. An indirect flow can then occur if one component stores data inside the app context and another one reads that data from it. Tracking such flows using existing taint analysis tools requires changes of the Android OS [14].

Enforcing privilege separation: Another difficulty lies in the fact that Android's permissions are too coarse-grained and many Android API calls have no differentiated access controls for different parts of a given app. This complicates privilege separation for different pieces of app logic. For instance, once the network access has been granted to an app, one cannot restrict the range of endpoints that the code (e.g., a third-party ad library) can connect to. Android does not currently support access control policies based, e.g., on the target URL.

B. Element-based Programming for Smart Homes

Element-based programming model was featured in HomePad [16], a system that provides a privacy-friendly runtime for apps that interact with smart home devices. HomePad's programming model enables users to find out how a given app uses the collected data, and apply fine-grained security policies based on the internal data flows generated by the app.

Element graphs: HomePad apps must be structured as a directed graph of interconnected *elements*. Elements consist of API functions that implement some well-defined behavior. The edges of the graph express all possible data paths, as HomePad only allows data to flow through these paths. Consider the HomePad app example in Figure 1. The app uploads camera frames to a cloud server when it identifies a person at the front door of the house. The application is implemented by three interconnected elements. The `IPCamera` and `HttpReq` elements are part of HomePad's native API, and are thus deemed *trusted*. The developer needs to create an app-specific

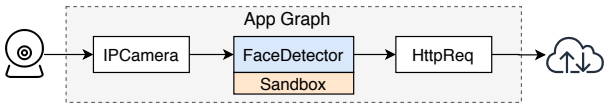


Fig. 1: Example of a HomePad application graph.

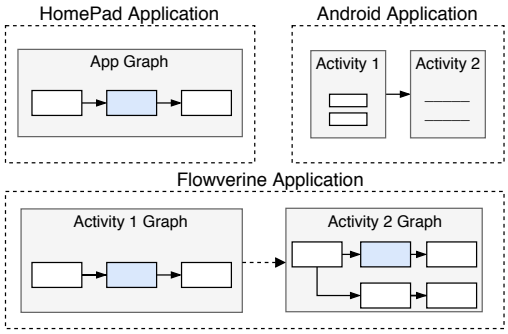


Fig. 2: Programming models compared.

element – `FaceDetector` – which contains code to receive camera frames, perform face detection, and send the results to the cloud. Because the `FaceDetector` code is not part of the native API (i.e., it is implemented by the developer, possibly importing a third-party library) this element is termed *untrusted*. Nevertheless, thanks to the connections indicated in the element graph and by sandboxing untrusted elements' code, HomePad only allows camera frames to be sent out via the `HttpReq` element configured with a predefined URL.

Taint tracking: By representing an app in the form of an element graph, HomePad allows for statically tracking both direct and indirect information flows within each app. This is achieved by the automatic generation of a *data flow model* written in first-order logic which takes into account i) the topology of the element graph to reason about the propagation of taint, and ii) a predefined specification of how each trusted element propagates taints from its inputs to its outputs depending on the semantics of the API call it invokes. Based on this model, HomePad can then determine if any of the generated app flows might violate a given security policy. Moreover, since trusted elements operate with specific data sources or sinks (e.g., exact camera, or URL, respectively), it is possible to implement fine-grained information flow security policies and enforce privilege separation. For instance, we can prevent an app from sending a given camera footage to any Internet address other than a particular (trusted) URL, while letting it freely send unrelated data to other Internet hosts.

C. Proposal: Element-based Programming for Android Apps

Given the HomePad benefits, we propose to adopt an element-based programming for building secure-by-design Android apps. As such, we introduce several innovations:

1. Android app components as element graphs: In Flowverine, each app component is written in the form of an element graph (see Figure 2). As in HomePad, an element executes some functional unit, and can only interact with other elements through the explicit edges connecting them. The graphs are

expressed in a declarative fashion, which allows for integration with popular visual programming tools for app development.

2. Trusted elements adopted for mobile API: The Flowverine API consists of a set of trusted elements. These are provided by certified modules that are assumed to work properly without undesirable side effects. Access to the native Android API, e.g., network calls, is mediated by specific trusted elements that can be used for different purposes, namely; i) obtain data from a given source (e.g., a hardware sensor, UI, or another app component), ii) send data to an external sink (e.g., a network host, UI, or another component), or iii) perform data transformation (e.g., data encryption). Each trusted element provides a well-defined interface.

