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]).

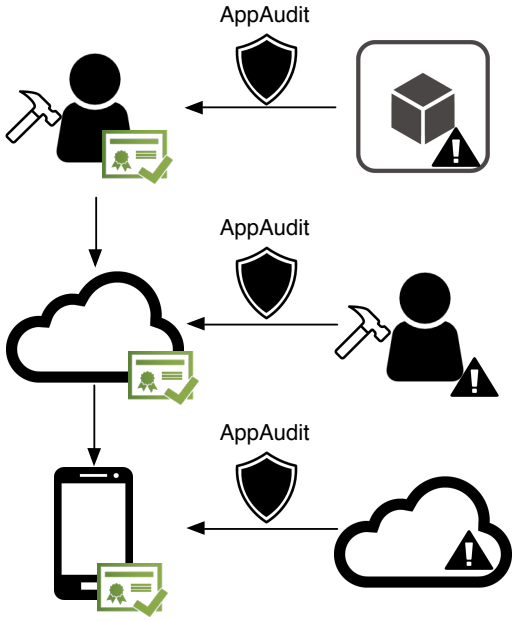

Figure 1: AppAudit use cases. AppAudit aims to prevent data-leaking apps from being produced, distributed and installed.

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.

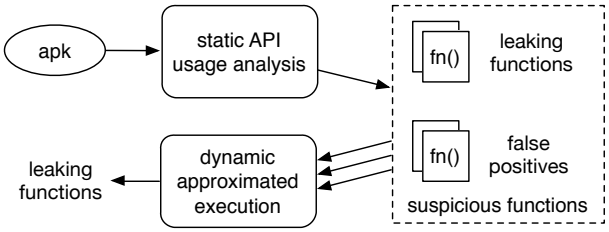

Figure 2: AppAudit architecture and workflow.

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`).

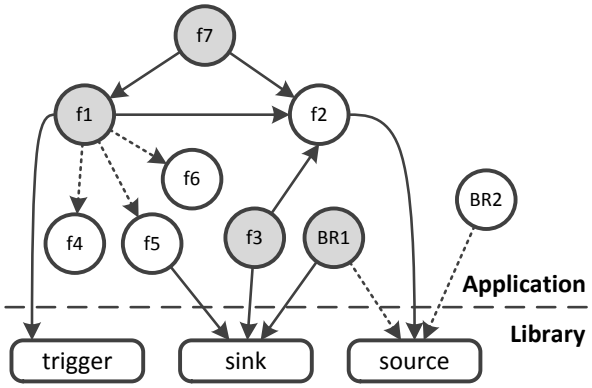

Figure 3: An extended call graph. Each vertex stands for a function. Solid lines represent traditional call relationships and dashed lines stand for extended calls. Grey vertices are the marked suspicious functions. BRs stand for BroadcastReceivers that can receive system events.

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.

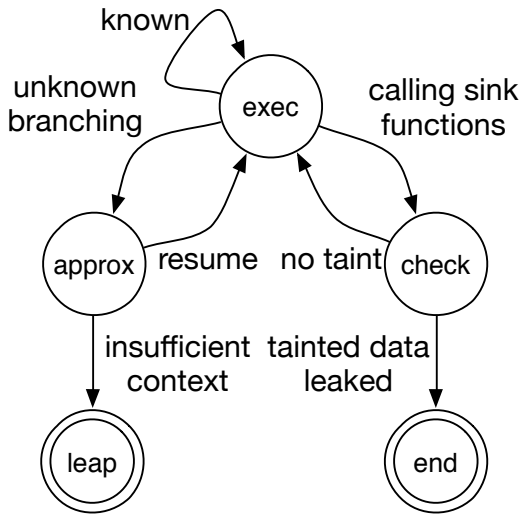

Figure 4: AppAudit approximated executor state machine.

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

and tackle unknowns. A memory object of the application is represented as a tuple $\phi(x) := \langle \phi_T(x), \phi_K(x), \phi_V(x) \rangle$. $\phi_T(x)$ specifies its type, which can be Java primitive types (e.g., int, long, char) as well as class types (e.g. String, StringBuilder). AppAudit introduces object kind $\phi_K(x)$ to distinguish known values and unknowns. $\phi_K(x)$ can be one of the following cases: 1) a *concrete object (CON)* that is created during the execution process, e.g. an object created by the `new` instruction; 2) a *prior unknown (PU)*, which exists prior to the execution process and contains no known values to the executor, e.g. a global variable; 3) a *derived unknown (DU)*, which was a prior unknown but is changed during the execution process. DUs mix known values and unknowns. For instance, a DU could have some known fields and some unknown fields. $\phi_V(x)$ stores the known value(s) of the object. For primitive types, $\phi_V(x)$ reflects its known value, e.g. an integer of value 5 is represented as $\phi_V(x) = \{val \mapsto 5\}$. If the value is unknown, $\phi_V(x) = \emptyset$. For class types, $\phi_V(x)$ stores all its known fields, e.g. $\phi_V(x) = \{field1 \mapsto \phi(y)\}$ representing $x.field1 == y$. Unknown fields will not appear. Arrays are special objects with indices as fields, e.g. an array of two elements is represented as $\phi_V(x) = \{0 \mapsto \phi(y), 1 \mapsto \phi(z), length \mapsto 2\}$. $\phi_V(x)[field]$ can query a particular filed of an object x. If the field is known, this query returns the known object. Otherwise, this expression returns a prior unknown.

