Spartan Jester: end-to-end information flow control for hybrid Android applications

Julian Sexton*
MITRE Corporation†

Andrey Chudnov*
Galois, Inc.

David A. Naumann*
Stevens Institute of Technology

***Abstract*—Web-based applications are attractive due to their portability. To leverage that, many mobile applications are *hybrid*, incorporating a web component that implements most of their functionality. While solutions for enforcing security exist for both mobile and web applications, enforcing and reasoning about the security of their combinations is difficult. We argue for a combination of static and dynamic analysis for assurance of end-to-end confidentiality in hybrid apps. We show how information flows in hybrid Android applications can be secured through use of SPARTA, a static analyzer for Android/Java, and JEST, a dynamic monitor for JavaScript, connected by a compatibility layer that translates policies and value representations. This paper reports on our preliminary investigation using a case study.**

I. Introduction

In principle, strong information flow control (*IFC*) is needed to assure mobile users their privacy and security requirements are being met. But there remain several challenges to the widespread use of IFC, one of which we tackle in this paper: cross-language tracking.

It has become increasingly popular to build mobile apps as mobile *web* apps. The Android operating system allows for the inclusion of locally and externally hosted webpages: the WebView component allows two-way transfer of information between Java and JavaScript (*JS*). IFC tools exist for Java code (including in Android), for JS (in the browser), and for other programming languages; but few address flows across language boundaries. (See Section V for related work.)

Static analysis for JS is especially difficult due to the dynamic nature of the language, so most work on IFC for JS relies on monitoring [1]–[4]. Moreover, hybrid apps often embed changing web pages that are loaded dynamically (though in some scenarios it may be possible to perform static analysis on the server prior to serving the page). These considerations lead us to use dynamic monitoring for JS in hybrid apps. On the other hand, the high performance cost of monitoring leads us to eschew it for non-JS app components. Cross-language and mixed static-dynamic IFC raises new challenges for connecting different mechanisms and consistently interpreting policies.

* Partially supported by NSF award CNS1228930

† The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author. Approved for Public Release; Distribution Unlimited. Case 17-0121.

a) Problem statement: In this paper we are concerned with reasoning about, and enforcing, the end-to-end security of Android apps which make use of a WebView component. We consider security requirements that refer to I/O devices and to UI elements including those in web pages loaded by WebView. We are interested in fine-grained information flow policies in which individual form fields or data store elements may be designated as sensitive. Practical policies inevitably require downgrading, after sanitization or conditioned on events. A typical example, for confidentiality, is sending the low four digits (only) of a CC-number in the clear, after an authenticated transaction is completed. Another example is indicating on the display whether a login attempt is successful, while not revealing the actual password or its hash. Our ultimate goal is to achieve strong IFC for hybrid apps, including the tracking of implicit flows. The goal in this paper is to investigate an approach, leading to an agenda for further work. We explore the challenges through a case study using two existing tools.

b) Approach: For a statically typed language like Java, a programmer-friendly way to specify fine-grained policies is to augment declarations with security-annotated types. The annotations can, for example, refer to principals allowed to read or write (as in Jif [5]). Alternatively, they can refer to input and output channels such as the microphone and display screen. In JS, security labels can be associated with input/output API calls or objects. But JS doesn't have static type annotations, so labels need to be specified in a separate file or as additional meta-data in the HTML page. For enforcement, static analysis has the advantage of having no runtime cost and the possibility to find vulnerabilities in advance of deployment. For Java, we adopt an existing static analyzer tailored to Android: *SPARTA* [6]. For the JS part of an app, however, we advocate dynamic monitoring. We choose inlined monitoring: *JEST* [4] performs a source-to-source transformation that instruments the code to be self-monitoring, avoiding the need to modify the JS engine (or any other part of the platform).

Because SPARTA provides Android-specific policy features, we start by formulating our policy using SPARTA. The policy is then translated into a corresponding JEST policy. Through the use of façades written in both Java and JS, we preserve the labels on values as they enter and exit the WebView component, enabling end-to-end policy enforcement.

c) Outline and contributions: In this paper, we present the first system to provide IFC for hybrid apps that uses dynamic monitoring for JS together with static analysis for

the Java part of the app. As a first step in the evaluation of our approach, we present an illustrative case study. Section II describes the case study app, its security requirements, and our attack model. Section III describes our integration of the tools, which involves bridging code in Java and JS as well as policy translation. Section IV discusses limitations of the work so far and challenges for further development. Section V reviews some related work and VI concludes.

