INFORMATION FLOW CONTROL FOR ANDROID APPLICATIONS
WITH EMBEDDED WEBPAGES

by

Julian Sexton

A THESIS

Submitted to the Faculty of the Stevens Institute of Technology
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE - CYBERSECURITY

Julian Sexton, Candidate

ADVISORY COMMITTEE

David A. Naumann, Advisor Date

Antonio R. Nicolosi, Reader Date

STEVENS INSTITUTE OF TECHNOLOGY
Castle Point on Hudson
Hoboken, NJ 07030
2016
INFORMATION FLOW CONTROL FOR ANDROID APPLICATIONS
WITH EMBEDDED WEBPAGES

ABSTRACT

The permission system in Android is intended to achieve security and privacy for the user. Application developers make informal claims about the security of their applications, but these are not enough to guarantee user security in a meaningful way. Information flow analysis tools exist to help validate the claims developers make. Existing tools lack the capability to provide a fine-grained analysis of applications which use WebView, an Android component which allows for embedding webpages within applications. Since Android allows information to pass between Java and JavaScript environments, information flow becomes difficult to accurately track for these applications. We propose a collaboration of analysis tools for Android and JavaScript, and evaluate it on a case study application with a WebView component. Through the translation of the user policies for each of these tools, as well as facades over the WebView APIs, we further the capabilities of Android analysis tools.

Author: Julian Sexton
Advisor: David A. Naumann
Date: May 3, 2016
Department: Computer Science
Degree: Master of Science - CyberSecurity
ACKNOWLEDGMENTS

I would like to thank the people who made the completion of this project possible. In particular:

Professor David Naumann, my advisor throughout this project and previous ones, for his constant guidance and support.

Andrey Chudnov, for his effort in developing and extending JEST and guidance towards the completion of this project.

Felipe Fonseca, for his work in creating our case study application.
# Table of Contents

Abstract iii

Acknowledgments iv

List of Figures vii

1 Introduction 1

2 Background 3

2.1 Android WebView 3

2.2 SPARTA 3

2.3 JEST 4

2.4 SPFacade and LogFacade 4

2.5 Problem Statement 5

3 Target Application 6

4 Approach 10

4.1 SPARTA Policy File 10

4.2 SPARTA Labels 11

4.3 JEST Policy 14

4.3.1 Conditional Release 16

4.4 SPARTA/JEST Label Translation 16

4.5 End-to-end Analysis 20

4.6 Limitations of SPARTA 20

4.7 Analyst Workflow 21

5 Related Work 23
5.1 Effective Inter-Component Communication Mapping in Android with Epicc: An
Essential Step Towards Holistic Security Analysis .................................. 23
5.2 TaintDroid: An Information-Flow Tracking System for Realtime
Privacy Monitoring on Smartphones ......................................................... 23
5.3 Analyzing Inter-Application Communication in Android ....................... 24
5.4 FlowDroid: Precise Context, Flow, Field, Object-sensitive and
Lifecycle-aware Taint Analysis for Android Apps .................................... 24
5.5 ANDROMEDA: Accurate and Scalable Security Analysis of Web
Applications ................................................................................................ 24

6 Conclusion .............................................................................................. 25

Appendix A Facades on Persistent Storages .................................................. 26

Bibliography .............................................................................................. 33
List of Figures

3.1 Case Study Screenshot ........................................................................... 7
3.2 Case Study Sequence Diagram ............................................................... 8
4.1 Lattice for annotations of case study’s WebView API ......................... 13
4.2 Facade for label translation .................................................................. 17
4.3 JEST Facade for Android ....................................................................... 19
Chapter 1

Introduction

The permission system in Android phones provides very little assurance to the user about what an application actually does with the information available to it. While it does notify the user what an application may access, it does not show what an application does access, nor does it provide concrete proof that an application adheres to any specified information policy. The ability to verify that an application follows a given information flow policy is desirable, and this is what the SPARTA project aims to do [1]. It is capable of tracking information across various Android APIs, and verifies that an application follows a specific policy on that information.

However, the introduction of Android’s WebView allows for information to be passed back and forth between the Android Java program and the JavaScript of a webpage loaded into the application. This presents a potential security issue even for apps which verify correctly with SPARTA; if information is passed into JavaScript, it can be freely manipulated, with no way for SPARTA (which operates solely on the Java language) to ensure the confidentiality and integrity of that information. The best SPARTA can do in this situation is show that information flows to and from the “web”; there is potential for clearer, more detailed policies which involve the JavaScript code as well as the Android application code.

We detail an approach to solving this problem. Through the use of JEST, a dynamic information flow checker for JavaScript [2], we improve the verification of an application by allowing information flow to be tracked from SPARTA statically, while providing assurance dynamically that the policy on information is not violated within the WebView component. We choose a combined static and dynamic approach, because Android applications are capable of executing strings containing JavaScript code; furthermore, the JavaScript code is able execute specified Android functions at runtime. This would be very difficult to analyze statically.

SPARTA and JEST were created as standalone tools, however. Since SPARTA is static and JEST
is dynamic, the collaboration of these two tools requires some translation between the policies and languages. We implement a facade on the Android system to integrate these two tools. This facade allows developers who wish to prove the security of their application to do so, without drastically changing the structure of their code. Furthermore, we hope that the policy translation we propose will potentially serve as an example for future collaborations between differing analysis tools.

