Analysis of Android Inter-App Security Vulnerabilities Using COVERT

Alireza Sadeghi
Department of Computer Science
George Mason University
Fairfax, Virginia, USA
asadeghi@gmu.edu

Hamid Bagheri
Department of Computer Science
George Mason University
Fairfax, Virginia, USA
hbagheri@gmu.edu

Sam Malek
Department of Computer Science
George Mason University
Fairfax, Virginia, USA
smalek@gmu.edu

***Abstract*—The state-of-the-art in securing mobile software systems are substantially intended to detect and mitigate vulnerabilities in a single app, but fail to identify vulnerabilities that arise due to the interaction of multiple apps, such as collusion attacks and privilege escalation chaining, shown to be quite common in the apps on the market. This paper demonstrates COVERT, a novel approach and accompanying tool-suite that relies on a hybrid static analysis and lightweight formal analysis technique to enable compositional security assessment of complex software. Through static analysis of Android application packages, it extracts relevant security specifications in an analyzable formal specification language, and checks them as a whole for inter-app vulnerabilities. To our knowledge, COVERT is the first formally-precise analysis tool for automated compositional analysis of Android apps. Our study of hundreds of Android apps revealed dozens of inter-app vulnerabilities, many of which were previously unknown. A video highlighting the main features of the tool can be found at: http://youtu.be/bMKk7OW7dGg.**

I. Introduction

The ubiquity of smartphones and our growing reliance on mobile apps are leaving us more vulnerable to cyber security attacks than ever before. In this context, smartphone platforms, and in particular Android, have emerged as a topic *du jour* for security research. These research efforts have investigated weaknesses from various perspectives, including detection of information leaks, analysis of the least-privilege principle, and enhancements to Android protection mechanisms.

Despite the significant progress, such security techniques are substantially intended to detect and mitigate vulnerabilities in a single app, but fail to identify vulnerabilities that arise due to the interaction of multiple apps. Vulnerabilities due to the interaction of multiple apps, such as collusion attacks and privilege escalation chaining [21], cannot be detected by techniques that analyze a single app in isolation. Thus, there is a pressing need for security analysis techniques in such rapidly growing domains to become compositional in nature.

To address this state of affairs, this paper contributes a novel tool for compositional analysis of Android inter-app vulnerabilities, which extends and implements our previous work [1]. COVERT combines static analysis with formal analysis techniques to enable compositional security assessment of complex software. Through static analysis of application packages, it extracts relevant security specifications in a format suitable for formal analysis. Given a collection of specifications extracted in this way, a formal analysis engine (e.g., model checker) is then used to verify whether it is safe for a combination of applications—holding certain permissions and potentially interacting with each other—to be installed simultaneously.

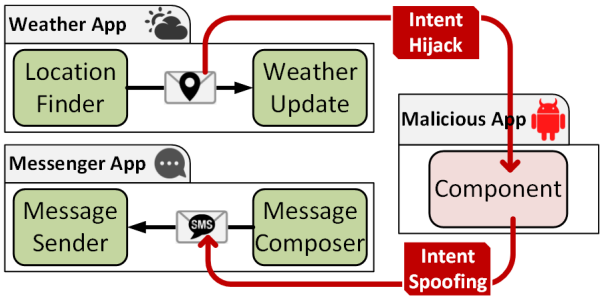

Fig. 1. A malicious app exploits the vulnerabilities of two other apps to send the device location data to the desirable phone number

COVERT further advances the current practices in assessing the inter-application vulnerabilities by providing the analysts with information that is significantly more useful than that provided by existing techniques (e.g., Fortify and IBM AppScan) that analyze the source code of an application in isolation. Our experiences with COVERT and its evaluation in the context of hundreds of real-world Android apps, collected from variety of repositories, corroborate its ability to find dozens of inter-app vulnerabilities, many of which were previously unknown.

The rest of this paper describes our analysis method, its implementation, and a summary of related work discussion.

II. Motivating Example

To motivate the research and illustrate our tool, we provide an example of a vulnerability pattern having to do with Inter-Process Communication (IPC) among Android apps. Android provides a flexible model of IPC using a type of application-level message known as *Intent*. A typical app is comprised of multiple processes (e.g., Activity, Service) that communicate using Intent messages. In addition, an app's processes could send Intent messages to another app's processes to perform actions (e.g., take picture, send text message, etc.). As an example, Figure 1 partially shows a bundle of two benign, yet vulnerable apps, installed together on a device.