II. Case study

The hybrid app in our case study provides a subset of the functionality of `org-mode` (http://orgmode.org/). Org is an extension of GNU Emacs providing personal productivity tools like calendar, notes, task tracker, etc. The information is stored in plain text files (*org files*) with a forest structure and straightforward syntax, not unlike XML in expressiveness. Our app parses, interprets, and displays org files. To support an on-the-go workflow, notes can be captured using the microphone and Android speech-to-text capabilities. Org calendar events can be exported to the Android system calendar. Finally, the app includes a mockup of `org-crypt`, an org-mode module that allows encryption of subtrees in org files. For simplicity, we forgo encryption and instead provide the ability to tag subtrees as private (using the "NODISPLAY" tag). A password is required to display these tagged subtrees and to export tagged events to the calendar. When a voice note is being captured, it can be tagged as private.

We use HTML and JS to implement the parsing, interpretation and display of org files. However, pure web apps do not have access to the microphone, speech-to-text, and system calendar APIs. We use a native Android component (in Java) to provide these functionalities. It hosts the web component locally using WebView, which uses WebKit to display HTML and execute JS. Native components can run arbitrary JS in the context of the web component's page, and JS in the web component can call back to the native component via an Android object that WebView instantiates in the global scope. Java methods annotated with @JavascriptInterface become fields of the Android object and can be called like any JS function. Lee et al [7] is a good source for more detail on WebView.

a) Security requirements and attack model: In our app, sensitive information originates from the Android file system, the microphone, and text input from the user (the password entry), and it can be displayed to the user or written to the org file or to the system calendar. We want information to flow from the microphone to the org file (capturing notes), and from the file to the user and the calendar. We want to protect the password and the private subtrees of the org file. The app should only release information from private subtrees if the user has provided the correct password. Moreover, no contents from the org file should ever flow to the network or to other apps or files; it should flow only to the display, and calendar entries can flow to the Android calendar. This set of requirements constitutes a confidentiality policy with conditional declassification.[1]

An appropriate strong security property is termination-insensitive non-interference with downgrading policies that specify what part of secrets can be declassified under what conditions. (See, e.g., [8]–[10].) For example, our app declassifies the result of a password check, and has other flows conditioned on successful authentication. We consider that the adversary knows the program and can observe network- and device-visible inputs and outputs, but not covert channels like power consumption and fine timing. Since the attacker knows the program, they can learn from implicit flow. They also know the IFC mechanisms in use. We are also concerned with possible script injections in the WebView, which falls in the purview of the *gadget attacker* [11]. A gadget attacker does not have special network privileges and can only read messages directed at her own web server. She can introduce arbitrary JS code in a web page, but we assume the WebView runtime rejects code that does not comply with ECMAScript and HTML standards.

We do not trust the app, and use tools to establish its security. We do trust the Android system including WebView and its JS run-time. We assume that they do not have vulnerabilities that would allow breaking API abstractions and separation guarantees or the language semantics. We assume that the code analyzed and instrumented by our tools is installed without modification.

b) Policy enforcement: Several solutions exist for tracking information flow and identifying vulnerabilities in Android apps, e.g. [12]–[15]. We prefer static analysis of Java, for the reasons mentioned earlier. For our case study we choose SPARTA [6], an open-source tool that caters for analysts to express and check fine-grained policies. It provides a detailed model of the Android system's information sources and sinks. Policies are specified using Java (JSR 308) annotations on types in declarations, together with a *policy file* that designates which sources are allowed to flow to which sinks.

For web apps, there are several implementations of dynamic IFC [1], [2], [4], [16], [17]. Some are browser modifications [1], [17], while others are implemented by program rewriting [2], [4], [16]. There are no dynamic IFC tools designed specifically for WebView, which restricts us to program rewriting. Of the three inlined monitor systems, JEST [4] has the best combination of language and browser API support. It is also open-source.