In practice, it would be very useful for a developer to be able to prove the security of their application in some measurable way. In order for this measure to mean anything, the developer has to be able to show the verification of their application through an end-to-end analysis. An application which undergoes review by an analysis tool which does not effectively consider all available routes for data to pass through has little guarantee on the state of its data afterwards. The WebView component in Android is one such route — without specific analysis of the JavaScript code in relation to the application’s policy, a developer who uses WebView is restricted in the guarantees he can make about the security of his application.

In the following sections we discuss the work specifically needed as a base for this project (Previous Work, chapter 2), the application we target for analysis (Target Application, chapter 3), the approach we took in solving the problem (Approach, chapter 4), and other projects which are similar in nature to this one (Related Work, chapter 5).

SPARTA and JEST are both open source distributions; this project uses the SPARTA v1.0 virtual machine as well as JEST\(^1\). The project is intended to be open source; please request access from either Julian Sexton (jsexton@stevens.edu) or David Naumann (dnaumann@stevens.edu).

\(^1\)As JEST is not publicly available at the time of publishing, the version will be provided with the distribution.
Chapter 2

Background

In this section we describe work specifically used in or as a direct base for, this project, and not work which separately works on the same or similar problems. Related works which were not prerequisites for this work are detailed in chapter 5.

2.1 Android WebView

The Android API for Java has a component called WebView. It uses the WebKit renderer to display web content within an Android application. This allows developers to program freely in HTML, JavaScript, and other web languages, and have it rendered within their application. Furthermore, it allows for loading pages from web sources, whether it be for the portability of a specific service, or for user’s general browsing needs. WebView not only allows arbitrary JavaScript code to be executed within the context of an application, but provides the web content with the ability to call specified functions in the application’s Java code, through the use of the “@JavascriptInterface” label on functions within the Android application. Any method made available to JavaScript through the @JavascriptInterface annotation is placed into a JavaScript object called “Android”, which can then be used to invoke callbacks into the application.

2.2 SPARTA

SPARTA [3][1] (Static Program Analysis for Reliable Trusted Apps) is a static analysis tool for Android applications built on the Checker Framework [4], in order to analyze information flow across an application in Android. It uses a set of “sinks” and “sources” to identify where information flows to and from, and tests this against a policy provided by the user. The policy consists of a file specifying allowed flows between sources and sinks, as well as annotations on the source code of the application. SPARTA then acts as a modified compiler which verifies that there are no direct flows
which violate the policy.

However, these sinks and sources are often not very fine-grained, and some applications would benefit from being able to specify a more detailed policy over various interfaces and storages. Though SPARTA does not produce warnings for indirect or implicit information flows, we chose to use it because it uses a diverse system of labels tailored to the Android system, allowing for the specification of precise policies on information flow.

2.3 JEST

JEST [2] is an analysis tool for JavaScript created by Andrey Chudnov. It was implemented to enforce information flow control on web applications with fine-grained policies. JEST uses a user defined policy, in which the user defines both a sensitivity lattice and mappings of those sensitivities to channels within the page. These channels are typically external URIs, or elements of the DOM. This policy and the webpage is then provided to JEST.

JEST statically inlines the webpage into self-monitoring code, based on the policy. This makes JEST specifically applicable to our problem, because unlike other JavaScript analyzers, it does not modify the JavaScript engine. While the current distribution does not support every JavaScript library, it has a precedent of using facades to interface with different I/O streams.

While JEST can perform inlining via a proxy server, we use JEST on a fixed set of webpages for an Android application. Once the pages have been inlined to be self monitoring, the runtime monitor soundly enforces the policy.

2.4 SPFacade and LogFacade

In the same way that SPARTA does not completely analyze WebView components, it also does not provide a granular analysis of persistent storage. We have worked on tracking SPARTA’s labels through some of the available persistent storages in Android. In particular, the Shared Preferences API, a key-value storage, is tracked with the generic label of “SHARED_PREFERENCES” in
SPARTA. However, in practice, it would be useful to be able to specify policies over specific fields of this storage, since information flowing in and out of it may need to have different policies. We approached the problem in SharedPreferences (and in the CallLog and MessageLog APIs) through runtime facades over the APIs. The user specifies a policy over the keys of the persistent storage; specifically, with which SPARTA sources and sinks each key should be associated.

The SPFacade included methods to be used instead of the `put()` and `get()` methods of the Android SharedPreferences API; the facade methods use Java’s Reflection API to retrieve SPARTA annotations provided by the user. These SPARTA annotations are compared with the user’s provided policy to ensure that at runtime, flows which violated a fine-grained policy on Android’s SharedPreferences are disallowed. Please see appendix A for more information about this facade.

Our LogFacade performed a similar function on the CallLog and MessageLog APIs; these primarily read-only APIs operate on the phone’s records of calls and SMS messages. It was desirable to be able to consider different types of information (such as phone numbers, call durations, and call timestamps) with differing sensitivities. As SPARTA does not provide this fine granularity, our facade again ensured that the information being retrieved from these storages was stored in correctly annotated variables, through the use of the Reflection API in Java.

