2015 IEEE Symposium on Security and Privacy

Effective Real-time Android Application Auditing

Mingyuan Xia
McGill University
mingyuan.xia@mail.mcgill.ca

Lu Gong, Yuanhao Lyu, Zhengwei Qi
Shanghai Jiao Tong University
{*iceboy, 185662, qizhwei*}*@sjtu.edu.cn*

Xue Liu
McGill University
xueliu@cs.mcgill.ca

Abstract

Mobile applications can access both sensitive personal data and the network, giving rise to threats of data leaks. App auditing is a fundamental program analysis task to reveal such leaks. Currently, static analysis is the *de facto* technique which exhaustively examines all data flows and pinpoints problematic ones. However, static analysis generates false alarms for being over-estimated and requires minutes or even hours to examine a real app. These shortcomings greatly limit the usability of automatic app auditing.

To overcome these limitations, we design AppAudit that relies on the synergy of static and dynamic analysis to provide effective real-time app auditing. AppAudit embodies a novel dynamic analysis that can simulate the execution of part of the program and perform customized checks at each program state. AppAudit utilizes this to prune false positives of an efficient but over-estimating static analysis. Overall, AppAudit makes app auditing useful for app market operators, app developers and mobile end users, to reveal data leaks effectively and efficiently.

We apply AppAudit to more than 1,000 known malware and 400 real apps from various markets. Overall, AppAudit reports comparative number of true data leaks and eliminates all false positives, while being 8.3x faster and using 90% less memory compared to existing approaches. AppAudit also uncovers 30 data leaks in real apps. Our further study reveals the common patterns behind these leaks: 1) most leaks are caused by 3rd-party advertising modules; 2) most data are leaked with simple unencrypted HTTP requests. We believe AppAudit serves as an effective tool to identify data-leaking apps and provides implications to design promising runtime techniques against data leaks.

***Keywords*-approximated execution; program analysis; privacy; mobile application;**

I. Introduction

In recent years, mobile devices have gained unprecedented success and become the most popular personal consumer electronics. Users store all kinds of personal data on these devices, e.g., text messages, call logs, locations, and browsing history. Mobile applications (or apps for short) can deliver rich functionalities and improve services by properly using these personal data. However, recent studies unveil many abuses of these data, which lead to data leaks intentionally [1], [2] (e.g. for improper advertising revenue) or unintentionally (e.g. exposing these data in plain-text over public networks [2]).

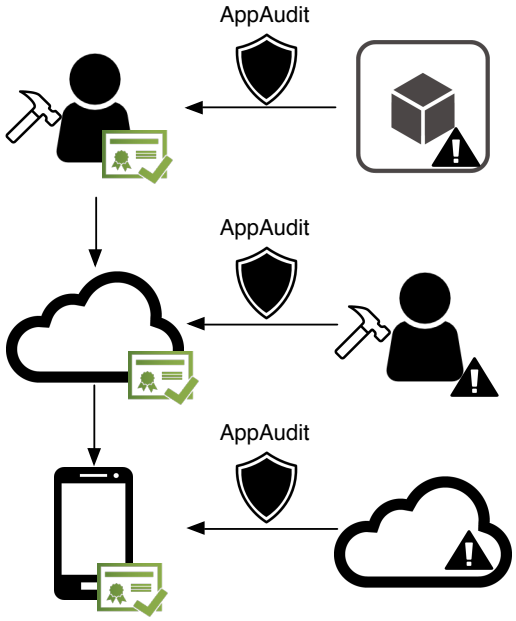

Figure 1: AppAudit use cases. AppAudit aims to prevent data-leaking apps from being produced, distributed and installed.

Data leaks tamper user privacy, which drives users to abandon apps, harming app developers as well as the app market. To address this crucial problem, market operators have been actively developing techniques to analyze and identify data-leaking apps, i.e., app auditing. Static program analysis [3], [4], [5], [6], [7] can comprehensively examine program data flows and reveal data-leaking code paths, which is the *de facto* technique for app auditing. However, static analysis is generally inefficient (time- and memory-consuming) and produces false alarms. Market operators have to spend great computing power to run such analysis and further invest human efforts to validate the results.

