Automated Detection and Mitigation of Inter-application Security Vulnerabilities in Android (Invited Talk)

Sam Malek
Computer Science Dept.
George Mason University
Fairfax, VA, USA
smalek@gmu.edu

Hamid Bagheri
Computer Science Dept.
George Mason University
Fairfax, VA, USA
hbagheri@gmu.edu

Alireza Sadeghi
Computer Science Dept.
George Mason University
Fairfax, VA, USA
asadeghi@gmu.edu

ABSTRACT

Android is the most popular platform for mobile devices. It facilitates sharing data and services between applications by providing a rich inter-application communication system. While such sharing can be controlled by the Android permission system, enforcing permissions is not sufficient to prevent security violations, since permissions may be mismanaged, intentionally or unintentionally, which can compromise user privacy. In this paper, we provide an overview of a novel approach for compositional analysis of Android inter-application vulnerabilities, entitled COVERT. Our analysis is modular to enable incremental analysis of applications as they are installed on an Android device. It extracts security specifications from application packages, captures them in an analyzable formal specification language, and checks whether it is safe for a combination of applications—holding certain permissions and potentially interacting with each other—to install simultaneously. To our knowledge, our work is the first formally-precise analysis tool for automated compositional analysis of Android applications.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Security, Verification

Keywords

Android, Mobile Security, Program Analysis

1. INTRODUCTION

Mobile app markets are creating a fundamental paradigm shift in the way software is delivered to the end users. The benefits of this software supply model are plenty, including the ability to rapidly and effectively acquire, introduce, maintain, and enhance software used by the consumers. Application frameworks are the key enablers of these markets. An application framework, such as the one provided by Android, ensures apps developed by a wide variety of suppliers can interoperate and coexist together in a single system (e.g., a phone) as long as they conform to the rules and constraints imposed by the framework.

This paradigm shift, however, has given rise to a new set of security challenges. In parallel with the emergence of app markets, we are witnessing an increase in the security threats targeted at mobile platforms. This is nowhere more evident than in the Android market (i.e., Google Play), where many cases of apps infected with malwares and spywares have been reported [11]. In this context, Android's security has been a thriving subject of research in the past few years. Leveraging program analysis techniques, these research efforts have investigated weaknesses from various perspectives, including detection of information leaks [4, 6, 9], analysis of the least-privilege principle [2, 7], and enhancements to Android protection mechanisms [3, 5, 8]. The majority of these approaches, however, are subject to a common limitation: they are 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 [3], cannot be detected by techniques that analyze a single app in isolation. Thus, security analysis techniques in such domains need to become compositional in nature, enabling one to reason about the overall security posture of a system (e.g., a phone device) in terms of the security properties inferred from the installed apps.

In our research, we are developing a novel approach, called COVERT, for automated compositional analysis of systems built on top of application frameworks. We use Android as an example framework to illustrate and evaluate our research. At the heart of our approach is a modular static analysis technique for Android apps, designed to enable incremental and automated verification of apps as they are installed, removed and updated on an Android device. Through static analysis of each app, our approach extracts essential information and captures them in an analyzable formal specification language. These formal specifications are intentionally at the architectural level to ensure the technique remains scalable, yet represent the true behavior of the implemented software, as they are automatically extracted from the code. We then use formal analysis techniques to verify certain properties (e.g., known security vulnerability patterns) in the extracted specifications. The rest of this paper provides an overview of our approach.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DeMobile'14 , November 17, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3225-5/14/11 ...$15.00.

2. APPROACH OVERVIEW

Figure 1 shows an overview of our approach which has two steps: 1) *Model Extractor* that uses static code analysis techniques to elicit formal specifications (models) of the apps comprising a system; and 2) *Formal Analyzer* that uses lightweight formal analysis techniques to verify certain properties (e.g., known security vulnerability patterns) in the extracted specifications.

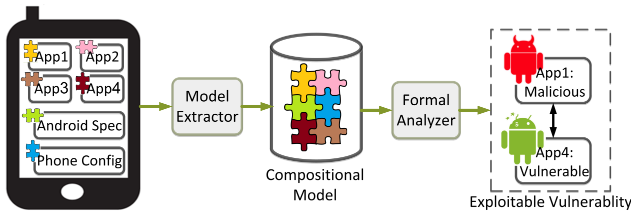

Figure 1: A high-level overview of COVERT.

Our approach relies on three types of models: 1) *app model* that Model Extractor generates automatically for each Android app; 2) *Phone configuration model* that describes the properties of a certain Android device; and 3) *Android framework spec.* that defines a set of rules to lay the foundation of Android apps, how they behave (e.g., application, component, messages, etc.), and how they interact with each other. This specification is constructed once for a given framework (e.g., Android, iPhone) as a reusable model to which all extracted app models must conform. It can be considered as an abstract specification of how a given framework behaves.