2.5 Problem Statement

Previously, our facades worked primarily on persistent data storages, but WebView allows information to flow into a completely separate runtime environment. This JavaScript environment can modify and leak the data in many ways without SPARTA being able to detect it. We use JEST to combat potential breaches of the application’s policy. These two tools have different policy definitions and expectations. We aim to build a facade as a working interface between them, to provide an end-to-end analysis of applications which use Android’s WebView.
Chapter 3

Target Application

When considering an application to use as the case study for information flow research, we considered several criteria. The application we used had to be simple enough to work clearly with SPARTA labels, while being complex enough to not have a trivial policy on information. Furthermore, the application had to be realistic: it had to perform some functionality on potentially sensitive data, such that some user would not only find the security policy useful, but also the application itself. We also needed an application which required both Android and JavaScript to perform its primary functions - if one of these two was totally unnecessary, or if there was no good reason to use one, then a typical developer would probably create the application with only one of these technologies.

The application we are targeting as an example for our approach is called the SimpleOrg App. It is a modified version of an Emacs Org Mode [5] reader created by Felipe Fonseca. The app relies on a webpage to render Org files, while explicitly not rendering any section or subsection tagged with “:NODISPLAY”. The user has the option of overriding this functionality - by placing a formatted password inside the Org file, the user can enter this password in the app to display all the tagged sections. This feature mimics org-crypt, a cryptography module for OrgMode which displays encrypted text for specially tagged sections. In addition, the app provides another functionality in which it allows the user to append to the Org file via the microphone and speech-to-text APIs. The user has the option to mark the recording as private, in which case, it will be appended with a “:NODISPLAY” tag. The app also allows for importing calendar events from the Org file into the user’s Android Calendar. This gives us a nontrivial policy on information. The application informally promises that not only will the sensitive sections of the document (including both sections marked “NODISPLAY” and the user’s password) be hidden from the display, they will not be imported into the user’s calendar. With these functionalities, we create flows of potentially sensitive information into and out of JavaScript, such that in order to track sensitive information throughout the app,
Figure 3.1: Screenshot of our case study. The file is displayed with formatting in the main area, hiding private sections, unless a password is entered in the input field. The menu allows for syncing to the calendar, and recording public and private additions to the loaded file. The webpage used in this app is fixed, and is included with the app upon download.
Figure 3.2: Message sequence diagram of the case study’s functionalities. Potentially sensitive information is highlighted in red.
more than just SPARTA labels are necessary. We also create an application which makes significant use of Android and JavaScript; while processing Org files in JavaScript can be made easier with certain libraries, access to the phone’s microphone is something that the Android system provides. We create two obvious sensitive flows of information in and out of the WebView component - the user’s password must be considered sensitive; when it is passed to the JavaScript it must not be released. Furthermore, information tagged with “:NODISPLAY” must not be allowed to be placed into the user’s calendar. These two flows are what we aim to track using a combination of JEST and SPARTA.

In Figure 3.2, we show the functionality of the application and its communication between Android and JavaScript. The first potentially sensitive flow comes through the requestPassword() function, which is made available to the JavaScript code. This function returns the user’s entered password, which should not be leaked to the display or calendar.

The next potentially sensitive flow is the file itself - when the user presses the Choose File button, they are prompted to select their org file, which may contain sensitive information tagged “:NODISPLAY”.

The user can choose to append to the file via the microphone. This is done entirely on the Android side; the captured message is written directly to the file, and the user must reload the file in the webpage.

Lastly, the user can choose to import events from their file into the calendar. This involves a transfer of information from JavaScript to Java - the dates and descriptions of each event have to be passed to the Java environment to be written to the Android calendar. Since this information comes from the file, it is important that information tagged sensitive is not sent to the calendar.
Chapter 4

Approach

In this section, we discuss our justification for the security of the sample application. We begin with the app’s informal policy.

Essentially, the goal of the app is to display Emacs Org mode files, hiding any section marked “:NODISPLAY”. These sections are not hidden, however, if the user enters a correct password (declared at the top of the Org file). This policy imitates the Emacs OrgCrypt module. The user can choose to append to their Org file via the microphone. The user can also choose to import dates from their Org file into their phone’s default calendar. Only sections which are allowed to flow to the display (either not tagged with “:NODISPLAY” or unlocked by password) are allowed to flow to the calendar.

In this section, we discuss our definition of a facade written in Java which translates between SPARTA and JEST. Through the use of SPARTA, JEST, and this facade, we make an argument for the security of the application.

4.1 SPARTA Policy File

SPARTA is completely static - it requires a policy file which specifies allowed flows; any flows not specified are prohibited and will cause SPARTA to warn about the application’s information flow claims. The policy file has the format:

{SOURCES} -> {SINKS} //first allowed flow
{SOURCES} -> {SINKS} //second allowed flow
...

We express our case study’s informal policy in SPARTA’s notation.

RECORD_AUDIO -> FILESYSTEM
FILESYSTEM -> INTERNET
USER_INPUT -> INTERNET
INTERNET -> WRITE_CALENDAR
We allow information from the microphone (RECORD_AUDIO) to flow to the filesystem; this represents the capability of the user to append to their Org file via speech-to-text. We allow information from files (FILESYSTEM) and user input (USER_INPUT) to flow to our WebView component (INTERNET), which represent the user password and the loaded Org file. Lastly, we allow information from the WebView component (INTERNET) to flow to the user’s Android Calendar (WRITE_CALENDAR).