3. Untrusted elements to host unmodified legacy code: As in HomePad, untrusted elements serve the purpose of running sandboxed code provided by the developer. In addition, Flowverine supports the inclusion of third-party legacy libraries, which often require direct access to the native Android API. To prevent privacy breaches Flowverine hosts legacy code inside untrusted elements and needs to implement additional runtime mechanisms to block any unauthorized API accesses.

D. Challenges Related to Android Specifics

Unlike HomePad apps with a simple structure, Android apps contain multiple components (e.g., activities or services) which interact with each other through asynchronous callbacks. Tracking sensitive data flows in such an intertwined system of classes and methods is a challenging task. To address this challenge, Flowverine implements a middleware that provides an abstraction layer for all app components, including the UI ones, and controls the propagation of events carrying sensitive data between them. Flowverine intercepts native API calls and enforces runtime security policies without changes in the underlying OS. We provide more details in the next section.

III. Design

This section presents Flowverine. We begin by describing its architecture, and then discuss its most relevant design details.

A. Architecture

Flowverine provides a software framework for development of privacy-sensitive Android apps such that the developers and users alike maintain fine-grained control over the sensitive information flows generated by these apps. To this end, Flowverine provides a middleware that exposes an API based on element-based programming and a set of mechanisms that i) analyze the internal app data flows using static and dynamic taint tracking, and ii) check such flows against an information flow control (IFC) policy to identify potential security or privacy breaches. An app developer can specify an IFC policy to validate the app compliance with the terms of the service's privacy policy (which states how the personal data will be collected and managed) and the data protection rules imposed by law. The user can specify an IFC policy (through a user-friendly interface) which prevents specific data flows that the user deems privacy sensitive.

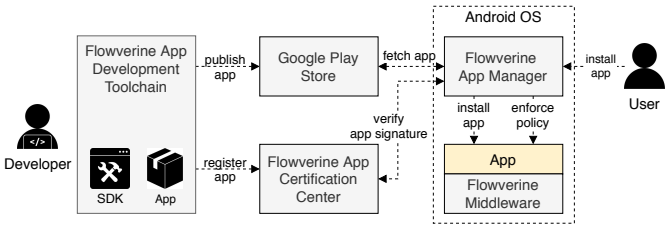


Fig. 3: Flowverine framework components and workflow.

Figure 3 presents Flowverine's components. It includes an *app development toolchain* that allows developers to build their apps, link them against the Flowverine API, and check compliance against a developer-provided IFC policy. If the app satisfies all the security requirements, the developer submits a signed app package to an app store and registers it in the Flowverine *certification service*, which validates that the app has been properly instrumented by the toolchain. A user can then install this app through the Flowverine *manager app* running on the user's smartphone. The manager app manages all Flowverine apps on the device, e.g., fetches the app package from the app store, and checks that the app has been properly certified by the Flowverine certification service. The manager app also provides a UI interface through which the user can specify an IFC policy and check apps' compliance with it. If the app passes the check, it can then be executed. Every Flowverine app is linked against the Flowverine *middleware* – i.e., a set of libraries – that provides all the runtime support for the execution of the app, which is based on an element graph. Next, we describe how a Flowverine app can be developed.

B. Application Development

The process of developing a Flowverine app involves i) the implementation of the app itself, and then ii) using the toolchain to build the app, check IFC policy compliance, generate the app package, and submit it for public release.

To implement an app, the developer creates individual element graphs for every app component. To illustrate this process, imagine we want to implement a simple Click-Counter app that displays the number of times the button on a screen was clicked. As in traditional Android programming, in Flowverine, this app has an associated Activity and a UI layout file written in XML. However, since this Activity will be implemented as an element graph, it will be programmed as a Java subclass of Flowverine's API `ActivityGraphDescriptor`. This class provides methods that allow the developer to specify the elements of the graph and their connections. Figure 4 shows what this graph looks like.