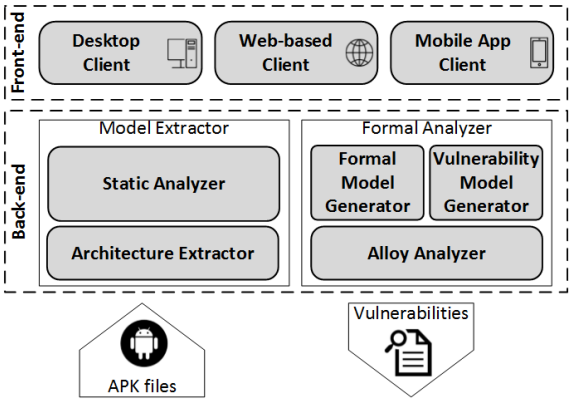

Fig. 2. COVERT's Overall Architecture.

App1 is a *Weather* application that first accesses the device location (GPS data), and then sends it to another component of the app via Intra-app Intent messaging. The Action field of the Intent is used as an address instead of explicitly addressing the Intent. This represents a common practice among Android developers [2], yet an anti-pattern that may lead to unauthorized Intent receipt.

A second vulnerability is in the *Messenger* app, where *MessageSender* uses system-level API `SmsManager`, resulting in a message to be sent to the phone number previously retrieved from the Intent. Although this app has the permission for SMS service, it fails to ensure that the sender of the original Intent message also has the permission.

Given these vulnerabilities, a malicious app can send the device location data to the desirable phone number via text message, without the need for any permission. As shown in Figure 1, the malicious app, first hijacks the Intents containing the device location info from the first app. Then, it sends a fake Intent to the second app, containing the GPS data and adversary phone number as the payload. While the example of Figure 1 shows exploitation of vulnerabilities in components from two apps, in general, a similar attack may occur by exploiting the vulnerabilities in components of either single app or multiple apps.

The above example points to one of the most challenging issues in Android security, i.e., detection of compositional vulnerabilities. What is required is a system-level analysis capability that (1) identifies the vulnerabilities and capabilities in individual apps, and (2) determines how those individual vulnerabilities and capabilities could affect one another when the corresponding apps are installed together. In the next section, we introduce COVERT that addresses these issues.

III. COVERT TOOL

To automatically detect the vulnerabilities that occur due to the interaction of a bundle of apps, we implemented COVERT tool. The input of the tool is a set of Android application package archives (APK files), and the output is a list of vulnerabilities identified in the app bundle.

As illustrated in Figure 2, COVERT tool is implemented in two layers: the back-end that performs analysis on the apps to find potential vulnerabilities, and the front-end that provides an interactive environment intended for use by the end users. This section describes the details of COVERT's components.

A. Back-end

The main components of COVERT tool that analyze the apps to detect security vulnerability issues are implemented in the back-end layer. As depicted in Figure 2, this layer consists of two modules: *Model Extractor* that leverages static analysis techniques to automatically extract an abstract formal model of Android apps, and *Formal Analyzer* that is intended to use lightweight formal analysis techniques to find vulnerabilities in the extracted app models.

1) Model Extractor: In order to automatically analyze vulnerabilities, COVERT first needs to extract a model of each app's behavior to reason about its security properties. In our approach, an app model is composed of the information extracted from two sources: manifest file and bytecode, which are processed by our *Architecture Extractor* and *Static Analyzer* modules, respectively.

Architecture Extractor examines the decoded manifest to capture the high-level architectural information of the application, including its components, their types, permissions that the app requires, and permissions enforced by each component that the other apps must have in order to interact with that component. Architecture Extractor also identifies public interfaces exposed by each application, which are entry points defined in the manifest file through *Intent Filters* of components.

After collecting architectural information Static Analyzer then extracts complementary information latent in the application bytecode. This additional information, such as Intent creation and transmission, are necessary for detecting inter-application vulnerabilities. For this purpose, COVERT utilizes different static analysis techniques to extract other essential information from the application bytecode. These techniques are briefly described as follows:

Intent Extraction: Intents are a special kind of event messages provided by Android to facilitate communication between application components. Intent messages can be used for both inter- and intra-app communications, and are thus essential information for security analysis. COVERT relies on inter-procedural data flow analysis [3] to extract the Intent information, including the sender component, the possible recipient component (in case explicitly specified), and also the Intent's *Action*, *Data* and *Categories* specifying the general action to be performed, additional information about the data to be processed by the action, and the kind of component that should handle the Intent, respectively.