However, SPARTA requires a transitive closure of these allowed flows, so that the developer explicitly acknowledges that a flow of $A \rightarrow B$ and $B \rightarrow C$ implies $A \rightarrow C$. Also, SPARTA rightfully considers flows from Intent sources to be sensitive; for the purposes of our application, they are not. We use Intents only to choose an Org file, the name of which is not sensitive, and since we do not have registered Broadcast Receivers, intents cannot be received from other applications in such a way that our application’s functionality would be affected. We allow information from intents to flow to anywhere else in the app. Furthermore, SPARTA has limitations when it comes to WebView; SPARTA does not check that the returned value of a function labeled “@JavascriptInterface” has a sink of INTERNET. So, to avoid the password being allowed to flow to an external network, we remove the flow USER_INPUT $\rightarrow$ INTERNET. Our revised policy file for our case study, then, is:

- RECORD_AUDIO $\rightarrow$ FILESYSTEM, INTERNET, WRITE_CALENDAR
- FILESYSTEM $\rightarrow$ INTERNET, WRITE_CALENDAR
- INTERNET $\rightarrow$ WRITE_CALENDAR
- INTENT $\rightarrow$ FILESYSTEM, INTERNET, WRITE_CALENDAR

### 4.2 SPARTA Labels

After designing the allowed flows, we finish creating the policy by annotating the application’s source code. In particular, there are some annotated functions which can be called by the WebView component. Note that the “INTERNET” permission in SPARTA would actually allow information to flow to external networks. However, JEST prevents this in our WebView component, and a
manual analysis of the application's Java source shows that the application does not actually access external networks. Ideally, a sink specific to local WebView pages would be used. For simplicity and for lack of precedent with local webpages in SPARTA's "INTERNET" sink, though, we chose to simply use "INTERNET".

```java
@JavascriptInterface
public @Source({"USER_INPUT"}) @Sink({}) String requestPassword() {
    ...
}
```

We provide the above function to the WebView component so that when the Org file is being rendered, if the user has entered a password, the JavaScript source has access to that password. Since the password has come directly from an Android form field, we label the return value to have a source of "USER_INPUT". Though in our policy we allow flows from "USER_INPUT" to the "INTERNET" to accurately represent the policy, here we label it with empty set of sinks. We do this because of a limitation of SPARTA; SPARTA does not check that the returned value of a function labeled "@JavascriptInterface" has a sink of INTERNET. So, to avoid confusion about the destination of the file's password in JEST, we label it with an empty set of sinks.

We provide the following functions to the WebView component as a way for the JavaScript environment to pass the date and description of a single calendar event back to the Java environment.

```java
@JavascriptInterface
public void scheduleDate(@Source({"INTERNET", "FILESYSTEM"}) @Sink({"WRITE_CALENDAR"}) String date) {
    ...
}
```

```java
@JavascriptInterface
public void scheduleDesc(@Source({"INTERNET","FILESYSTEM"}) @Sink({"WRITE_CALENDAR"}) String desc) {
    ...
}
```

These functions place the descriptions and dates into a queue, to be later inserted into the user's calendar (when the JavaScript environment is finished passing back the dates and descriptions of all insensitive calendar events).
Figure 4.1: Sub-lattice of SPARTA labels which spans the labels needed for the WebView API of the sample app. Note that in SPARTA, including more Sources denotes a more sensitive label (information is more tainted), while including more Sinks denotes a less sensitive label (allowed to flow to more places).
4.3 JEST Policy

Though we have represented the app’s informal policy as a policy for the SPARTA toolset, SPARTA is incapable of tracking information through the WebView component. So, we have decided to use JEST for tracking information through the WebView component. JEST too, requires a policy file. JEST policy files typically deal with channels such as external URIs, for information to enter and exit web applications. For our application, the channels we are primarily concerned with are the entrances and exits of information from the Android code. Our embedded webpage has no form fields or external network traffic, but it does have a “Choose File” button with which the user selects their Org file. Please see Figure 3.2 for a more detailed explanation of the sensitive flows to and from JavaScript.

So, we design our JEST policy based on the policies of the functions which invoke or are invoked by the WebView component. The policy files for SPARTA and JEST are specified differently, however. The JEST policy file represents a lattice generated by the base sensitivities specified, as well as a mapping of channels to these sensitivities. The JEST monitor enforces that at runtime, information flow does not violate the structure of the lattice. The SPARTA policy file, on the other hand, is specified by declaring the allowed flows. To create a JEST policy, then, we use a subset of the lattice of SPARTA labels, where a label is defined as follows.

\[
\text{label} := "@Source(sources) @Sink(sinks)"
\]

\[
sources := \{\} \mid \{\text{source}^*\}
\]

\[
sinks := \{\} \mid \{\text{sink}^*\}
\]

\text{source} is one of SPARTA’s defined Sources

\text{sink} is one of SPARTA’s defined Sinks
The subset of the lattice we use is enough to span the labels on the app’s interface to the WebView component. See Figure 4.1 for the sub-lattice we use. From that lattice, we write our policy according to JEST’s specifications; instead of using URIs and DOM elements as channels, we use single parameter functions as channels.

```
policy bitvector principals {
    sourceINTERNET = func://Android.scheduleDate@1, func://Android.scheduleDesc@1;
    sourceFILESYSTEM = func://Android.scheduleDate@1, func://Android.scheduleDesc@1, dom://fileInput, file://*;
    sourceUSERINPUT = func://requestPassword;
    sinkEMPTY = func://requestPassword, dom://fileInput, file://*;
}
```

This policy maps `scheduleDate()` and `scheduleDesc()` to sensitivities labeled “sourceINTERNET” and “sourceFILESYSTEM”, since those are the sources on the parameters for the two functions. The sink on each of those parameters is “WRITE,CALENDAR”, which is represented by not including the “sinkEMPTY” sensitivity for those channels (since having less sinks implies more sensitive data). The @1 in each URI indicates that the first parameter is used.