This graph consists of two trusted elements – `ViewClick` and `TextUpdater` – and an untrusted one – `HandleClick`. The former implement UI functions and serve as interfaces to the button and a text view defined by their respective IDs. The latter contains the code that handles a button click event and increments the counter. This code is provided in Listing 1. According to the app graph, the events generated by `ViewClick` will be routed by the Flowverine runtime to `HandleClick`, which in turn will increment the counter and

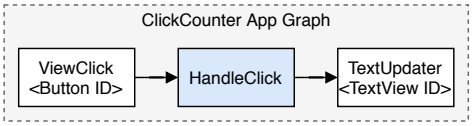


Fig. 4: Element graph of ClickCounter app.

```
@CustomElement(name="HandleClick")
public class HandleClick extends Element{
  int ctr = 0;
  @EventReceiver
  public void onEvent(...) {
    sendEvent(new Event<String>("Cnt: "+(++ctr)));
}}
```

Listing 1: Implementation of HandleClick element.

generate an output event. The Flowverine runtime will route this event to `TextUpdater` which will display the counter value on screen. Next, we present the Flowverine runtime internals.

C. Application Execution Runtime

The Flowverine runtime (see Figure 5) consists of a middleware comprising several libraries, which are included in the app package along with the code responsible for the implementation of the app's element graph. At runtime, Flowverine materializes the elements of the app graph into three sets of Java objects : i) *stubs* that point to the implementation of the trusted elements referred to in the element graph, ii) *sandboxes* initialized with instances of untrusted elements' classes, and iii) a *path descriptor* which restricts communication between trusted elements' stubs and untrusted elements' sandboxes according to the connections in the app graph. These objects are created when the app starts and destroyed when it terminates.

Elements communicate by sending events to the corresponding stub through an internal message broker: *event bus*. For instance, `ViewClick` element sends an event on every button click. These events are routed by the event bus strictly as specified in the path descriptor, therefore ensuring that no information flows can occur besides those specified in the app's element graph. The functions implemented by the trusted elements – through a set of built-in drivers – is covered below.

D. Trusted Elements API and Drivers

The Flowverine API consists of a set of trusted elements that developers can use to create their apps' element graphs. The logic of these elements is implemented by a set of drivers which are part of the Flowverine middleware. One of the obstacles in adopting element-based programming for Android, is the complexity of Android API, both in terms of number of calls, and the sophistication of operations they implement (e.g., multithreading). To cope with this complexity, we created various types of drivers which interface with specific classes of functions provided by the Android API. Next, we briefly mention the most important types of Flowverine drivers:

1. UI drivers: As opposed to smart home apps, mobile apps have very rich user interfaces. The Android API has many classes for creating UI widgets named Views. A View

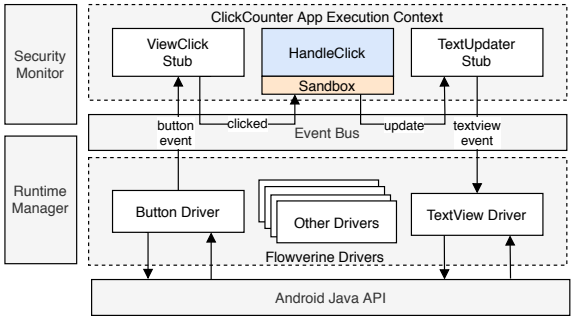


Fig. 5: Execution runtime of a ClickCounter application.

represents a UI object on the screen which the user may interact with. Flowverine's API offers trusted UI elements, e.g., the `ViewClick` and `TextUpdater`, which provide standard functionalities of Button and TextView, respectively. These elements have specific input and output ports which can be connected to other elements. An input port of `TextUpdater` element can be used to update a TextView on the screen, and an output port of `ViewClick` can be used to emit a button click event to any downstream element in the app graph.

2. Component drivers: Activities are very common components. An Activity represents an app's screen and is in charge of UI-dependent tasks. Throughout its lifecycle an Activity instance transitions through different states and provides a set of callbacks that are invoked when it enters a new state – e.g., `onCreate` or `onDestroy`. With Flowverine, developers can handle these state changes by using the elements provided as part of the Activity Life Cycle Module. For instance, the `ActivityCreated` element notifies the elements connected to its output port when the graph's Activity is created.