In addition to our object representation, AppAudit also tracks taints on objects similar to dynamic taint analysis [11], [12]. For each memory object x, we define $\tau(x)$ as its tainting state. Each source API could generate a specific type of taint, representing a particular type of personal data (e.g. text message, location, etc). Taints propagate along with the object. Any object derived from a tainted object will also be tainted. If a sink API meets a tainted object, our executor will report a leak. We will explain our tainting rules in details after introducing the execution rules.

B. Basic Execution Flow

We use five examples to demonstrate the basic workflow of the executor and the expressiveness of our object representation. We assume that the `source()` API generates a tainted integer (denoted as *taint*) and the `sink()` API checks if its parameter is tainted. All parameters and global variables (static class fields) are prior unknowns when the execution starts, whose values are unknown to the executor.

1) In `foo1` shown below, c is first assigned a new concrete object with no known fields, i.e., $\langle T, Concrete, \emptyset \rangle$. Then $c.f$ is assigned and c becomes $\langle T, Concrete, \{f \mapsto taint\} \rangle$. Finally, `sink()` checks $c.f$. And since it is a taint, the executor reports a leak.

```
foo1(T x, T y) {
  c = new T();
  c.f = source();
  sink(c.f);
}
```

2) In `foo2` shown below, x starts as a PU. Then $x.f$ is assigned with a concrete object (the taint), which changes x from a prior unknown to a derived unknown $\langle T, DU, \{f \mapsto taint\} \rangle$. A derived unknown implies that this object was unknown (PUs) but some known values have been assigned to it during the execution. Therefore, when the concrete object c gets $x.f$, it gets the known value (the taint) assigned to $x.f$ before. Finally, the executor successfully reports the leak on `sink()`.

```
foo2(T x, T y) {
  x.f = source();
  c = new T();
  c.f = x.f;
  sink(c.f);
}
```

3) In `foo3` shown below, the condition checks if a concrete object c is equal to a prior unknown x. By definition, a prior unknown is created before execution while a concrete object is created afterwards. Thus the executor can safely evaluate the condition to be false and no leak will be reported.

```
foo3(T x, T y) {
  c = new T();
  if (c == x)
    sink(source());
}
```

4) In `foo4` shown below, the condition compares two prior unknowns. Since the executor does not know if x and y refer to the same object, this condition ends up as an unknown. The branching depends on an unknown condition and thus the executor reverts to the approximation mode, which will be discussed in details later.

```
foo4(T x, T y) {
  if (x != y)
    sink(source());
}
```

5) In `foo5` shown below, x changes to a derived unknown with a concrete field but y is still a prior unknown when its field is checked. Thus, the executor also needs to revert to approximations.

```
foo5(T x, T y) {
  x.f = source();
  sink(y.f);
}
```

From these examples, we illustrate how our object representation keeps record of both known and unknown objects and tracks their propagation to reflect the data flows of personal data.

C. Complete Execution Rules

Table II lists the complete execution rules used in AppAudit executor. Rule (1) to (7) have been covered by above-mentioned examples. The rest handle other bytecode instructions:

Function Call. Rule (8) shows that our dynamic analysis is naturally inter-procedural. When a function call is being made during execution, the executor will step into the function and pass the parameters accordingly.

Arithmetic Operations. Rule (9) and rule (10) outline how to evaluate binary and unary arithmetic expressions. These expressions take only primitive types. Basically the executor will compute concrete values when both operands have concrete values. When unknowns are present, the result will be unknown accordingly.

Comparison Operations. Rule (11) tackles comparison expressions. Similar to arithmetic operations, if both operands are concrete (known), the result can be evaluated naturally. When unknowns are present, the result is also unknown except for one case. If one operand is a concrete object while the other is an unknown (PU or DU), the result is definitely evaluated to false.

Array Operations. Rule (12) and rule (13) handle array operations, which are similar to rule (4) and rule (5). Changing an array element can also change a prior unknown to a derived unknown.

Branching Operations. Rule (14) and rule (15) handle branching instructions. For conditional jumps with unknown conditions, the executor will revert to the approximation mode.

D. Tainting Rules

Personal data is marked as *tainted*, which is propagated during execution. The taint tracking capability of AppAudit is largely similar to the taint propagation rules used in dynamic taint analysis [11], [8]. In our rules, rule (1) and rule (6) set taints explicitly. Rule (9) and rule (10) taint the result as long as one of its operands is tainted. Rule (12) taints x as long as i is tainted. This rule handles encryption libraries that perform substitution to encrypt data, hence tainted inputs should lead to tainted outputs.