The “fileInput” node is an element of the DOM which receives the file when the user chooses their Org File. It is marked with “sourceFILESYSTEM”, since it originates on the Android filesystem, and is also marked with “sinkEMPTY”, since the data in the file is potentially sensitive.

In the case of the function `requestPassword()`, we would like the label on the return type of the function, and not the parameter, since information is being returned to JavaScript from the Java environment. Though we can create this sensitivity label in the JEST policy, we must enforce that this label matches the SPARTA annotation in the Java source. We enforce this through our facade in section 4.4.

Once we have a policy file for JEST, we run the WebView component’s code through JEST’s
inliner, which essentially rewrites the JavaScript code to be self monitoring (upon execution).

4.3.1 Conditional Release

Our informal policy requires that some data from the file be released to the Android calendar, conditionally, on whether it is tagged with “:NODISPLAY”. As mentioned earlier, however, we mark the data in the file with “sinkEMPTY”. JEST supports conditional release, through a declassification mechanism.

```java
if ((docpassword == userpassword && docpassword != "") || (keep)) {
    newlines.push(declassify(declassify(lines[k],
        "func://Android.scheduleDesc@1"),
        "func://Android.scheduleDate@1"));
}
```

Here, we declassify lines of a file, on the same condition that we release them - either the user has entered a correct password, denoted by `docpassword == userpassword && docpassword != ""` or we have set a flag `keep` which marks insensitive items.

4.4 SPARTA/JEST Label Translation

Now that the app’s Java code has successfully verified statically in SPARTA, and the app’s JavaScript code has been inlined by JEST for the provided policy, there is still one more task, which must be done dynamically. At runtime, information will be passed from the app to the WebView component; however, there is nothing yet to enforce that a variable representing certain information in the Java code has an equivalent sensitivity to the same information in the JavaScript code.

We design a facade in Java over return values to enforce this equivalence. This facade, to be called upon returning control from Java to JavaScript, ensures that the value passed to JavaScript is labeled in a way that JEST is able to correctly recognize. This facade uses Java’s reflection API to translate SPARTA labels into JEST labels; information being passed from Java to JavaScript maintains equivalent sensitivity. The above are excerpts from our facade on return values. The facade requires two things. The first is a translation file, which is essentially a mapping between
public String returnToWV(Object ins, String fieldname) {
    ...
    HashMap<String, HashSet<String>> translation =
        parseTranslationFile(translationPolicy);
    ...
    ArrayList<String> sensitivities = new ArrayList<String>();
    ...
    f = ins.getClass().getField(fieldname);
as = f.getDeclaredAnnotations();
    ...
    /* pull out all the sources and sinks for the given field */
    for (int i = 0; i < as.length; i++) {
        if (as[i] != null && isSink(as[i])) {
            sinks = ((Sink) as[i]).value();
        } else if (as[i] != null && isSource(as[i])) {
            sources = ((Source) as[i]).value();
        }
    }
    ...
    /* finding the equivalent of each source in the JEST policy file (using our translation file) */
    for (int j = 0; sources != null && j < sources.length; j++)
    {
        for (String key : translation.keySet()) {
            if (translation.get(key).contains(sources[j]))
            {
                sensitivities.add(key);
            }
        }
    }
    ...
    /* returns a JSONObject which acts as a labeled JEST value */
    return (new JESTBox((String) f.get(ins), sensitivities)).toString();
}

@JavascriptInterface
public @Source(\"USER_INPUT\") String requestPassword() {
    ...
    return returnToWV(insertionObject, "password");
}

Figure 4.2: This facade translates labels from SPARTA to JEST, based on a policy file provided by the user.
SPARTA sources and sinks to JEST channels. The policy should map a JEST channel with one sensitivity level to a single SPARTA source or sink (or “EMPTY” to represent an empty set of sources/sinks). When translated, that source or sink will be mapped to every sensitivity level the channel is associated with.

The semantics of the translation file are as follows:

\[
\text{policy} := \{(\text{sourceline}|\text{sinkline})^*\}
\]

\[
\text{sourceline} := \text{SOURCE jestchannel source newline}
\]

\[
\text{sinkline} := \text{SINK jestchannel sink newline}
\]

\text{newline} is a newline

\text{source} is one of SPARTA’s defined Sources

\text{sink} is one of SPARTA’s defined Sinks

\text{jestchannel} is a channel in the JEST policy, which has the equivalent sensitivity level of the given sink or source

The second is an “insertion class”. This insertion class is denoted by the Object \text{ins}. Since SPARTA does not require that every variable be annotated, this facade requires that the user manually provide an annotated variable. By providing the facade with an object and the name of the field, we are able to use Java’s Reflection APIs to retrieve not only the annotation on the variable, but also the value of the variable. We then convert the annotation to a label in JEST through the translation file, and send both the translated label and value to JEST.

The return value of this facade is intended to be returned immediately to JavaScript in a function labeled @\text{JavascriptInterface}, without modification. We do this in the function \text{requestPassword}().