3. System drivers: This class of drivers includes trusted elements for supporting multithreading, inter-component communication (ICC), and inter-process communication (IPC). For multithreading support, Flowverine apps can execute tasks in parallel with the `AsyncFork` and `AsyncJoin` elements. With ICC driver elements the graphs of different app components can be connected. Lastly, IPC drivers enable apps to interact with each other via the `Send`/`Received` trusted elements.

4. I/O drivers: Flowverine implements several drivers for interfacing with network, storage, and sensors. For networking, Flowverine includes a Web driver which allows apps to perform HTTP requests through trusted elements, such as `HttpGetReq` or `HttpPostReq`. These elements must be set up with i) the destination URL, and 2) the expected data types received in the response. Other drivers provide access to Bluetooth (BLE driver) and location services.

E. Protection against Untrusted Element Code

The code of the untrusted elements can be written by the app developer or be part of a third-party library. In either case, to ensure that the app's data flows are strictly bound to the data paths indicated in the app's element graph, such code cannot be allowed to execute without restrictions. Flowverine adopts several mechanisms for securing legacy third-party libraries:

1. Sandboxing untrusted elements: To prevent untrusted element code from interfering with other classes of the runtime sharing the same ART virtual machine, we take advantage of Java's class loading model. Flowverine includes a custom-made sandbox classloader which is in charge of resolving classes within an isolated namespace. Each untrusted element instance is placed inside its own sandbox such that only the classes associated to it by the app developer can be loaded and instantiated. Any attempts to access (blacklisted) classes from the runtime environment will throw an exception. Some (harmless) classes are whitelisted and are delegated to the parent class loader, i.e., the class loader of the runtime.

2. Weaving untrusted elements code: It is also necessary to prevent untrusted elements' code from performing operations that circumvent the data paths defined in the app graph. This may cause a buggy code to interfere with the system or, worse, a malicious code (e.g., spyware shipped with a third-party library) to leak sensitive data. Therefore, an untrusted element code must be prohibited to perform the following operations:

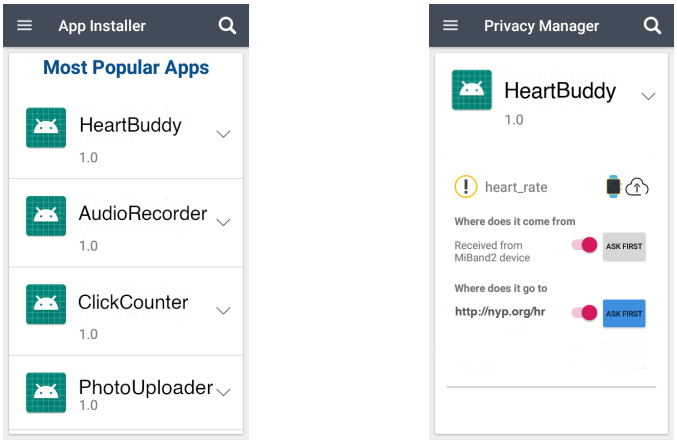
- Direct calls to the Android API methods, which are reserved to be invoked by the Flowverine middleware.
- Execution of native (C/C++) code, which could be used to inject malicious code in the Flowverine runtime.

To this end, the code is sanitized using Aspect-Oriented Programming (AOP). With AOP, we define a set of execution points patterns to be executed only by the middleware. By weaving the app in search of points that match these patterns, and injecting a safety-guard code, we can assure that untrusted elements' code has no access to Android's API or to a Java native interface. In Flowverine, weaving is performed at build time, by a tool of Flowverine toolchain named *code weaver*. It runs on Java bytecode files and inspects all the app code provided by the developer, including any imported libraries.

Weaving is particularly useful in the case of legacy third-party libraries which have not been modified to use Flowverine's trusted element API. At runtime, if an untrusted element attempts to execute a flagged Android API call, the safety-guard code takes over and lets the security monitor (see Figure 5) decide what to do. The default procedure is to terminate the app, but the security monitor may allow the operation to proceed as long as the resulting data flow follows the app's graph connections. For instance, it can intercept an HTTP call and forward it to the Flowverine network driver, which, in turn, translates this call into an event compatible with a trusted HTTP element. If such an element exists in the app's graph and connects with the currently executing untrusted element, then this operation can be seamlessly carried out.