In this paper, we propose AppAudit, a program analysis framework that can analyze apps efficiently (in real-time) and effectively (report actual data leaks). Figure 1 demon-

DOI 10.1109/SP.2015.60

strates the three use cases of AppAudit. First, AppAudit can be integrated into IDEs to check apps for developers before release. This helps to identify problematic 3rd-party modules, which are the main causes of data leaks [1]. Second, AppAudit can be deployed as an automatic app auditing service at app markets. AppAudit's high accuracy helps market operators to wipe out human involvement in validating analysis results and thus fully automates app auditing procedure. AppAudit's high efficiency greatly reduces the waiting time for developers to get auditing feedback from the market after they upload apps. Third, AppAudit can be installed on mobile devices to check apps before installation. As Android allows users to install apps from any market and developer, AppAudit can protect users against data-leaking apps from untrusted sources or app markets that lack auditing service.

To achieve these goals, AppAudit relies on the synergy of a new dynamic analysis and a lightweight static analysis. AppAudit works with two stages. At the first stage, AppAudit performs an efficient but over-estimating static API analysis to sift out suspicious functions. The static analysis is lightweight at the cost of reporting false positives. Then at the second stage, we propose *approximated execution*, a dynamic analysis that can simulate the execution of a program while perform customized checks at each program state. The dynamic analysis executes each suspicious function, monitors the dissemination of sensitive data and reports data leaks that can happen in real execution. AppAudit relies on this analysis to prune false positives from the static stage. Previous pure dynamic analysis [8] fail to automatically explore code paths in depth due to the presence of unknown values, resulting in lower code coverage and more false negatives than static analysis. Our dynamic analysis overcomes this shortcoming with an innovative object model to represent unknown values and mechanisms to handle execution with unknowns.

Our contribution is three-fold:

- We propose approximated execution, a novel dynamic analysis that can execute part of a program while performing customized checks on its program state at each step. The executor can faithfully simulate actual program execution and function with the presence of unknowns.
- We present AppAudit, an Android app auditing tool that can check apps effectively and efficiently. AppAudit embodies an API analysis to select suspicious functions and then relies on the approximated executor to prune false positives. Our experiments show that AppAudit achieves comparable code coverage with static analysis and produces no false positives with significantly less time and memory.
- We apply AppAudit to examine more than 400 free Android apps collected from various markets. Our tool successfully identifies 30 data leaks in these apps and their containing modules. We also uncover that 3rd-party advertising libraries are the major causes of data leaks and HTTP requests are the most prominent leaking venue.

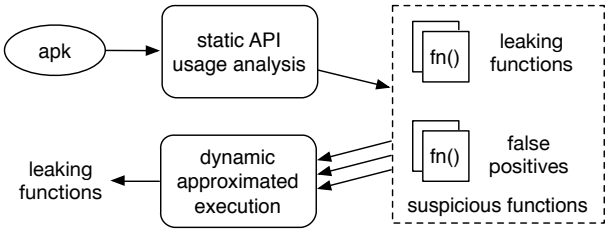

Figure 2: AppAudit architecture and workflow.

The rest of the paper is organized as follows. Section II presents the design overview. Section III elaborates our static API analysis. Section IV elaborates our innovative execution engine for dynamic analysis. Section V evaluates the accuracy and performance of AppAudit and presents our findings on real free apps. Section VI introduces the related work and Section VII concludes the paper.

II. AppAudit Design Overview

The app auditing service intends to find code paths that leak sensitive user data. Mobile apps nowadays grow larger and more complicated, with many 3rd-party libraries and thousands of functions. Static analysis can encounter scalability problems for large code base, because of non-scalable analysis structures, such as precise flow graph or heavy analyses such as points-to analysis and symbolic execution. As a result, static analysis is generally time-consuming, especially with large real applications. Meanwhile, static analysis could generate false alarms because some analyzed code paths could never happen in real execution. These limitations greatly confine the use cases of static analysis.

To tackle false positives and analysis efficiency, we start with a very lightweight static API analysis and rely on a dynamic analysis to prune its false positives, as shown in Figure 2. The API analysis aims to sift out suspicious functions and narrow down the analysis scope. Then AppAudit largely depends on the dynamic analysis to execute the bytecode of each function to confirm actual data leaks. Multiple suspicious functions can be examined in parallel to improve performance. Compared with pure static analysis solutions, AppAudit only explores code paths that could happen in real, thus generating few false positives. The major challenge of dynamic analysis is caused by unknown values during the analysis. When dynamic analysis meets unknowns, it can hardly explore deeply into code paths, which will cause false negatives. To overcome this, we design a novel object model to represent and propagate unknowns. We also design several execution mechanisms to increase the depth of our analysis and avoid false negatives.