Secondly, we extend JEST to include facades on calls to the Android system, and to treat single-parameter functions as channels which have the sensitivity of their parameter. In doing so, JEST’s
for (var f in Android) {
    androidFields[f] = ESSLField(FunctionProxy(function () {
        var unboxedArgs = [];
        var fnchannel = "func://" + f.name;
        for (var i = 0; i < arguments.length; i++) {
            var cl = policy.locationLevel(fnchannel + "0" + i.toString(), true);
            var dl = arguments[i].l.join(pclabel);
            if (dl.leq(cl))
                unboxedArgs.push(ToStringBox(arguments[i]).v);
            else stop("Android", 0, 0, "Policy violation");
        }
        var facaderet = Android[f].apply(Android,
            unboxedArgs);
        var jsonret = JSON.parse(facaderet);
        var retl = lowlevel();
        for (var i = 0; i < jsonret.level.length; i++) {
            retl = retl.join(policy.locationLevel(jsonret.level[i]));
        }
        return primbox(jsonret.value, retl);
    }));
}

Figure 4.3: This facade is an extension to JEST which we use to process labels on functions in the object named “Android”.

monitor now disallows calls to the app’s WebView interface which violate JEST’s policy; information is only passed from JavaScript to Java if the information is less sensitive than the channel it is being passed to.

In Figure 4.3 we define a facade for JEST over the Android object. This facade iterates over the functions in the object, inlining them to check for correct JEST labels on parameters, and also to interpret the return value provided by the Java facade. The Java facade returns a value and a set of channels; this facade actually labels the returned value with the labels of the provided channels.

Note that the name of the Android object is specified by the user in the application source code. For simplicity, we use a single name, but this could be generalized to have a generic name in the JEST facade, or to simply enforce the name “Android” in the application source.
4.5 End-to-end Analysis

At this point, the Java portion of the app has been verified against a policy for information flow by SPARTA; though SPARTA does not recognize implicit flows, we performed a manual analysis to ensure that the Java code never branches on sensitive values. The JavaScript code used in this app has been inlined by JEST against a policy developed as a subset of the lattice of SPARTA labels. Furthermore, at runtime, information being passed from Java to JavaScript is labeled in a way JEST understands by a facade invoked by the Java code. Again at runtime, JEST disallows passing information back to Java in cases where it would violate the policy. JEST is sound and is capable of verifying with implicit flows. We claim then, to provide an end-to-end analysis of this application which has meaningful information flow to and from Android’s WebView.

4.6 Limitations of SPARTA

As mentioned earlier, SPARTA does not attempt to handle implicit or indirect flows of information. This is easily demonstrated in our application with the minor addition of a branch on sensitive data.

```java
public @Source({"INTENT", "RECORD_AUDIO"}) String
newScheduleEntry(
    @Source({"INTENT", "RECORD_AUDIO"}) String desc ,
    @Source({"INTENT", "RECORD_AUDIO"}) boolean display)
{
    ...
    if (!display && desc.indexOf(0) == 'm') {
        EditText et = (EditText)
        this.findViewById(R.id.passwordfield);
        this.findViewByld(R.id.passwordfield);
        et.setText(desc);
    }
    ...
}
```

The above code correctly produces a warning that `desc` cannot flow to the display, as it is not allowed by our policy. This means that `et.setText()` is a channel that potentially makes information observable (on the display). The following code, however, produces no warnings.

```java
public @Source({"INTENT", "RECORD_AUDIO"}) String
newScheduleEntry(
    @Source({"INTENT", "RECORD_AUDIO"}) String desc ,
    @Source({"INTENT", "RECORD_AUDIO"}) boolean display)
{
    ...
    if (!display && desc.indexOf(0) == 'm') {
        EditText et = (EditText)
        this.findViewById(R.id.passwordfield);
        this.findViewByld(R.id.passwordfield);
        et.setText(desc);
    }
    ...
}
```
Although both are communicating information on an observable channel, the latter example produces no warnings because it is an indirect flow of information. One could consider the latter example repeated in a loop to release all the sensitive information contained in `desc`. Though SPARTA can be circumvented through the use of indirect and implicit flows, a manual analysis of our program showed that there are no branches on sensitive data within the Java code.

### 4.7 Analyst Workflow

The approach we took to ensuring the security of this application significantly reduces the effort required by an analyst seeking to verify an Android application which uses WebView. The analyst would start by first designing an informal policy for the app they wish to verify. They would translate this informal policy into a SPARTA policy file and annotations on the source code.

Our lattice is a subset of a larger lattice on all possible SPARTA labels. With a single implementation of this larger lattice as a JEST policy, there would essentially be no requirement for an analyst to design a translation file for the facade we provide. The analyst would however, need to assign their specific channels to these sensitivities.

The analyst would run JEST’s inliner with the policy they created, and get an inlined version of their JavaScript code. Using this self monitoring code in place of their embedded webpage, they would simply have to replace return statements in functions labeled `@JavascriptInterface` with a return of the result of our facade.
While there is some work to be done before verification can be achieved easily, much of the process can be automated, and this paper simply uses potential outputs of that automation.
Chapter 5

Related Work

In this section we will discuss work which tackles similar issues in Android, but is not directly used in our project.