F. Validation of Information Flow Control Policies

By ensuring that an app can only generate information flows explicitly declared in the app's element graph, Flowverine helps prevent security breaches that may result from programming errors or by the inclusion of malicious libraries. Flowverine provides complimentary tooling support for validating the information flows of a given app against an information flow control (IFC) policy. Although for different contexts, IFC policies are useful to both app developers and users.



(a) App installer screen. (b) Privacy report of an app.

Fig. 6: App Manager screenshots.

An IFC policy consists of a set of rules aimed to flag specific information flows between sources and sinks in a given app. The sources and sinks are identified by the elements of the app's graph. A source is an element that collects sensitive data type (e.g., an interface to a connected fitness tracker). A sink is an element that can be used for sending data to a remote location (e.g., an HTTP trusted element). Each rule identifies some relevant information flow based on a 3-tuple: source, sink, and data type. These rules are expressed in the form of Prolog predicates and are matched against a model of the app's element graph also written in Prolog. This model consists of a set of predicates, some of which are automatically generated based on the app's graph topology, and others are predefined. The predefined predicates specify how information propagates internally for each trusted element of the Flowverine API. These predicates are loaded for each trusted app element and come as part of the Flowverine software. Flowverine validates if any of the app flows violate the IFC policy by using a Prolog engine to query the app's model based on the policy rules.

Flowverine provides two sets of tools for IFC policy validation. App developers can use a *policy checker* included in the toolchain to check for undesired information flows. For debugging and testing purposes the app developers can specify their own IFC policy in JSON (then converted to a Prolog predicate and checked against the app's element graph). App users can use the App Manager to supervise the information flows generated by Flowverine apps and block any sensitive flows. Figure 6 presents two screenshots of the App Manager's UI. When installing an app (see Figure 6.a), the user can select an app from a list provided by the Flowverine app certification center. During installation, the App Manager generates a default IFC policy that reflects the flows in the app's element graph. This policy is shown to the user (see Figure 6.b), who can block specific data flows or disable the app. The user may additionally force the app to ask permission every time it attempts to obtain or send out a certain data type.

IV. Implementation

We implemented a Flowverine prototype, and we have prepared a public release as an open-source project. In total we wrote about 23K lines of Java code. This includes the Flowverine middleware and trusted element API (9K LoC), the App Manager (3.5K LoC), Flowverine toolchain (1K LoC), the certification service (0.7K LoC), and five testing Flowverine apps (9K LoC). We adopted tuProlog as Flowverine's Prolog engine, and leveraged AspectJ for code weaving. We implemented a specific Flowverine BLE driver for interacting with a Xiaomi Mi Band 2, which we used to develop a privacy-sensitive fitness tracker app.

Our current prototype has several limitations. Given the extent of the Android API, we have only implemented a representative set of trusted elements for the Flowverine API. In particular, our API is limited to: system drivers, Activity and Service components, five different UI views, and I/O drivers for networking, BLE interfacing, and location services.

V. Evaluation

We evaluate Flowverine on several fronts, as described next.

A. Case Study

To help understand some key challenges in building secure mobile apps, we introduce a simple health-monitoring app named HeartBuddy. The app obtains a heart rate value from a connected fitness tracker – via a Bluetooth Low Energy (BLE) connection – displays it on the screen, and periodically sends an average value to a hospital's cloud service (nyp.org) for diagnosis of various heart-related diseases. The app also displays an ad banner fetched from a remote server (adspull.com).

Due to the private nature of a heart rate data the app developer must ensure that only the average values are sent to the specified cloud service and nowhere else. Likewise, the app users expect this property to hold at any time. Additionally, the developer needs to guarantee that there is no interference between the main app functionality and the ad library activity. The ad library can never have access to heart rate data.