III. Efficient Static API Analysis

The goal of the static API analysis is to find functions that can potentially cause data leaks. Overall, static analysis is over-estimating and AppAudit relies on a dynamic analysis to prune its false positives. In this section, we focus on tuning the static API analysis for improved performance.

A. Call Graph Extensions

A conventional call graph models the calling relationships between functions. A function can reach a particular API if there exists a path from the function to the API. To leak data, a function must be able to reach a *source API* that retrieves personal data and a *sink API* that transmits data out of the device. Thus, finding data leaks is equivalent to finding one path from the function to a source API and another to a sink API. Dynamic Java language features and the Android programming model can result in missing paths in a conventional call graph. Thus, AppAudit incorporates series of call graph extensions to capture the following cases:

Java Virtual Calls and Reflection Calls. In Java, a virtual call can have many call targets (base class methods or derived class methods) and a reflection call can essentially reach an arbitrary function in the program. In both cases, the actual call target depends on the runtime calling context which is not visible to static analysis. In our static call graph, we assume that virtual calls can reach any matching method from all inherited classes while a reflection call will directly be marked suspicious. This is a simple (thus efficient) but over-estimating heuristic. Though more precise heuristics exist [9], AppAudit aims to postpone fine-grained assessment to the dynamic analysis.

Static Fields as Intermediates. It is very common that two functions exchange sensitive data via a static field. In such cases, one function will indirectly call a source API and the other will call a sink API. To complete this colluding procedure, there must be a third function that calls both in order. Thus in the call graph, this third function will be marked suspicious and examined by the dynamic analysis.

Android Life Cycle Methods. An Android app interacts with the system by exposing a set of life cycle methods. When the user navigates across the app, the Android system invokes these life cycle methods in a particular order. In our call graph, we create a dummy node that simulates these ordered function invocations. If the app leaks data via life cycle methods, this dummy node will be marked as suspicious and the dynamic analysis can examine the life cycle methods in order.

Multi-threading. Multi-threading is a common programming practice in Android apps. A common idiom is to retrieve some data in the main thread and then spawn a child thread to send it via the network. In a conventional call graph, the retrieving function does not directly call the sending function. To tackle this discontinuity, we treat the function that registers a callback as calling the callback directly. In addition to standard Java multi-threading support, we also apply this technique to two Android-specific asynchrony constructs (`AsyncTask` and `Handler`).

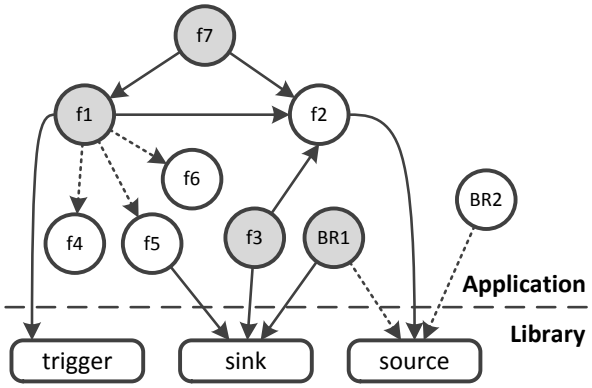

Figure 3: An extended call graph. Each vertex stands for a function. Solid lines represent traditional call relationships and dashed lines stand for extended calls. Grey vertices are the marked suspicious functions. BRs stand for BroadcastReceivers that can receive system events.

GUI Event Callbacks. Android apps heavily utilize all kinds of GUI widgets. These widgets rely on various call-back functions to respond to different user actions. We apply the technique used in the case of multi-threading to handle these GUI call-back functions.

Android Remote Procedure Call (RPC). Android provides a system-wise RPC mechanism to notify apps of various system events. Apps can send messages to each other through the same mechanism. Messages are encapsulated in *intents*. Some intents might contain sensitive user data. For example, when receiving an incoming SMS, the Android system will generate an intent with the content of the SMS and send it to apps of interest. An app declares a special class called `BroadcastReceiver` in its manifest file to receive intents. In our analysis framework, we treat all BroadcastReceivers that can handle sensitive intents as calling a dummy source API to retrieve sensitive data.