E. Execution Extensions and Optimizations

Our executor contains several extensions and optimizations to accelerate the execution speed while maintaining the same instruction semantic.

Dynamic Dispatch (Reflection and Virtual Calls). Java virtual calls and reflections are dynamically dispatched. During execution, these call targets will be resolved.

Inlining Call-back Functions. As mentioned before, call-back functions are widely used and hide implicit data flows. Thus, when the executor encounters a trigger API, it will execute the callback function being registered after the current function is finished.

Exception Control Flow. Exceptions can affect control flows. Some instructions and APIs can generate exceptions (e.g. array indexing instructions and file related APIs). Currently our executor supports only plain exceptions (no nested exceptions). Unhandled exceptions are ignored during execution.

Library Emulation. Library functions contain large body of instructions and lots of calls into other library functions. The executor emulates some library functions for improved analysis performance. To emulate a particular library function, the executor manipulates its object representations directly to achieve the same effect of the emulated function. For example, to emulate `swap(x,y)`, the executor swaps the object representation directly without executing its bytecodes. Library emulation is commonly implemented to accelerate the calls to standard Java library functions.

Infinity Avoidance. During execution (both the *exec* and *approx* mode), our executor can run into infinity due to infinite loops and recursions in the application. For example, the application can spawn a thread that uses an infinite loop to check network updates. In real execution, this thread could be interrupted by user exiting the application. However, in approximated execution, this infinite loop will never end. To ensure that our analysis can always terminate, we design a threshold-based approximation to detect and terminate infinities. Both avoidance mechanisms lead the executor to the *leap* state when infinite loops or recursions are detected.

We introduce a counter to record how many instructions have been executed for a particular function. If this counter exceeds the total instruction count of the function times a certain threshold, we will cut short the execution.

Similarly for infinite recursions, we monitor the call stack during execution. If the depth exceeds a designated threshold, the executor assumes a stack overflow happens and then terminates the execution.

We obtain these two thresholds through empirical experiments. First we turn on instruction tracing such that every instruction being executed by AppAudit will be logged. Then we gradually increase the threshold until the infinity avoidance mechanism no longer cuts short any code paths. Finally, we double this fix point as our final threshold to ensure that these thresholds work for other real apps.

F. Approximation Mode

As shown in the execution rules, unknown values can be stored, propagated and evaluated with our object model. However, when a conditional jump instruction meets unknown values, the executor will fail to perform control flow

#	Instruction	Execution Semantic
(1)[1]	`x=12`	$\phi(x) \leftarrow \langle int, \mathbf{CON}, \{val \mapsto 12\}\rangle$
(2)[2]	`x=new T()`	$\phi(x) \leftarrow \langle T, \mathbf{CON}, \emptyset\rangle$
(3)[12]	`x=y`	$\phi(x) \leftarrow \phi(y)$
(4)[2]	`x.f=y`	$\begin{cases}\phi(x) \leftarrow \langle \phi_T(x), \mathbf{DU}, \phi_V(x) \cup \{f \mapsto \phi(y)\}\rangle, & \text{if } \phi_K(x) = \mathbf{PU}\\ \phi_V(x) \leftarrow \phi_V(x) \cup \{f \mapsto \phi(y)\}, & \text{otherwise}\end{cases}$
(5)[2]	`x=y.f`	$\phi(x) \leftarrow \phi_V(y)[f]$
(6)[1]	`x=source()`	$\phi(x) \leftarrow taint$
(7)[1]	`sink(x)`	switch to check mode
(8)[12]	`call fn(`$e_0, \cdots$`)`	assign parameters according to Rule (3)
(9)[1]	`x=y binop z`	$\begin{cases}\phi(x) \leftarrow \langle \phi_T(y), \mathbf{CON}, \kappa_{binop}(\phi_V(y), \phi_V(z))\rangle & \text{if } \phi_K(y) = \mathbf{CON} \wedge \phi_K(z) = \mathbf{CON}\\ \phi(x) \leftarrow \langle \phi_T(y), \mathbf{PU}, \emptyset\rangle & \text{otherwise}\end{cases}$
(10)[1]	`x=unop y`	$\begin{cases}\phi(x) \leftarrow \langle \phi_T(y), \mathbf{CON}, \kappa_{unop}(\phi_V(y))\rangle & \text{if } \phi_K(y) = \mathbf{CON}\\ \phi(x) \leftarrow \langle \phi_T(y), \mathbf{PU}, \emptyset\rangle & \text{otherwise}\end{cases}$
(11)[12]	`x=y cmp-op z`	$\begin{cases}\phi(x) \leftarrow \langle Bool, \mathbf{CON}, \kappa_{cmp}(\phi_V(y), \phi_V(z))\rangle & \text{if } \phi_K(y) = \phi_K(z) = \mathbf{CON}\\ \phi(x) \leftarrow \langle Bool, \mathbf{CON}, \{val \mapsto false\}\rangle & \text{if } \phi_K(y) \neq \phi_K(z) \wedge \phi_T(y) \in PRIMITIVE\_TYPES\\ \phi(x) \leftarrow \langle Bool, \mathbf{PU}, \emptyset\rangle & \text{otherwise}\end{cases}$
(12)[12]	`x=a[i]`	$\begin{cases}\phi(x) \leftarrow \phi_V(a)[\phi_V(i)[val]] & \text{if } \phi_K(a) = \mathbf{CON} \wedge \phi_K(i) = \mathbf{CON}\\ \phi(x) \leftarrow \langle ELEM_T, \mathbf{DU}, \emptyset\rangle & \text{otherwise}\end{cases}$
(13)[12]	`a[i]=x`	$\begin{cases}\phi_V(a) \leftarrow \phi_V(a) \cup \{\phi_V(i)[val] \mapsto \phi(x)\} & \text{if } \phi_K(a) = \phi_K(i) = \mathbf{CON}\\ \phi(a) \leftarrow \langle \phi_T(a), \mathbf{DU}, \phi_V(a) \cup \{\phi_V(i)[val] \mapsto \phi(x)\}\rangle & \text{if } \phi_K(a) \neq \mathbf{CON} \wedge \phi_K(i) = \mathbf{CON}\end{cases}$
(14)	`jmp-op cond,l`	$\begin{cases}pc \leftarrow \kappa_{jmpop}(\phi_V(cond), pc, l) & \text{if } \phi_K(cond) = \mathbf{CON}\\ \text{switch to approx mode} & \text{otherwise}\end{cases}$
(15)	`jmp l`	$pc \leftarrow l$