In Flowverine, the HeartBuddy app can be implemented as a graph of elements displayed in Figure 7. On the left side, there is BLE Service activity responsible for interacting with a connected fitness tracker and properly decoding its signals (proxied by native Android API). On the right, we see two app activity graphs: one implementing the main app functionality, and the other one responsible for ad banner activity. A new heart rate data event emitted by a trusted `HeartRateDecoder` element arrives to an untrusted `HandleNewHR` element, which forwards it to the `Reducer.Average` element that computes an average heart rate value and feeds it to the second untrusted app element – `SendHR`. The latter one is responsible for preparing an HTTP POST request to a hospital's cloud server. This request will be sent when the user clicks the "Send" button on the screen (an event handled by the `ButtonView.Click` element). Finally, the `TextView.Update` element updates the current heart rate value on the screen. The ad banner operations are controlled by a second isolated graph consisting

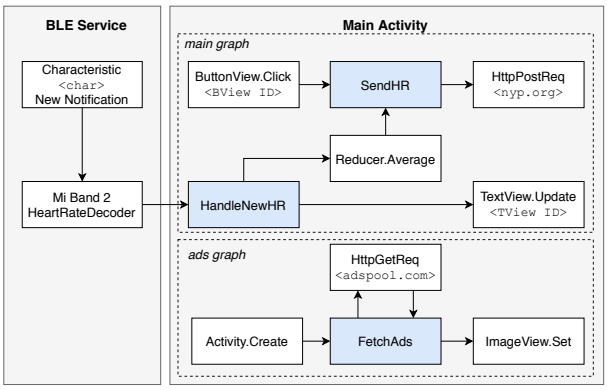


Fig. 7: Flowverine HeartBuddy app design.

of four elements. This graph starts execution when a new activity is created (invoked by `Activity.Create` element). The untrusted `FetchAds` element receives the latest ad data by making an HTTP GET request to an adspull.com service and displays it on the screen via `ImageView.Set`.

Since the main activity graph and the graph responsible for ad activity are completely separated, there are no data flows between their respective elements. Flowverine also ensures that the network calls are restricted to endpoints that were defined in the app package: `HttpReq` controls the destination and type of requests. By analyzing both app graphs, Flowverine can effectively track the heart rate data propagation and transformation. The security monitor detects the data type leaving the user phone (averaged value) which is in accord with the user expectations. The security monitor also ensures that a third-party ad library will not have access to heart rate data and will not be in conflict with GDPR regulations.

B. Comparison with Vanilla Android

Our comparison between Flowverine and the vanilla Android system is twofold. First, we analyze the security models of both systems: the former which is based on an IFC model, and the latter on a discretionary permissions system. Our goal is to evaluate if apps developed with Flowverine are more transparent regarding their sensitive data flows, and if our framework allows users to understand and have a fine-grained control over how installed apps treat sensitive data. To this end, we use the example HeartBuddy app (see Figure 6). In Flowverine, the App Manager reports to the user that: (1) the app collects heart-rate data (i.e. data type) from a fitness tracker (i.e. source), and (2) the app sends collected data to *nyp.org* (i.e. sink). The user is then offered the option to either block a given flow or require the app to ask for permission each time the flow occurs. In vanilla Android, the permissions system allows the user to deny the app's access to BLE service, but not to the Internet. Thus, in scenarios where mobile apps need to send sensitive data to the cloud, Flowverine's reports are more informative and give the user better control over the app's activities than Android's native permission system.

Secondly, we assess the development effort required to write Flowverine apps in terms of lines of code (LoC) as compared to the standard Android app programming model. Table I

TABLE I: Apps created for bare Android and with Flowverine.

App Name	Accessed Resources	Lines of Java Code (LoC)	
		Traditional	Flowverine
ClickCounter	UI	15	21
PhotoUploader	UI, Filesystem, Internet	98	64
HeartBuddy	UI, Dialog windows, Internet, Bluetooth, Mi Band 2 services	480	174

TABLE II: Build time for traditional and Flowverine apps.

App Name	Build time (ms)		Overhead (ms)
	Traditional	Flowverine	
ClickCounter	1556	2164	608 (39%)
PhotoUploader	1938	2772	834 (43%)
HeartBuddy	1692	2374	682 (40%)

presents the results of this comparison for three apps of various complexity: ClickCounter (see Figure 4), PhotoUploader (which uploads a photo to a cloud service), and HeartBuddy. We see that for very simple apps, Flowverine requires more lines of code. However, the LoC number is significantly lower (sometimes almost 3x less) for complex Flowverine apps that rely on multiple existing trusted elements to interact with various resources (e.g. device sensors, storage, network). Developers can thus benefit from higher-level programming abstractions for writing their apps.