JEST can instrument scripts in web pages and can work as an HTTP proxy server or as a command-line tool. Policy can be specified procedurally, as a JS function, or in JEST's custom declarative language (for an example, see Fig. 2). The policy assigns labels to various sources and sinks of information: network locations, user input and cookies. The interaction with the browser APIs is done via *façades* which wrap underlying APIs and augment them with information-flow tracking. We modified JEST to include a façade for the Android object of

[1]Our implementation uses Google speech-to-text, for simplicity, but this could be deployed in offline mode so we refrain from formulating a policy that allows flows to google.com.

```
RECORD_AUDIO -> FILESYSTEM, INTERNET, WRITE_CALENDAR
FILESYSTEM   -> INTERNET, WRITE_CALENDAR
INTERNET     -> WRITE_CALENDAR
INTENT       -> FILESYSTEM, INTERNET, WRITE_CALENDAR
```

Fig. 1. SPARTA policy file for the case study

JEST	SPARTA	
Principal	Sources	Sinks
sourceINTERNET	{INTERNET}	{WRITE_CALENDAR}
sourceFILESYSTEM	{FILESYSTEM}	{WRITE_CALENDAR}
sourceUSERINPUT	{USER_INPUT}	{WRITE_CALENDAR}
sinkEMPTY	{ANY}	{}

TABLE I
SPARTA↔JEST LABEL CORRESPONDENCE

WebView: when the monitored JS app loads, the JEST runtime system scans the existing Android object and recreates it as a façade with the same structure. We augmented the policy language to refer to the arguments and return values of the methods of the Android object as sources and sinks. The Android façade uses that to assign labels to results and check labels on parameters.

III. Bridging between SPARTA and JEST

a) Security policy: A SPARTA annotation is a set of *sources* paired with a set of *sinks*. Annotations can be associated with function parameters and return values, variables and class fields, as in Fig. 3. The policy file defines which annotations are valid. The policy file is a list of pairs of sets of sources and sinks, interpreted as: information from each source element in the pair is allowed to flow to each sink element in the same pair. (See Fig. 1.[2]) An individual annotation is valid if for every sink element there exist pairs in the policy that collectively list all the elements in the source element of the type. The tool checks that each annotation is valid, and of course checks that the annotations are consistent with the data flows in the program.

SPARTA annotations form a lattice, that is a product of two free lattices over the sets of sources and sinks respectively. Lattice join is defined as point-wise set union (for sources) and set intersection (for sinks). The can-flow relation (i.e., lattice order) is defined as point-wise subset inclusion (for sources) and subsumption (for sinks).

JEST policies are based on an arbitrary lattice. We choose the declarative form of specification, where labels are sets of so-called *principals* and the label lattice is a free lattice of the set of all principals in the policy. The policy maps principals to channels or principals (see Fig. 2). Information originating from each channel has a label that is the set of all the principals that map to it, in addition to all the principals that occur together with the channel on the right-hand-side. In the example, all file inputs (channel `file://`) are labeled with {`sourceFILESYSTEM`,`sinkEMPTY`}. In order to establish a relationship between our SPARTA (Fig. 1) and JEST (Fig. 2) policies, we establish a one-to-one correspondence between labels that preserves lattice ordering and joins (Table I).

We now discuss the Java functions exposed to the HTML/JS component, shown in Fig. 3. These functions define the interface between the HTML and Java components of the app. For example, requestPassword returns the password attempt entered by the user; this information is labeled with the source of USER_INPUT. The JEST policy counterpart is the channel, <−**func**://requestPassword, that denotes the result of calling Android.requestPassword() from JS. The `func://` pseudo-protocol is our extension to the JEST policy language that allows to specify policies for calls to the host environment. According to the policy, the return value is going to be assigned the sourceUSERINPUT label, which, according to Table I, corresponds to ({`USER_INPUT`}, {}), the SPARTA label on the return of requestPassword.

Our extension to the JEST policy language also allows us to talk about flows to the arguments of Java functions. For example, −>**func**://scheduleDate@1 is an output (from JS) channel that corresponds to the first argument passed in the call to Android.scheduleDate (...) . By studying the correspondence table, JEST policy and Fig. 3, one can see that the upper bound on the runtime label checked by JEST before the function call corresponds to the label on the method scheduleDate.