Path Extraction: Existence of paths from sensitive data to statements that send it out, may cause privacy leaks. Such a path may occur within the scope of a single component or across multiple components. COVERT analyzes the app using static taint analysis technique to track sensitive data flow tuples $\langle Source, Sink \rangle$. To achieve a high precision in

```
1  sig GeneratedIntentHijack{
2    disj vulCmp, malCmp: Component, vulIntent:Intent,
3    disj vulPath, malPath:DetailedPath }{
4    vulIntent.sender = vulCmp & no vulIntent.component
5    malCmp in intentResolver[vulIntent] & no vulCmp.app & malCmp.app
6    vulCmp.app in existingApps.apps & not (malCmp.app in existingApps.apps)
7    vulPath in vulIntent.detailedPaths
8    vulPath.sink = IPC & vulPath.source in SensitiveSources
9    malCmp.detailedPaths = malPath & malPath.source = IPC
10   malPath.sink in SensitiveSinks }
```

Listing 1. Alloy specifications of Intent Hijack vulnerability in Android.

data flow analysis, COVERT's analysis is flow-, field-, and context-sensitive, meaning that it distinguishes a variable's values between different program points, distinguishes between different fields of a heap object, and that in analysis of method calls is sensitive to their calling contexts, respectively. In the interest of scalability, the analysis, however, is not path-sensitive. For single component taint analysis, COVERT relies on FlowDroid [4], but for analyzing sensitive data paths across components, it performs a formal, compositional analysis, discussed in Section III-A2.

Permission Extraction: To ensure the permission policies are preserved during an inter-component communication, one should compare the granted permissions of the caller component against the enforced permissions at the callee component side. Therefore, the permissions actually used by each component should be determined. In doing so, COVERT relies on API permission maps available in the literature, and in particular the PScout permission map [5], one of the most recently updated and comprehensive permission maps available for the Android framework. API permission maps specify mappings between Android API calls/Intents and the permissions required to perform those calls.

2) Formal Analyzer: COVERT relies on *lightweight formal analysis* techniques, and in particular *Alloy* [6] for modeling and analysis purposes. Alloy [6] is a formal specification language based on first order logic, optimized for automated analysis. The *Formal Model Generator* module of the Formal Analyzer first translates the set of app models extracted by Model Extractor into the Alloy specification language. Formal models are then combined together with a formal specification of the application framework, and checked as a whole for vulnerabilities that occur due to the interaction of apps comprising a system. For this purpose, COVERT uses the Aluminum [7] extension of the Alloy Analyzer as the analysis engine. The analysis is conducted by exhaustive enumeration over a bounded scope of model instances. Here, the exact scope of each element, such as Application and Activity, required to instantiate each vulnerability is automatically derived from the specification.

To perform the compositional analysis on a set of formal models, we designed specific Alloy signatures that model a set of security properties required to be checked. These signatures express properties that are expected to hold in the extracted specifications. Listing 1, for example, expresses the *Intent Hijack* signature. In short, the signature states that an implicit Intent (`vulIntent`) containing sensitive data retrieved from a sensitive source at a vulnerable component (`vulCmp`) could be hijacked by a malicious component (`malCmp`) that leaks this data through a sensitive sink. If a signature is satisfied, the analyzer reports it as a vulnerability, along with the information useful in finding the root cause of the violation.

Finally, the *Vulnerability Model Generator* module (recall Figure 2) refines and translates the Alloy solver results to the verification report, which is returned to the user with the detail specification of each detected vulnerability. More details on COVERT's back-end are described in [1].

B. Front-end

In order to facilitate the end-user interactions with COVERT back-end engine, we implemented client applications for different platforms: *Desktop Application*, which is a stand-alone tool that calls back-end components and visualizes the generated results. *Mobile* and *Web-based* applications that work together to analyze the installed apps in a mobile device and show the vulnerability report on web browsers.

Desktop Client is a JavaFX [8] application that provides a graphical user interface and enables end-users to analyze a set of APK files, which could be downloaded from online app stores such as Google Play or grabbed from their own mobile devices using adb [9] tool. The features of this application, which is available in COVERT's web page [10], are described in more detail at the end of this Section.

As an alternative client support, *Mobile* and *Web-based* applications work together to analyze the installed apps in a mobile device and generate the vulnerability report, without the need for directly providing the app bundles to the back-end engine. Mobile app, on the one hand, runs on a mobile device and retrieves the information of installed apps on the same device, including the package name, and the version code. This information, along with the device's identifier, are then sent to the back-end server, where the identified apps are downloaded and analyzed. On the other hand, the user can access the analysis results via the web-based application, by providing the device identifier as the access key.

To achieve a high level of scalability, a central repository of app modelsis maintained in the back-end server. Thereby, before extracting each app model, which is an expensive task, COVERT first searches the central repository for that app, by using the combination of its ⟨package name, version code⟩ as the key. If the app model already exists in the repository, the existing model is reused for the analysis.