C. Performance

To evaluate the performance of Flowverine, we used a server with a 2.80GHz Intel i7-7700HQ CPU and 16GB of RAM for build-time and validation experiments. To evaluate the Flowverine's runtime and App Manager performance, as well as its memory and battery consumption we use Neffos C5A smartphone running Android 7.0 Nougat equipped with a 1.30 GHz quad-core processor 1GB RAM and a 2300 mAh battery.

To measure how much time Flowverine adds to application compilation and packaging, we compare build times of the three use-case apps developed using Flowverine and traditional Android programming models (see Table II). Flowverine adds on average 700 ms to the build time. In all cases the overhead was mostly due to Code Transformer's weaving process.

On average it takes 7.7 sec for Flowverine to perform an integrity check on a newly published app package. The validation time depends mainly on the app size, but with the infrequent release cycle of apps, this delay can be tolerated.

Next, we analyzed the time that the App Manager needs to inspect an app graph, extract data flows information, and display this information to the user. The inspection time correlates closely with the app graph complexity: with more app elements generating various data types there are more potential data flows for App Manager to inspect. It takes between 2 to 7 sec to analyze the app graphs consisting of 3 and 21 elements respectively.

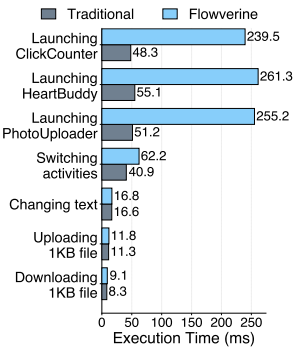


Fig. 8: Runtime benchmarking tests results.

We also evaluate the Flowverine impact on apps' startup time and some of the common app activities. The results are presented in Figure 8. Flowverine adds, on average, 200 ms to an app's launch time, and 20 ms when switching app activities. However, for other app activities, e.g. network calls, the overhead is negligible (<1 ms). While Flowverine has a noticeable impact on app startup time, there is no meaningful performance loss on app activities after that. We note, however, that further performance optimizations are possible.

Lastly, Figure 9 features the results of memory consumption comparison. Flowverine apps use slightly more memory due to the sandboxing mechanism which replicates classes bytecode definitions consequently increasing the amount of memory used by the app process. Also note that in our experiments Flowverine had insignificant impact on app's battery usage.

D. Security Assessment

Our system must defend against potential security vulnerabilities introduced by buggy or malicious code contained in a mobile app. Such vulnerabilities could result in the circumvention of the data path restrictions enforced by the app's element graph, and / or in the violation of a given IFC policy. To assess how Flowverine mitigates potential attacks, we consider the following scenarios:

1) Untrusted elements directly interacting with the device resources (direct access attack).
2) Unconnected untrusted elements sharing data with each other (data sharing attack).
3) Malicious code set to run outside untrusted elements (middleware bypass attack).
4) Altering the app's bytecode after weaving-based sanitization (weaving disable attack).

Flowverine introduces several mechanisms to make app code more resilient to attacks. Its sandboxes prevent (1) and (2) by blocking the execution of dangerous classes that aim to access the device's resources, and loading the classes of untrusted element code in independent class loaders so that they do not share memory. Flowverine's code weaver tool checks all the apps bytecode, preventing attacks of the third kind. Lastly, the certification service of Flowverine only validates apps that have been correctly sanitized by the code weaver, essentially preventing attacks of the fourth type.

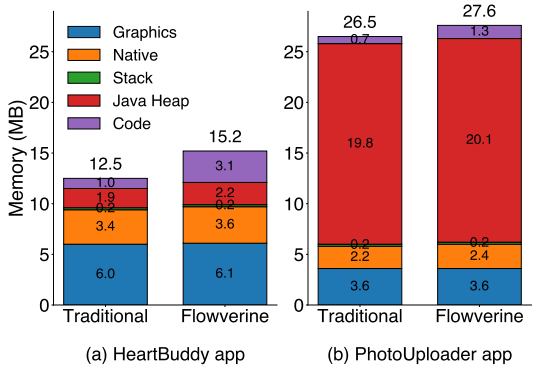


Fig. 9: Memory usage: traditional apps vs. Flowverine apps.