In addition to bridging between JEST and SPARTA enforcement, the JEST policy is also designed to help refine the policy relative to what can be expressed in SPARTA concerning the private subtrees of the org file. The principal sinkEMPTY serves as a privacy label. There are no Android output channels assigned to it, but the data read from every file gets labeled with it. This is to ensure that the private subtrees are not sent anywhere, not even to the Java component, prior to parsing and checking the password. The label is downgraded explicitly in the JS function enforcePolicy. This function parses the org-file, and filters out the subtrees tagged with `NODISPLAY` unless the password matches user's guess.

```
if (docpassword == userpassword && docpassword != ""
    || keep)  newlines += lines[k] + "\n";
```

After collapsing implicit flow due to exceptional conditions, we declassify the result to be sent to the scheduleDate and scheduleDesc functions in Java.

```
return declassify(result, "func://scheduleDate@1");
```

b) Compatibility layer: The main challenge of bridging between SPARTA and JEST stems from the fact that the former is a static analysis, while the latter enforces the policy dynamically. In JEST all values are labeled — or *boxed*, in JEST parlance. Java code, on the other hand, is compiled and run as usual; run-time values do not have labels attached to them. This means that the host component needs to supply labels for the values it provides to the WebView component to

[2]Only the flows with underlined sinks are due to the security requirements. The rest are due to the SPARTA requirement that flows in the policy be transitively closed. Capturing notes is enabled by RECORD_AUDIO −> FILESYSTEM. Loading of the org file into the HTML module of WebView is enabled by FILESYSTEM −> INTERNET. Writing events from the org file into the Android calendar is enabled by INTERNET −> WRITE_CALENDAR, and INTENT −> FILESYSTEM is for the file selection dialogue.

```
policy stringset principals {
  sourceINTERNET = dom://nonsense_source_internet, ->func://scheduleDate@1,
                   ->func://scheduleDesc@1;
  sourceFILESYSTEM = dom://nonsense_source_filesystem, ->func://scheduleDate@1,
                   ->func://scheduleDesc@1, dom://fileInput, file://*;
  sourceUSERINPUT = <-func://requestPassword;
  sinkEMPTY = dom://nonsense_sink_empty, dom://fileInput, file://*;
}
```

Fig. 2. Policy associating principles with channels in JS code, using JEST's declarative policy language

```
@JavascriptInterface public @Source ({"USER_INPUT"}) @Sink ({}) String requestPassword ()
@JavascriptInterface public void scheduleDate (@Source ({"INTERNET","FILESYSTEM"})
                                               @Sink ({"WRITE_CALENDAR"}) String date)
@JavascriptInterface public void scheduleDesc (@Source ({"INTERNET","FILESYSTEM"})
                                               @Sink ({"WRITE_CALENDAR"}) String desc)
```

Fig. 3. Signatures of the app's Java methods that can be called from JS code

fulfill the expectations of JEST. This applies to both function calls from Java to JS, where the arguments need to be labeled, as well as to the calls in the reverse direction, where the results of Java functions need to be labeled.

Labeled values flowing from WebView to Android need to be checked against the static policy in SPARTA. If the dynamic JEST label is incompatible with the static SPARTA label according to the security policy correspondence, a run-time policy violation needs to be raised.

Calls from JS to Java (Fig. 4 ①), go through the JEST façade for the Android object. The façade uses our extensions to the policy language and JEST run-time policy representation to query the labels of the parameters of the methods of the Android object. The façade checks that the label of each argument is within the upper bound dictated by the JEST policy, which corresponds to the assumptions made by SPARTA. Then it unboxes the parameters, extracting the raw JS values from the argument boxes and passes them to the corresponding method of the actual Android object (②). The WebView run-time routes the call to the Java function.

Those functions that return a value use an adapter class WebViewFacade, that we've developed (③). WebViewFacade inspects the SPARTA label on the value to be returned using Java's Reflection APIs (and a design pattern to make the annotations available), and translates it to a set of JEST channel names. It then encodes the value and channel names in JSON and returns its string representation (④). The Android object facade parses the string as JSON and extracts the value and the channel names. It boxes the return value together with a label that is the join of all the channel names and returns it to the caller (⑤). Note that since only string values can be exchanged between Java and WebView, the façades convert values to and from strings in the JSON format.