[1] this bytecode accepts primitive types
[2] this bytecode accepts class types

Table II: The execution rules. κ is a series of evaluation functions that perform real calculation when values are known.

decision. In this case, the executor changes to approximation mode.

Unknown Branching Approximation. The executor relies on this approximation to continue when it encounters branching instructions with unknown conditions. This approximation is designed to skip unknown loops, since these loops cannot provide useful known information from unknowns. Table III shows the four basic control flow structures compiled by an Android compiler. For the three looping structures, the branching approximation always chooses not to take the conditional branch to skip these loops. However, as we cannot distinguish ifs and loops, this approximation will only explore the "then" branch for unknown if-else structures. This bias is benign. Consider the following program:

```
foo() {
  T a = new T();
  T b = new T();
  bar(a,a);
  bar(a,b);
}
bar(T x, T y) {
  if (x == y)
    return ;
  else
    sink(source());
}
```

In this example, `bar` will be executed. When executing `bar`, both `x` and `y` are prior unknowns, which trigger the approximation to guide the executor to explore only the "then" branch and thus no leak will be reported. Due to insufficient calling contexts, the "else" branch will not be explored when analyzing `bar`.

Then according to our API analysis, `foo` will also be analyzed if `bar` has been analyzed, as `foo` is a caller of `bar`. When analyzing `foo`, the executor will analyze `bar` again with two concrete calling contexts. Under the `bar(a,a)` context, the condition will be evaluated to true and no leak will be reported. Under the `bar(a,b)` context, the condition will be evaluated to false and the leaking "else" branch will be explored.

Observed from this case, the unknown branching approximation only affects a function with insufficient calling contexts (`bar`). The approximation will result in fewer code paths being explored. But then the executor will reach callers (`foo`) of this function and re-analyze the unsuccessful function (`bar`) with more concrete calling contexts from its caller. If the program contains leaking paths, then at least one of these calling contexts will be sufficient to reach the leaking point. Thus the bias introduced before will be amortized. If the program does not contain leaking paths, then the approximation will skip some code paths but none

```
if (<cond>) { <then> }
else { <else> }
<rest>
```

(a) if statement

```
for (<init>;
    <cond>;
    <incr>) {
  <body>
}
<rest>
```

(b) for loop

```
while (<cond>) {
  <body>
}
<rest>
```

(c) while loop

```
do {
  <body>
} while (<cond>);
<rest>
```

(d) do-while loop

(a) if statement

```
cond_label:
▷ jump, !<cond>, else
  <then>
  goto rest
  <else>
  goto rest
  <rest>
```

(b) for loop

```
  <init>
cond_label:
▷ jump, <cond>, body
  <rest>
  <body>
  <incr>
  goto cond_label
```

(c) while loop

```
cond_label:
▷ jump, <cond>, body
  <rest>
  <body>
  goto cond_label
```

(d) do-while loop

```
  <body>
cond_label:
▷ jump, <cond>, body
  <rest>
```

Table III: Four basic control flow structures and their compiled bytecode streams.

of them will leak data. In short, the unknown branching approximation will not miss leaking code paths.

G. Accuracy Analysis

Since our executor performs dynamic analysis, we would like to ensure that it does not miss important (leaking) code paths. When running in execution mode, the executor can faithfully reproduce the actual path of the real execution. When the executor turns to the approximation mode, it will explore only a few possible code paths, which could lead to false negatives. We have analyzed the side-effect of unknown branching approximation. When the application contains leaking paths, the executor will have a proper calling context for executing regardless of this approximation. Thus this approximation only misses non-leaking paths and is benign to the overall accuracy. Our infinity avoidance relies on two thresholds to cut short infinite loops and recursions. Both are obtained from empirical experiments.