To generate the app models, Model Extractor takes as input a set of Android application package archives (APK files). Each Android app has a manifest file that represents its configuration specification. To parse the manifest files, Model Extractor first disassembles APK files using the publicly available *ApkTool* tool [1]. It then examines the app manifest file to determine the involved components, their types, *Intent Filters* that declare the kinds of Intents each app component accepts, permissions that the app requires, and permissions enforced by each component that the other apps must have in order to interact with that component.

Besides such high-level, architectural information collected from the manifest file, Model Extractor utilizes static analysis techniques to extract other essential information from the APK files. Our model extractor is built on top of the *Soot* static analysis tool developed for analyzing Java bytecode [12].

Application interactions in Android occur through Intent messages. An Intent message is an event for an action to be performed along with the data that supports that action. Model Extractor analyzes disassembled outputs to examine Intent creation and transmission. It also examines whether the intent-receiver components actually use any API that needs the permission the sender lacks, which in turn, may lead to security issues, such as privilege escalation. The component, Intent, Intent filter and permission elements are tracked and represented such that they have all the necessary attributes required to detect inter-application vulnerabilities. The extracted information are represented as analyzable formal specifications. For each Intent message, for example, our model extractor tracks the following information: (1) its sender, (2) the target component, (3) the type of *action* (if any) it has, (4) *data* to be processed by the action, and (5) *categories* of component that should handle the Intent.

The set of app models extracted in this way are then combined together and with the Android framework spec. and device configuration model, and checked as a whole for vulnerabilities that occur due to the interaction of apps comprising a system. Finally, the analysis report is returned to the user with the list of vulnerabilities.

We use *Alloy* as a specification language [10], and the Alloy Analyzer as the analysis engine. Alloy is a formal modeling language with a comprehensible syntax that stems from notations ubiquitous in object orientation, and semantics based on the first-order relational logic [10]. Alloy Analyzer is a constraint solver that supports automatic analysis of models written in Alloy. The analysis process is based on a translation of Alloy specifications into a Boolean formula in conjunctive normal form (CNF), which is then analyzed using off-the-shelf SAT solvers.

The analyzer provides two types of analysis: *Simulation*, in which the analyzer demonstrates consistency of model specifications by generating a satisfying model instance; and *Model Checking*, which involves finding a counterexample—a model instance that violates a particular assertion. We use the former to compute model instances, represented as satisfying solutions to the combination of models captured from app implementations. This shows the validity of such extracted models, confirming that the extracted models are self-consistent, mutually compatible and consistent with the Android specifications modeled in a separate module. To carry out the verification analysis, we use the latter. To that end, we develop assertions that model a set of security properties required to be checked. These assertions express properties that are expected to hold in the extracted specifications. If an assertion does not hold, the analyzer reports it as a counterexample, along with the information useful in finding the root cause of the violation.

Our approach can be applied in an offline setting to determine if a particular configuration for a system comprised of several apps harbors security vulnerabilities. The approach could also be applied at runtime to continuously verify the security properties of an evolving system as new apps are installed, and old ones are updated and removed. To that end, we expect lightweight-monitoring services would be deployed on the mobile devices to collect the necessary data and ship it for analysis to an analysis engine running on a backend server (or cloud). In cases where vulnerabilities are detected, mitigation strategies could be carried through actuator services deployed on mobile devices. Such actuators may be implemented in different ways. A simple mitigation strategy is to prevent the installation of apps or remove apps that create security vulnerabilities when deployed with certain other apps. An alternative, perhaps more complex, mitigation strategy is to employ a dynamically enforaceable approach, such as restricting communications between certain apps.

3. REFERENCES

[1] android-apktool - a tool for reverse engineering android apk files. https://code.google.com/p/android-apktool/.

[2] Au, K. W. Y. et al. Pscout: Analyzing the android permission specification. In *Proc. of CCS'12* (2012).

[3] Bugiel, S. et al. Towards taming privilege-escalation attacks on android. In *Proc. of NDSS* (2012).

[4] Chin, E. et al. Analyzing inter-application communication in android. In *Proc. of MobiSys* (2011).

[5] Dietz, M. et al. Quire: Lightweight provenance for smart phone operating systems. In *Proc. of USENIX* (2011).

[6] Enck, W. et al. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of USENIX OSDI* (2011).

[7] Enck, W. et al. On lightweight mobile phone application certification. In *Proc. of CCS* (2009).

[8] Fragkaki, E. et al. Modeling and enhancing android's permission system. In *Proc. of ESORICS* (2012).

[9] Hornyack, P. et al. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proc. of CCS* (2010).

[10] Jackson, D. Alloy: a lightweight object modelling notation. *TOSEM 11*, 2 (2002), 256–290.

[11] Shabtai, A. et al. Google android: A comprehensive security assessment. *Security & Privacy, IEEE 8*, 2 (2010), 35–44.

[12] Valle é-Rai, R. et al. Soot - a java bytecode optimization framework. In *Proc. of CASCON'99* (1999).