The example app does not use calls from Java to JS in WebView. However, we have experimented with ways to support those. Java Reflection can be used in a similar way to what is used for JS-to-Java calls. However, there is a complication due to the way JEST wraps inlined code in an anonymous function closure. This is done to enable JIT optimizations and to protect the monitor from adversarial JS code, while maintaining transparency (not altering secure executions) [4]. Details of the solutions explored are omitted for lack of space.

SPARTA's annotation for the WebView API is coarse grained: Java functions that call into or are called from the component are expected to be labeled with the source and sink of INTERNET, regardless of the origin of the web page. Furthermore, SPARTA does not check that the result of a function that could be called from WebView (labeled with @JavascriptInterface) has the INTERNET sink specified. To get tighter enforcement, our SPARTA policy file (Fig. 1) does not allow flow from USER_INPUT to INTERNET. The desired flows are enabled by judicious use of SPARTA's "SuppressWarnings" option in connection with the File Chooser callback, informally justified as an intended declassification.

More detail about bridging SPARTA and JEST can be found in Sexton's thesis [18].

IV. Discussion

a) Steps to secure an app: The façades we created, to interface between JS and Java in a hybrid app, are not specific to the case study. Using these façades, here are the steps needed to secure an app using our approach, once the informal requirements have been determined. (1) Write a SPARTA policy file and add SPARTA annotations, to express policy for the Java code. (2) derive the JEST lattice and channel mapping based on the SPARTA annotations on the Java functions marked as @JavascriptInterface . (3) Run SPARTA to check the Java code. (4) Run JEST on the embedded page (at build time, or via a proxy server for dynamically loaded pages).

It should not be difficult to write a translator to automate step (2) but we did not do that yet. In our experience, step (3) is not trivial. SPARTA does only intra-procedural inference of annotations, so manual annotation is needed. And there are false positives that require fine tuning, workarounds or manual overrides. Downgrade policies also need to be written explicitly, using the JEST declassification construct or documented use of "SuppressWarning" in SPARTA.

JEST also has imprecisions which manifest in false positives. The combination of `ReferenceError` exceptions that

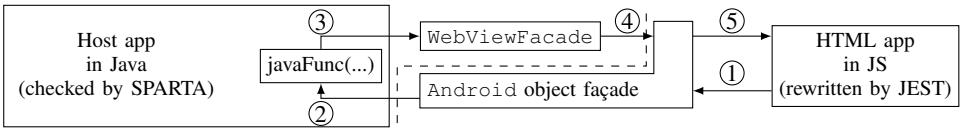

Fig. 4. Data flow in Java-WebView interactions (dashed line indicates Java versus JS)

can be thrown for any property access and the object-oriented API for string operations has given rise to a number of subtle implicit flows that have led to both policy and no-sensitive-upgrade [19] violations. JEST cannot rule out exceptions thrown by the language run-time statically, so it has to consider any potential source of them as an implicit flow. This is not an issue if the exception guards have public labels. However, our security policy is very strict about where the information is allowed to flow, so most data has non-public labels. Dealing with this problem, so that step (4) results in a usable app, has required adding several `upgrade` instructions to explicitly raise labels on variables, and `try-catch` statements to limit the impact of run-time errors and the implicit flows arising from them.

b) Security argument and implicit flow: Our aim is usable means to achieve reasonably high assurance. We would like to be able to give a security argument along the following lines. (a) The informal security goals are captured by the identified channels, their labeling, and the explicit downgrades. (b) SPARTA and JEST soundly enforce the policy on Java and JS code respectively. (c) Our technique for interoperation between SPARTA and JEST, and our specific translation of policy, ensures sound information flow tracking for executions that cross the Java/JS boundary. Unlike (a), points (b) and (c) could, in principle, be formalized. On that basis one could make a rigorous correctness argument, taking into account the declassifications, using results in [8] (for static analysis) and [20] (for monitoring).