VI. Related Work

Information flow analysis tools for Android: Most of these tools [11, 12, 14] employ some kind of taint tracking to inspect the paths of tainted data samples. While these tools provide high coverage, they may often lead to a high false-positive rate, and overlook some control-flow data leaks. MutaFlow [13] detects this last type of leak but fails to detect a delayed attack. Furthermore, a study carried in 2018 [17] shows that FlowDroid and IccTA fail to track flows that involve ICC calls with complex strings formed from sensitive data. TaintDroid [14] and TaintART [15] overcome these challenges, but require changing Android's core as they use dynamic taint tracking. Additionally, these tools operate at a variable level and are prone to side-channel attacks [18]. Flowverine provides a complementary technique that combines both static and dynamic taint analysis without changing the Android OS.

Extensions to Android's permission system: Most proposed extensions [5–8] enforce control over app data access but ignore internal app data flows, making it difficult to determine, e.g., the Internet locations where sensitive data is sent by a given app. Some systems that monitor how data flows within apps [9, 10], rely on TaintDroid [14] to detect leaks, which means they inherit TaintDroid's limitations discussed above. Furthermore, all these solutions [3–6, 9, 10] but two [7, 8] involve changes to Android's core. Aurasium [7] and AppGuard [8] do not modify the OS, but instead rely on dynamic instrumentation of apps. However, both of these techniques interfere with Android's ecosystem and raise compatibility problems. Another problem with the studied solutions is that they all control access to data at a very low abstraction level, such that it becomes hard for app developers and users to understand how apps use sensitive data and for what purposes.

In summary, Flowverine finds itself in-between two worlds. On the one hand, similarly to taint tracking tools, it checks the propagation of sensitive data samples from their sources to potential sinks. Flowverine however avoids overtainting by operating on a higher level of abstraction (variable level vs. user-friendly data type) and relying on predicates that describe data propagation rules within the app. On the other hand, Flowverine serves as a privacy enhancement to the Android ecosystem without requiring any changes to its core modules.

VII. Conclusion

We proposed Flowverine, a new system for Android that helps app developers and users to protect sensitive data manipulated by mobile apps. Flowverine's dataflow programming model allows app developers to expose how apps internally collect and share user's sensitive data. Our evaluation shows that Flowverine performs well, and that, despite the introduction of a new programming model, new secure-by-design apps can be created without much additional effort. Flowverine can also make app development accessible to a wider community, as it can be easily integrated with visual programming tools.

Acknowledgments: We thank the reviewers for their comments. This work was partially supported by national funds provided by Fundação para a Ciência e a Tecnologia (FCT), via the UIDB/50021/2020 project, and a CISCO research grant.

References

[1] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. of CCS*, 2016.
[2] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *Proc. of SANER*, 2018.
[3] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proc. of MCSA*, 2011.
[4] Z. Xu and S. Zhu, "Semadroid: A privacy-aware sensor management framework for smartphones," in *Proc. of DASP*, 2015.
[5] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava, "ipshield: A framework for enforcing context-aware privacy," in *Proc. of NSDI*, 2014.
[6] D. Wu and S. Bratus, "A context-aware kernel ipc firewall for android," in *Proc. of ShmooCon*, 2017.
[7] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proc. of USENIX Security 12*, 2012.
[8] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard: Enforcing user requirements on android apps," in *Proc. of TACAS*, 2013.
[9] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proc. of CCS*, 2011.
[10] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff, "Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android," in *Proc. of ISTP*, 2012.
[11] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. of PLDI*, 2014.
[12] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proc. of ICSE*, 2015.
[13] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller, "Detecting information flow by mutating input data," in *Proc. of ACE*, 2017.
[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. of OSDI*, 2010.
[15] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proc. of CCS*, 2016.
[16] I. Zavalyshyn, N. O. Duarte, and N. Santos, "Homepad: A privacy-aware smart hub for home environments," in *Proc. of SEC*, 2018.
[17] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/icc-ta, amandroid, and droidsafe," in *Proc. of ISSTA*, 2018.
[18] G. S. Babil, O. Mehani, R. Boreli, and M. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices," in *Proc. of SECRYPT*, 2013.