In addition, our executor embodies a taint analysis to track the dissemination of personal data. Thus the correctness of taint rules also affects the accuracy of the executor. We identify the following cases that could affect accuracy of a dynamic taint analysis:

Taint Sanitization. Currently, our tainting rules only add and propagate taints but never remove them. This could lead to inaccuracy and false positives. One typical case is about rule (9) and rule (10) in Table II. Currently, the result of an arithmetic operation will be tainted as long as one operand is tainted. However some arithmetic operations always return the same result regardless of the value of the operands. For example, $x = y \oplus y$ always returns zero. For such cases, the taint on the result should be removed. Our current implementation does not have taint sanitization. Nevertheless, although these cases are possible for hand-crafted applications, the standard Android Java compiler never generates such idioms.

Array Indexing. For an array operation $x = a[i]$, currently x will be tainted if i is tainted. This is because encryption functions usually use an array to map plain-text inputs to encrypted outputs. Thus the taints on the output is dependent on the input. This is commonly employed by other taint analyses [11], [8] to deal with encryption libraries. However, if the array is zero-valued, then x will always be zero regardless of the index i. Again, the current propagation rule over-taints the results and could lead to false positives.

Control Flow Dependent Taints. This is a well acknowledged drawback in most taint analysis [11], [8], [13].

```
if (x == 1) y = 1;
else if (x == 2) y = 2;
```

In this case, the values of the two variables are correlated so should be their taintness. However, by using the control flow structures, the executor is unaware of the correlation and always produces an untainted y. ScrubDroid [13], [14] presents more attacking cases for a standard taint analysis. We expect to integrate a more powerful code structure recognition module to detect such cases in the future.

V. Evaluation

In this section, we evaluate AppAudit in terms of its accuracy and usability. We demonstrate the three use cases of AppAudit, with regards to market operators, app developers and mobile users. We also present a characterization study about data-leaking apps, providing guidance for designing effective data leak prevention tools.

A. Implementation

The AppAudit prototype is implemented with Java, and reflectively loads Android SDK for API signatures. Table IV presents the breakdown of source lines of code for different components. We leverage dex2jar for disassembling [15] and APKParser [16] for manifest parsing. The API analysis accounts for a relatively small portion of the entire code

Component	Percent.	Description
preprocessors	11.2%	Disassembler and manifest parser
emulation	28.5%	Library and device emulation
core engine	15.8%	Core approximated executor
objmodel	20.2%	Object representation
apianalysis	7.0%	Call graph based API analysis
util	17.3%	Utility
Total	100%	10,559 lines of code

Table IV: The SLOCs for different components.

base. The approximated executor is the main contributor to the code base, which implements an Android Dalvik [17] bytecode virtual machine.

Portability. Our current prototype is implemented in Java, which can run on different platforms. We have an optimized version for server configuration and an Android port with simple GUI.

API emulation efforts. As shown in Table IV, API emulation accounts for 28.5% of our code base. Currently API emulation is done manually. We have emulated 54 classes and 130 functions, which are the most frequently used in the apps in our evaluation datasets. API emulation is a tedious task and we are exploring automated ways to generate emulated code for all standard Java library APIs.

Device emulation. We emulate a Samsung Galaxy Nexus (i9250) smartphone running Android 4.0.3, with WiFi and cellular connections. The specific model number, serial, OS version code, CPU types are dumped from a real phone. These information are exposed to the app in the standard Android class `android.os.Build`. We also emulate a basic `/proc` file system to present the low-level information about the emulated device.

Parallelized Execution of Multiple Approximated Execution. To further improve the analysis speed on multi-core platforms, AppAudit executes multiple (four by default) code paths concurrently. Each code path is executed in a separate execution context and shares no states between each other. Thus the dynamic analysis is fully parallelizable and the parallelism can be adjusted for different use cases.

Native code. Some Android apps can link and call into native libraries. Currently, our executor does not execute native code and will simply return an unknown when it meets a native function. We expect a binary executor to provide fine-grained data flow information about native functions.

B. Evaluation Methodology

Our evaluation contains four parts. First we use a micro-benchmark suite to validate the completeness of our static API analysis. Second, we use malware samples to evaluate the accuracy of AppAudit. In particular, we want to answer these two questions: 1) Can our dynamic analysis guarantee no false positives? 2) Can AppAudit provide comparable code coverage as static analysis (a low false negative rate)? Third, we use real-world apps to evaluate the usability as well as usefulness of AppAudit. Our real app based evaluation aims to answer the following questions: 1) What is the analysis time and memory consumption? 2) How could AppAudit be used in different use cases? Fourth, we present characterization study of data-leaking apps uncovered by AppAudit. We aim to show the common properties among these apps so as to provide guidance for designing effective prevention tools.