Even as an informal security argument, our story currently has a large gap: like many other tools, SPARTA does not track implicit flow. The Pidgin [21] and Joana [22] tools for Java, both based on program dependence graphs, do handle implicit flow soundly, but are not equipped with the infrastructure needed for Android.[3] As explained by Pistoia in his state-of-the-art tutorial on information flow control [25], implicit flow checking is also implemented in IBM's static analysis tools—but it is rarely used, because for their applications there are impractically many false positives. As we noted above, significant manual effort was needed to overcome false positives due to JEST's sound implicit flow tracking. The Checker Framework on which SPARTA is built would enable modifying SPARTA to enforce the standard Denning rules for implicit flow (no low writes in high branches) [5], but one would again have to contend with false positives and explicit declassifications of intended or unavoidable implicit flows.

[3]Work is underway to make Joana applicable to Android [23]. The other Java-oriented IFC tools with implicit flow, Jif [5] and Paragon [24], are even farther from usability with Android because they make incompatible modifications to the Java language.

For implicit flow across the Java/JS boundary, our approach is to check that the Java-to-JS and JS-to-Java function calls are all in "low contexts", i.e., not control dependent on sensitive data. This we have done by manual inspection. Given a Java analyzer tracking implicit flow, it should be possible to coordinate the checks with JEST but we have not explored that in detail.

c) Other open problems: For the mapping between SPARTA and JEST policies, the label and policy translation was done by hand. It would be preferable to have a single policy file that talks about the end-to-end policy for the app and from which the policies for JEST and SPARTA are generated. JEST policy is standalone, except for declassifications in the code, but SPARTA policy is effectively split into a standalone policy file and type annotations on functions that are exposed to WebView. The latter are intertwined with the code, and thus would be hard to generate automatically from a single policy description. That's why we believe that it is most convenient to generate a JEST policy from the annotations and static SPARTA policy. There are limitations to this approach: SPARTA labels don't concern some channels that JEST supports, i.e. cookies and DOM elements, and the label on any WebView interaction is very coarse grained (`INTERNET`) and could potentially allow interacting with an arbitrary network location. In our case-study, however, the JEST policy did not talk about the additional channels, and the HTML part of the app was not expected to communicate externally at all.

One issue that deserves further attention is our use of runtime inspection of Java type annotations. This seems reminiscent of runtime monitors in which labels are a potential storage channel [26]–[28]. Since SPARTA annotations are statically associated with types we believe our façades are not susceptible to this issue, but this merits further investigation.

V. Additional related work

Two recent works deal with cross-language flows in the web setting [29], [30]. For hybrid mobile apps, the closest works are [7] and [31]. Both are built on IBM's WALA static analysis framework. Jin et al. [31] target Android mobile apps using Apache Cordova [32]. Cordova (originally PhoneGap) is a framework for building cross-platform mobile apps using HTML and JS. On Android the framework is implemented using WebView. The goal of [31] is detection and prevention of cross-site scripting vulnerabilities in such apps; the static taint analysis Actarus [33] is used for JS. To model interactions with both Cordova and Android run-times, JS mock-ups of the Cordova APIs are injected in apps to be analyzed. The apps are also transformed to make entry points discoverable

by the analyzer. HybriDroid [7] builds an accurate model of the API layer between Java and JS components of hybrid apps in Android to enable end-to-end static analysis. It offers cross-language taint analysis (i.e., explicit flow tracking), again relying on WALA. The main focus is on explaining and precisely modeling the control and data flow between the two languages and the semantics of WebView APIs. Brucker and Herzberg [34] also use WALA for static analysis, specifically for Cordova-based hybrid apps. The focus is on detecting cross-language control flow and using that capability to evaluate apps. These works inherit from WALA its limitations of handling DOM features, as well as the inherent lack of precision in static analysis for dynamic JS constructs like "`with`" for objects with statically unknown scope, and `eval`. Their focus is on the important problem of cross-site scripting vulnerability; they do not handle implicit flow or fine-grained policies with multi-level lattices and declassification.