The first three cases are handled in an ad-hoc manner when constructing the call graph. The rest three all involve call-back functions so that we design a unified mechanism. We define those APIs that can register call-back functions as *trigger APIs*. Each trigger API can register a specific type of callbacks. In our call graph, if a function calls a trigger API, then this function will be treated as calling all possible callback functions of that type. Table I provides a partial list of the trigger APIs currently used in AppAudit. For example, `Context.startService()` registers a callback with the Android system to invoke the life cycle functions of a `Service` class. Thus if a function calls `startService()`, we treat it as calling the `onCreate()` function of all classes that inherit the

Category	Trigger API	Extended function calls

Android RPC	`Context.startService()`
	`Context.startActivity()`
	`Context.sendBroadcast()`
	`AlarmManager.setRepeating()`
	... and 4 more
		`u.onCreate()`, $\forall u$ extends Service
		`u.onCreate()`, $\forall u$ extends Activity
		`u.onReceive()`, $\forall u$ extends BroadcastReceiver
		all the three above

GUI Callbacks	`setOnClickListener()` ... and 180 more [10]	`u.onClick()`, $\forall u$ extends OnClickListener

Multi-threading	`Thread.start()`
	`AsyncTask.execute()`
	... and 14 more
		`u.run()`, $\forall u$ extends Thread
		`u.doInBackground()`, $\forall u$ extends AsyncTask

Table I: Trigger APIs and extended function calls.

`Service` class.

B. API Usage Analysis

Checking whether a given function is suspicious is equivalent to finding a path from the function to a source API and a path to a sink API. We first build a standard call graph from program bytecode and then extend it with dummy functions and extra calling relationships according to above-mentioned cases. To accelerate the construction algorithm, we omit Android library functions except for source, sink and trigger APIs. We want to focus on application functions and avoid analyzing the Android runtime library. After the extended call graph is constructed, we perform a breadth-first search to mark all suspicious functions. For example, with the extended call graph in Figure 3, the static API analysis can reveal four suspicious functions (BR1, f1 and f7, f3) while a conventional call graph can only reveal f3.

Overall, the extended call graph is an over-approximated call graph with calling relationships that will not happen in real execution. Consequently, our static API analysis could mark "good" functions as suspicious in trade for the analysis performance. While previous work [9] employs more complicated analyses to achieve better heuristic at the cost of performance, AppAudit takes an opposite direction and relies on dynamic analysis to prune false positives.

IV. Approximated Execution

The static API analysis is over-approximating, which could result in false positives. We use a dynamic analysis to confirm actual data leaks and prune false positives.

The approximated executor is a dynamic analysis that executes the bytecode instructions of a suspicious function and reports if sensitive data could be leaked during the execution. The executor has a typical register set, a program counter (pc), a call stack as its execution context. It relies on a novel object model to represent application memory objects. The executor has three working modes, as shown in Figure 4. It starts with "execution (exec)" mode, where it interprets bytecodes and performs operations. Source APIs can generate sensitive data objects, where we mark them as "tainted". Tainted objects propagate with the execution and taint any object that is derived from them. Whenever the executor encounters a sink API, it changes to "check" mode to check the parameters for the sink API. If tainted objects are found, the executor reports the leak and terminates ("end" final state). Otherwise, it reverts back to the normal execution mode. When certain bytecode instruction cannot be executed due to unknown operands (e.g. a conditional jump instruction with unknown condition), the executor switches to "approximation (approx)" mode for approximations to continue the execution. If the approximations fail, commonly due to too many unknowns or insufficient execution contexts, the executor will terminate the execution of current function and start executing one of its caller function ("leap" final state). The caller function is expected to provide a more concrete execution context to analyze the incomplete execution.

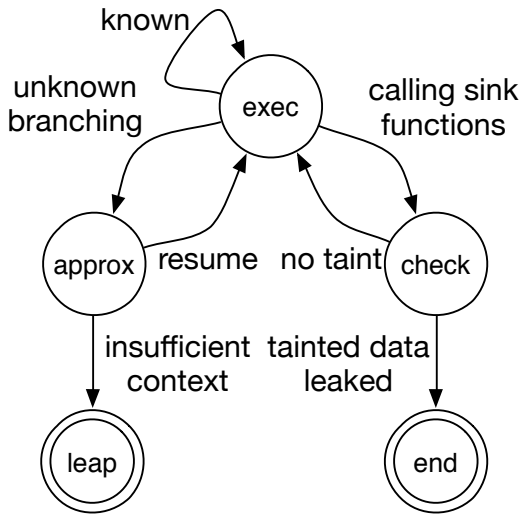

Figure 4: AppAudit approximated executor state machine.

A. Object and Taint Representation

The executor starts from the function entry with the absence of its calling context (the values of parameters and global variables). We design an object model to represent