5.1 Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis

Epicc is a static analysis tool created for mapping Inter-Component Communication (ICC) in Android. The tool is used to detect vulnerabilities in ICC, and also to identify potential attack vectors on these vulnerabilities. Furthermore, they state that one application of their tool is to detect whether two applications are colluding, whereas previously this was only done dynamically. Since Epicc considers Java classes as input, it seems unlikely that it would be able to perform a fine-grained analysis of JavaScript code, in its current state. It is possible that the techniques it uses to detect vulnerabilities in intents could be extended to detect some vulnerabilities in invocations to the WebView API, but without actually analyzing the JavaScript execution or source code, it is unlikely that all the vulnerabilities introduced by WebView could be found with just Epicc [6].

5.2 TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones

TaintDroid is a widely used dynamic analysis tool for Android, which uses taint-tracking to track sources of sensitive data. It does this on a method, variable, message, and file level. The creators of this tool analyze 30 popular applications with it and are able to decide which ones leak sensitive user data to their servers, and even to advertisement companies. TaintDroid taints information in Android applications at a source level, and then performs analysis through the Dalvik VM interpreter; it is unclear whether their tainting process can be applied to WebView applications. Furthermore,
Android has switched from Dalvik to ART in recent versions, which may present a difficulty for TaintDroid [7].

5.3 Analyzing Inter-Application Communication in Android

ComDroid is an analysis tool for detecting inter-app communication vulnerabilities. They focus primarily on Intents, and while they make recommendations about fixing the vulnerabilities they discovered through the use of their tool, it seems that their goal was to get statistics about how many applications actually have these inter-app vulnerabilities. Though ComDroid does not specifically consider WebView, it furthers end-to-end analysis beyond single applications [8].

5.4 FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps

FlowDroid is a static analysis tool for taint analysis in Android. To achieve this, it uses a precise model of the Android life cycle, as well as models of the available call back methods. By doing so, it claims to lower its rate of false positives, and has greater efficiency than previous work because of its use of on-demand algorithms. FlowDroid focuses primarily on analysis of Java source code, and so does not seem capable of providing a fine-grained analysis of the WebView component [9].

5.5 ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications

Andromeda is a project which considers static analysis in web applications - it handles Java, JavaScript and .NET applications, and has a focus on sound, efficient, analysis of code. To do this, it uses lazy evaluation in combination with taint analysis, a technique which serves as the base for other projects, such as FlowDroid [10].
Chapter 6

Conclusion

This project utilized two main tools for analysis of an Android app: SPARTA and JEST. SPARTA is a static analysis tool for Android apps, which has built in knowledge of the various components of the Android system. The policy specification is simple and based on sinks and sources which often correspond directly to Android permissions. It is then possible to have SPARTA infer annotations for the entire app, with guidance from the analyst. From there, it is a matter of either correcting the app to not leak information or correcting the annotations and policy. It is not difficult to learn how to use it effectively. Furthermore, as SPARTA is static it is possible to have a dynamic analysis phase separate to SPARTA without actually interfering with SPARTA’s verification process.

JEST, on the other hand, is a dynamic monitoring system which was made with the primary use case of collaboration in web apps. Since it creates self-monitoring code, it was ideal for use in Android’s WebView - it did not require modification of the JavaScript engine. In JEST, there is little work to be done after deciding on a policy - some type casts may need to be made to include a breach of the policy as legal. However, for this project, we needed to extend JEST’s facades to include an “Android” object. Creating facades for JEST is not trivial, and requires some thought about which labels should be preserved and the types of parameters and return values.

We created a facade in Java which is generic; it can be used for any application. The policy file it requires should be tailored to the user’s JEST and SPARTA policies. The lattice we created and the JEST and SPARTA policies are specific to our case study. The facade we created in JEST is mostly generic; it will handle functions of any name, but currently, it only handles functions in the “Android” object. This requires the user to use the name “Android” when initializing the object for WebView, but in the future may be made more generic.
Appendix A

Facades on Persistent Storages

The SharedPreferences storage in Android is a key-value persistent storage which allows developers
to store minor values between sessions of the app. Since the developer can store any information
in here, it is very likely that the information can have different sensitivities. SPARTA only has
a single source and sink for the SharedPreferences, so fine-grained policies over the keys of the
SharedPreferences are not possible. We wrote the following facade over the Android API to allow
for more fine-grained policies on this persistent storage.

This facade requires an “insertion object”; this object contains fields explicitly annotated by
the user with SPARTA labels. The facade retrieves values from and inserts values into these fields.
However, it only does so if the annotation on the field and the policy for the specified key in the
SharedPreferences match. In doing so, it dynamically enforces a policy on a key-value storage while
simultaneously not interfering with SPARTA’s analysis.

The primary methods we were concerned with in the SharedPreferences class were `get()` and
`put()`, since those were the primary operations on the values in the storage.

```java
package com.example.spfacadeproject;
...

public class SPFacade{
...
    public static boolean checkLevels(String key, Field f, Field xf, Context c){
        int i, j, k;
        String curr;
        ArrayList<String> sinkLines = new ArrayList<String>();
        ArrayList<String> sourceLines = new ArrayList<String>();
        String[] words;
        boolean ret = true;
        boolean currentFound = false;
        Annotation[] ann = f.getAnnotations();
        Annotation[] xann = xf.getAnnotations();
```
FlowPermission[] sinks = new FlowPermission[1];
FlowPermission[] srces = new FlowPermission[1];