Dataset	# Samples	Description
droidbench	56	A micro-benchmark [18] that stresses the completeness of taint analysis
malware	1005	Android malware genome project [19]
freeapp	428	Popular free apps from the official market

Table V: Evaluation datasets.

Evaluation datasets. Table V summarizes the datasets used in our evaluation.

1) DroidBench [18] dataset. DroidBench contains a suite of hand-crafted Android applications that exploit various features of the language and programming model to bypass taint analysis. We use DroidBench to validate the completeness of our static API analysis.
2) Malware dataset. Our malware dataset contains 1,005 samples from the Android malware genome dataset. We select the ones related to data leaking based on extensive reference of studies from mobile security companies and labs [20], [21], [22], [23], [24], [25], [26]. Malware samples have well understood malicious behavior [19], which serves as a good accuracy index for data leak detection tools.
3) Free app dataset. Real apps are normally much larger and more complicated than malware. Thus, we choose these samples to evaluate the analysis performance and usefulness of program analysis tools. Our initial sampling began around March 2013 when half of the dataset were collected. We notice some user feedbacks about data leaks online. So we collect newer versions of these apps around January 2014 to outline how developers respond to these reported data leaks. Collected apps comprise not only top free apps but also newly uploaded apps during that sampling time period.

Evaluation candidates. We compare AppAudit with two state-of-the-art pure static analysis tools.

FlowDroid [4] leverages a precise flow graph to find leaking data flows. FlowDroid achieves high precision by accurately modeling the runtime behavior of Android applications with a flow graph. On the contrary, AppAudit largely relies on executing bytecode to reproduce and confirm leaks in real execution. FlowDroid is open-source and thus we can compare the results of both across all three evaluation datasets.

AppIntent [3] is a static analysis based on symbolic execution. Its main goal is to prune false positives and optimizes the performance of symbolic execution. AppAudit also leverages a dynamic analysis to reduce false positives, which naturally becomes a competitive approach for the same purpose. AppIntent is not publicly available and we only have its results on the malware dataset.

C. Completeness of Static API Analysis

AppAudit adopts a two-stage design where the static API analysis will narrow down the analysis scope for the dynamic analysis. So the static stage should completely include all possible data leaks. We use DroidBench to evaluate the completeness of our static API analysis. FlowDroid is the only previous approach compared in this analysis since AppIntent is not available to test on this benchmark.

DroidBench contains 65 test cases in total. We exclude 9 unsupported cases and use the rest 56 for our completeness evaluation. Four excluded cases are related to control flow dependent taints (see Section IV-G). This problem is itself an interesting and hard research topic, which is currently not supported by FlowDroid [4] (the state-of-the-art static analysis) and AppAudit. Three excluded cases are because AppAudit does not treat password input widgets as source APIs so far. Two excluded cases declare GUI callbacks via XML files, which is not fully supported by AppAudit.

Among the remaining 56 DroidBench tests, AppAudit produces no false positives and two false negatives. As a comparison, FlowDroid has four false positives and two false negatives. Overall, AppAudit achieves fewer false positives and as few false negatives as FlowDroid. AppAudit eliminates all false positives with its dynamic analysis. The dynamic analysis only executes possible code paths and thus false positives caused by impossible code paths will be pruned. The two false negative cases of AppAudit both leak data when particular user inputs happen in a particular order. AppAudit fails to report these leaks because it cannot model infinite possibility of user input orderings. Previous work [3] argues that some particular ordering of user inputs might imply user awareness of the data leak, which indicates that a detection tool should not report such leaks.

D. Detection Accuracy

Our malware dataset contains 23 malware families, covering a wide range of malicious data-stealing behavior.

We compare AppAudit with both AppIntent and FlowDroid. We do not consider existing dynamic analysis like TaintDroid [8] for accuracy comparison because 1) existing dynamic analysis requires user inputs and can hardly be automated; 2) static analysis can achieve better code path coverage than existing dynamic analysis.

We also compare AppAudit with a collection of commercial solutions, including off-the-shelf anti-virus software and the Google Application Verification Service (AppVerify) shipped with Android 4.2 [27]. The results of commercial anti-malware are obtained from VirusTotal [28], a website that scans submitted mobile apps with latest mobile anti-virus solutions. In terms of AppVerify, we reference the results from an existing study [27].

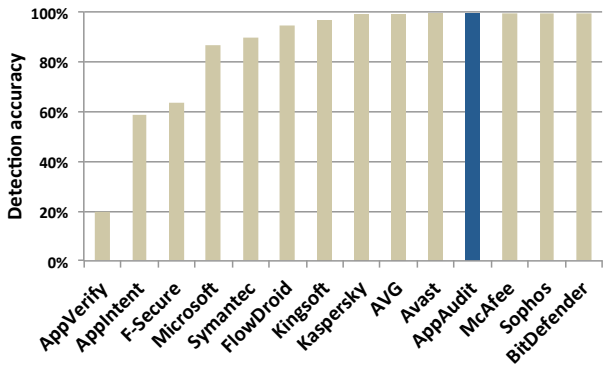

Figure 5: The overall true positives on Android malware genome dataset (99.3%).