In the following, we illustrate some key features of our tool through a real-world example. Figure 3 shows a snapshot of COVERT's front-end desktop application after loading the results of back-end analysis for sample apps from our experimental collection. In part (a), the detected vulnerabilities are categorized based on inter-component vulnerability classes identified by prior research [2], [11]: Intent Hijack, Intent Spoofing, Inter/Intra-app data leakage, privilege escalation, etc.. Part (b) represents the elements involved in a particular instance of each vulnerability in a hierarchical structure. Here,

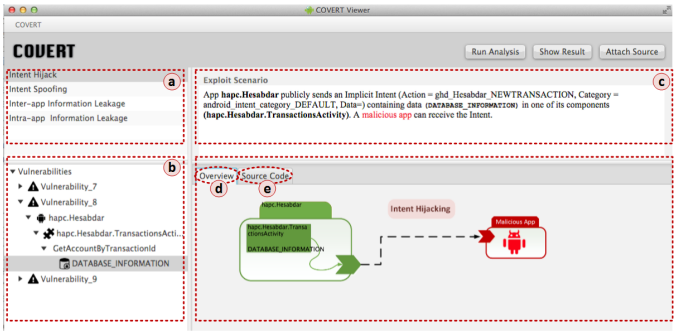

Fig. 3. A Snapshot of COVERT's Desktop Client Application: (a) vulnerability categories (b) detail elements of vulnerabilities (c) a potential exploit scenario (d) graphical overview of the exploit scenario (d) decompiled source code of vulnerable component.

for example, the expanded vulnerability is an instance of Intent Hijack, detected in *Hesabdar* app[1]. Parts (c) and (d) then describe details of the detected vulnerability using both text and graphical notations. As narrated and visualized in Figure 3(c) and (d), The Hesabdar's `TransactionsActivity` component handles user account information and sends the information as payload of an implicit Intent to another component. When a component sends an implicit Intent, there is no guarantee that it will be received by the intended recipient. A malicious application can intercept an implicit Intent simply by declaring an Intent filter with all of the actions, data, and categories listed in the Intent, thus stealing sensitive account information by retrieving the data from the Intent.

Finally, to enable security analyst to inspect the vulnerable apps more carefully, the decompiled source code of the vulnerable component(s) is shown in part (e) (this tab is not visible in the snapshot illustrated in Figure 3).

IV. Related Work

Since the emergence of Android platform, dozens of security analysis tools have been developed for mobile apps. Most related tools to COVERT are the ones with the focus on performing static program analysis over Android applications.

Although several tools [2], [12], [13] have been developed to identify inter-component vulnerabilities, but they do not consider inter-app security issues. COVERT's analysis, however, goes far beyond single application analysis, and enables compositional analysis of the overall security posture of a system, greatly increasing the scope of vulnerability analysis.

DidFail [14], perhaps most closely related, introduces an approach for tracking data flows between Android components. It leverages Epicc [12] for Intent analysis, but consequently shares Epicc's limitation of not covering data scheme, which negatively affects the precision of this approach in inter-component path matching. Moreover, DidFail is a purely program analysis tool, and does not incorporate a formal analysis technique.

[1]Hesabdar is an accounting app for personal use and money transaction that, among other things, manages account transactions and provides a temporal report of the transaction history.

V. Conclusion

This paper presents COVERT, a tool that analyzes Android applications in a compositional manner to detect inter-app and inter-component security vulnerabilities. COVERT, at its core, consists of a back-end engine that extracts formal model of apps and analyzes the extracted models together to find the vulnerabilities. On top of the core, COVERT comes with desktop, mobile and web-based front-end applications that facilitate the end-user interactions with the analysis engine. The experimental results corroborated its ability to reveal inter-app vulnerabilities in real-world Android apps, many of which were previously unknown.

VI. Acknowledgements

This work was supported in part by awards D11AP00282 from the US Defense Advanced Research Projects Agency, H98230-14-C-0140 from the US National Security Agency, HSHQDC-14-C-B0040 from the US Department of Homeland Security, and CCF-1252644 from the US National Science Foundation.

References

[1] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app vulnerabilities," *George Mason University, Tech. Rep. GMU-CS-TR-2015-1*, 2015.
[2] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of MobiSys*, 2011, pp. 239–252.
[3] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of SOAP*. ACM, 2012, pp. 3–8.
[4] 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 *Proceedings of PLDI*, 2014.
[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of CCS*, 2012.
[6] D. Jackson, "Alloy: a lightweight object modelling notation," *TOSEM*, vol. 11, no. 2, pp. 256–290, 2002.
[7] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Aluminum: Principled scenario exploration through minimality," in *Proceedings of ICSE*, 2013, pp. 232–241.
[8] "Javafx." [Online]. Available: www.goo.gl/nWyRWv
[9] "Android debug bridge." [Online]. Available: www.goo.gl/06Ee9u
[10] "Covert website," www.sdalab.com/tools/covert.
[11] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *the 13th Intl. Conf. on Information security*, 2010.
[12] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis," in *Proceedings of USENIX Security*, 2013.
[13] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of CCS*, 2012.
[14] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of SOAP*, 2014, pp. 1–6.