for(i = 0; i < ann.length; i++){
    if(ann[i] != null && ann[i].getClass().equals(xann[0].getClass())){
        sinks = ((Sink) ann[i]).value();
    }
    else if(ann[i] != null && ann[i].getClass().equals(xann[1].getClass())){
        srces = ((Source) ann[i]).value();
    }
}

try{
    InputStream SPPolicy =
        c.getAssets().open("SHPR_policy");
    Scanner scanner = new Scanner(SPPolicy);
    while(scanner.hasNextLine()){
        curr = scanner.nextLine();
        words = curr.split("");
        if(words.length > 1){
            if(words[0].equals("SINK")){
                sinkLines.add(curr);
            }
            else if(words[0].equals("SOURCE")){
                sourceLines.add(curr);
            }
            else{
                System.out.println("
                    "Invalid policy file.");
            }
        }
    }
    scanner.close();
}
catch(Exception e){
    System.out.println(e.toString());
}

for(i = 0; i < srces.length; i++){
    for(j = 0; j < sourceLines.size(); j++){
        curr = sourceLines.get(j);
        words = curr.split(" ");
if(words.length > 2 && words[1].equals(key)){
    for(k = 2; k < words.length; k++){
        if(words[k].equals(srces[i].toString())){
            currretfound = true;
        }
    }
}
ret &&= currretfound;
currretfound = false;
}
for(i = 0; i < sinks.length; i++){
    for(j = 0; j < sinkLines.size(); j++){
        curr = sinkLines.get(j);
        words = curr.split(" ");
        if(words.length > 2 && words[1].equals(key)){
            for(k = 2; k < words.length; k++){
                if(words[k].equals(sinks[i].toString())){
                    currretfound = true;
                }
            }
        }
    }
}
ret &&= currretfound;
currretfound = false;
}
return ret;
}
public void get(String key){
    getHelp(sp, key, field, xf, loc, context);
}
public static void getHelp(SharedPreferences sp, String key, Field f, Field xf, Object l, Context c){
    Object fv;
    try{
        fv = f.get(l);
        if(checkLevels(key, f, xf, c)){
            if(fv instanceof Integer){
                f.set(l, Integer.valueOf(}
sp.getInt(key, (Integer) fv));

} else if(fv instanceof Float){
    f.set(l, Float.valueOf(sp.getFloat(key, (Float) fv)));
}

} else if(fv instanceof Long){
    f.set(l, Long.valueOf(sp.getLong(key, (Long) fv)));
}

} else if(fv instanceof String){
    f.set(l, sp.getString(key, (String) fv));
}

} else if(fv instanceof Boolean){
    f.set(l, Boolean.valueOf(sp.getBoolean(key, (Boolean) fv)));
}

} else if(fv instanceof Set<?>){
    f.set(l, sp.getStringSet(key, (Set<String>) fv));
}

} else{
    System.out.println("Annotations of "+key+
" inconsistent with policy file.");
}

} catch(Exception e){
    System.out.println(e.toString());
}

public void put(String key){
    putHelp(spe, key, field, xfield, loc, context);
}

public static void put(String spname, int fl, String key, Object o, String fname, Context c){
    Field f;
    Field xf;
SharedPreferences s;
SharedPreferences.Editor se;
try {
    f = o.getClass().getField(fname);
    xf = SPFacade.class.getField("x");

    s = c.getSharedPreferences(spname, fl);
    se = s.edit();
    se.commit();

    putHelp(se, key, f, xf, o, c);
} catch (NoSuchFieldException e) {
    e.printStackTrace();
}

} 

public static void putHelp(SharedPreferences.Editor spe,
String key, Field f, Field xf, Object l, Context c){
    Object fv;
    try{
    fv = f.get(l);
    if(checkLevels(key, f, xf, c)){
        if (fv instanceof Integer){
            spe.putInt(key, (Integer) fv );
        }
        else if(fv instanceof Float){
            spe.putFloat(key, (Float) fv );
        }
        else if(fv instanceof Long){
            spe.putLong(key, (Long) fv);
        }
        else if(fv instanceof String){
            spe.putString(key, (String) fv);
        }
        else if(fv instanceof Boolean){
            spe.putBoolean(key, (Boolean ) fv);
        }
        else if(fv instanceof Set<?>){
            spe.putStringSet( key, (Set<String>) fv);
        }
    }
    spe.commit();
}
else{
    System.out.println("Annotations of "

} catch (Exception e) {
    System.out.println(e.toString());
}
}
Here are the semantics of the policy file required by the facade.

\[
policy := \{\text{sourceSpec} | \text{sinkSpec} | \text{newline}\}* \\
\text{sourceSpec} := \text{SOURCE keyname sources} \\
\text{sinkSpec} := \text{SINK keyname sinks} \\
\text{sources} := \{\} | \{\text{source}^{*}\} \\
\text{sinks} := \{\} | \{\text{sink}^{*}\} \\
\text{newline} \text{ is a newline} \\
\text{keyname} \text{ is the name of a key in the storage} \\
\text{source} \text{ is one of SPARTA’s defined Sources} \\
\text{sink} \text{ is one of SPARTA’s defined Sinks}
\]

Below is an example policy.

\[
\begin{align*}
\text{SINK signature SHARED_PREFERENCES DISPLAY} \\
\text{SOURCE signature SECURE_HASH} \\
\text{SINK app_usage SHARED_PREFERENCES} \\
\text{SOURCE app_usage CALL_LOG}
\end{align*}
\]
Bibliography