Malware family	TP+TN	FP	FN	Sample #
AnserverBot	187	0	0	187
Badnews	2	0	0	2
BeanBot	1	0	7	8
BgServ	9	0	0	9
DroidDreamLight	46	0	0	46
DroidKungFu1	34	0	0	34
DroidKungFu2	30	0	0	30
DroidKungFu3	309	0	0	309
DroidKungFu4	96	0	0	96
Endofday	1	0	0	1
Geinimi	69	0	0	69
GGTracker	1	0	0	1
GingerMaster	4	0	0	4
GoldDream	47	0	0	47
jSMSHider	16	0	0	16
KMin	52	0	0	52
DroidKungfuSapp	3	0	0	3
LoveTrap	1	0	0	1
NickyBot	1	0	0	1
Pjapps	58	0	0	58
Plankton	11	0	0	11
RogueSPPush	9	0	0	9
SndApps	10	0	0	10
Spitmo	1	0	0	1
Total	998	0	7	1005

TP: True Positive, TN: True Negative;
FP: False Positive, FN: False Negative

Table VI: The breakdown of detection accuracy on Android malware genome dataset.

Overall Detection Accuracy. Figure 5 shows the comparison of overall detection accuracy (true positives plus true

negatives) among all analysis tools and anti-virus solutions. AppAudit outperforms two state-of-the-art static analysis tools and a number of commercial solutions with a detection accuracy of 99.3%. AppIntent overkills some cases with its pruning mechanism. FlowDroid fails on 6 samples due to memory exhaustion. Table VI provides a breakdown for false positive and negative cases for AppAudit.

False Positives. Overall, AppAudit achieves no false positives while FlowDroid reports one false positive from DroidDreamLight samples. We inspect this case to understand the reason. Generally, DroidDreamLight samples collect personal data and then send them to a list of remote servers. These samples decrypt a configuration string with a hard-coded DES key to obtain a list of target servers. However, the particular case has a malformed configuration string and thus no target servers will be obtained and no data leaks will actually happen. The decryption contains lots of substitutions with array operations, which stresses static analysis to correctly model them. With our dynamic analysis, AppAudit can faithfully perform the complete decryption and obtain the decrypted string. Consequently, AppAudit validates that the leaking code snippet in this case is actually dead code due to the malformed configuration and successfully prunes this false positive case.

False Negatives. AppAudit reports seven false negative cases on the malware dataset, all from the `BeanBot` family. Our manual de-compilation and check reveal that `BeanBot` retrieves personal data and then sends a text message to a cellular number for the service code of the carrier. Once it receives the response text message, it will leak the user data [29], [30]. This shows a typical case where the sending behavior is dependent on external inputs. Since AppAudit cannot predict the content of the incoming text message, it cannot firmly report this case as a data leak.

This false negative scenario outlines the major difference between static and dynamic analysis in handling data leaks that depend on external inputs. Such a situation shows that a leak will be triggered given some external inputs (input-sensitive leaks). Dynamic analysis would tend not to report this as a leak, because the analysis cannot firmly ensure that the leaking path will be visited. On the contrary, static analysis tends to treat the path as leaking as long as it finds one possible input that could lead to a leak. Under such circumstances, both analyses are guessing if the leaking path will be visited (dead code or not) based on unknown external inputs.

A better indicator for this situation is to determine whether this data leak is user-intended or not [3]. If the input is a user input, then probably the user agrees to let the app send the data and thus such input-sensitive leaks should not be labeled as actual leaks. If the input is a message from an untrusted source (like the case with `BeanBot`), then such input-sensitive leaks are more likely to be actual leaks. Modeling such inputs would be an interesting future direction for AppAudit.

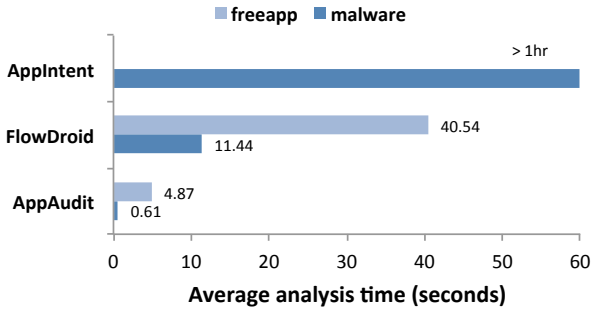

Figure 6: The average analysis time per app for AppAudit and two static analysis tools. Note that FlowDroid only finishes 61% of the samples (due to OutOfMemory exceptions and 10-minute timeout). Its average time only includes successful cases.

E. Usability

Real Android applications are generally much larger and more complicated than malware. To examine the practicality of various app auditing tools, we conduct an experiment to compare the analysis time as well as memory consumption for existing tools and AppAudit. Our performance experiment runs on a desktop PC equipped with a quad-core 3.4GHz i7-3770 processor and 8G memory, running 64-bit CentOS 7 and Oracle Java 7.