VI. Conclusion

Despite the significant differences between SPARTA and JEST, we were able to use them together to enforce a confidentiality policy in a hybrid Android app, across components written in Java and HTML/JS. To do this, we built a compatibility layer that the components use to interact; it is responsible for adjusting value representations between the two systems and for translating labels. It uses the Java Reflection API to read SPARTA label annotations. Use of Java annotations for policy specification has advantages such as compatibility with various Java toolchains, and we believe our approach is compatible with other Java IFC tools using Java annotations. Although we described our technique in terms of a particular app, the façades are generic and can be applied to other apps without changes. To apply our approach to an app, you only need to specify the policies, use our Java façade, add SPARTA type annotations in Java components, and add downgrade and upgrade annotations in the JS components.

Acknowledgment

The authors would like to thank Felipe Fonseca for developing an early version of the org-mode app.

References

[1] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information flow control in WebKit's JavaScript bytecode," in *POST*, 2014.

[2] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: tracking information flow in JavaScript and its APIs," in *ACM SAC*, 2014.

[3] A. Almeida-Matos, J. Fragoso Santos, and T. Rezk, "A secure information flow monitor for a core of DOM," in *TGC*, 2014.

[4] A. Chudnov and D. A. Naumann, "Inlined information flow monitoring for JavaScript," in *ACM CCS*, 2015, see github.com/achudnov/jest.

[5] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *ACM POPL*, 1999, see Jif at https://www.cs.cornell.edu/jif/.

[6] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han *et al.*, "Collaborative verification of information flow for a high-assurance app store," in *ACM CCS*, 2014, see SPARTA tool at https://types.cs.washington.edu/sparta/.

[7] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for Android hybrid applications," in *31st IEEE/ACM ASE*, 2016.

[8] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *IEEE S&P*, 2008.

[9] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *IEEE CSF*, 2009.

[10] M. Vanhoef, W. DeGroef, D. Devriese, F. Piessens, and T. Rezk, "Stateful declassification policies for event-driven programs," in *IEEE CSF*, 2014.

[11] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security," in *IEEE CSF*, 2010.

[12] 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 *ACM PLDI*, 2014.

[13] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *9th Int. Conf. on Mobile Systems, Applications, and Services*, 2011.

[14] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in Android: An essential step towards holistic security analysis," in *USENIX Security*, 2013.

[15] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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," *ACM Trans. on Computer Systems*, vol. 32, no. 2, 2014.

[16] J. Santos and T. Rezk, "An information flow monitor-inlining compiler for securing a core of JavaScript," in *ICT Systems Security and Privacy Protection*, 2014.

[17] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "FlowFox: a web browser with flexible and precise information flow control," in *ACM CCS*, 2012.

[18] J. Sexton, "Information flow control for Android applications with embedded webpages," 2016, master's Thesis, see http://www.cs.stevens.edu/~naumann/pub/SextonMSthesis.pdf.

[19] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *ACM PLAS*, 2009, pp. 113–124.

[20] A. Chudnov, G. Kuan, and D. A. Naumann, "Information flow monitoring as abstract interpretation for relational logic," in *IEEE CSF*, 2014.

[21] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," in *PLDI*, 2015.

[22] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *Int. J. Inf. Sec.*, vol. 8, no. 6, 2009.

[23] M. Mohr, J. Graf, and M. Hecker, "Jodroid: Adding Android support to a static information flow control tool," in *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering*, vol. 1337, 2015.

[24] N. Broberg, B. van Delft, and D. Sands, "Paragon for practical programming with information-flow control," in *APLAS*, 2013.

[25] M. Pistoia, "Program analysis for mobile application integrity and privacy enforcement (tutorial)," in *ACM CCS*, 2015, pp. 1698–1699.

[26] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your IFCException are belong to us," in *IEEE S&P*, 2013.

[27] A. Bichhawat, "Exception handling for dynamic information flow control," in *ICSE Companion*, 2014.

[28] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: mixing static and dynamic typing for information-flow control in Haskell," in *ICFP*, 2015.

[29] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld, "JSLINQ: building secure applications across tiers," in *ACM CODASPY*, 2016.

[30] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *ACM PLDI*, 2016.

[31] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation," in *ACM CCS*, 2014.

[32] A. Foundation, "Cordova," https://cordova.apache.org/.

[33] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable JavaScript," in *ISSTA*, 2011.

[34] A. D. Brucker and M. Herzberg, "On the static analysis of hybrid mobile apps," in *ESSoS*, 2016.