Analysis Time. Figure 6 compares the average analysis time per app of the three candidate program analysis tools when examining real apps. Since AppIntent is not publicly available, we only reference its results for malware samples. FlowDroid and AppAudit both have two working modes. The *single* mode reports only one data leak and the *full* mode reports all data leaks. We choose to report the analysis time with *single* mode, which is the most efficient mode for both tools.

As shown, AppAudit performs much faster than static analysis tools. Specifically, AppAudit performs 8.3x faster than FlowDroid, the best-performing static analysis so far. With long analysis time, static tools are generally not acceptable for mobile users. Meanwhile, longer analysis time requires market operators to spend more resources to run the analysis.

To further improve AppAudit performance, we measure the breakdown of AppAudit analysis time. The breakdown shows that around 30% to 40% of the analysis time is spent on disassembling. We are planning to adopt a multi-threaded implementation to accelerate this phase. Meanwhile, we also discover that some functions are executed repeatedly during the dynamic analysis and return value caching could be a direction for optimization as well.

Requirements	Market operators	Developers	Mobile users
Platform	server	desktop	mobile device
Analysis time	days	minutes	real-time
Memory	$< 100G$	$< 16G$	$< 1G$
Result granularity	brief/complete	complete	brief

Table VII: App auditing use cases and requirements.

Memory Consumption. Memory footprint is also an important constraint of program analysis tools when examining complicated and large real applications. AppIntent requires 32GB and FlowDroid needs about 2GB to 4GB memory by default. Static analysis tools generally require large memory because they need to accommodate huge data structures (such as flow graphs and symbolic representation). The space complexity of these data structures is proportional to the size of the application code base. This constraint makes static analysis memory-consuming for analyzing large real apps.

On the contrary, dynamic analysis is more memory efficient. AppAudit only requires a heap size of 256MB, which can run on mobile devices, PCs and servers. According to our measurement, the peak memory consumption AppAudit is only 10% of FlowDroid in most tested cases. In our implementation, we apply several optimizations to control the overall memory consumption of AppAudit. First, we trigger a manual garbage collection after the API analysis to keep minimum analysis data structures in memory after the static stage (e.g. bytecode of the app, its class hierarchy). These analysis structures take around 2MB to 20MB memory space according to our measurement. Second, when executing the target app, the memory consumed by the executor is proportional to memory consumption of the target app. When some memory objects are no longer needed by the target app, AppAudit will also dereference them such that they will be automatically garbage collected by the JVM hosting AppAudit.

Use cases and requirements. We discuss the use cases of app auditing and elaborate the requirements imposed on the auditing tool for each case. Table VII summarizes the three cases and their requirements. We obtain the memory constraints with regards to the memory capacity of the individual platform that runs app auditing.

First, app market operators demand an app auditing tool to identify data-leaking apps uploaded to the market. Usually, this use case demands low false positive and false negative rates but do not have strict requirements for the time, memory and result granularity, since the analysis commonly runs on powerful cluster servers. AppAudit outstands for this case for its high detection accuracy and low resource consumption, which ensures detection quality while greatly saves the resource investment for automatic app auditing.

The second use case of app auditing is to allow app developers to check their apps before publishing. In this case, developers demand the tool to report all possible data leak problems within the capability of a development machine, such as a desktop PC. AppAudit and FlowDroid can both report all data leaks found in an app. When working in the *full* mode, both tools require more time than the *single* mode shown in Figure 6. Our measurements show that AppAudit runs 4.7x to 7.8x slower for individual apps while FlowDroid encounters more OutOfMemory exceptions and observes similar slowdowns. Nevertheless, AppAudit still manages to finish analysis within one minute and stands for a competitive choice for this use case.

The final use case is to run auditing tools on mobile devices and help users to avoid installing data-leaking apps. In this case, the analysis has strict memory and time constraint. However, it is only expected to provide brief auditing results, sometimes just whether the app will leak data or not. Figure 6 shows the analysis time on a desktop PC, which shows that AppAudit is the only tool that can fulfill this task on mobile devices. Other tools require memory that is unrealistic even for high-end devices nowadays. We port AppAudit to Android and run it to check apps installed on an LG Nexus 5 smartphone. This device is a late-2013 model with a quad-core 2.3GHz CPU and 2GB RAM. We then experiment the analysis time again with the mobile version of AppAudit. The results show a 1.5x to 2.3x slowdown as compared to Figure 6.

F. Characterization of Data Leaks in Real Apps

AppAudit uncovers 30 data leaks in 400 real apps we collected. For all detected data leaks, we manually confirm them by decompiling related apps and examine the leaking code paths. Based on the reported cases by AppAudit, we can easily characterize data leaks in terms of the leaking component (simply the class name), the leaking sources and venues. Table VIII summarizes our characterization results. In this table, we crawl number of downloads to highlight the number of affected users. We also crawl the privacy policy of these leaking apps to clarify if data leaks are made clear to users. Our characterization results show the following interesting findings:

Finding 1: Most data leaks are caused by 3rd-party advertising libraries. From Table VIII, we found that 28 out of the 30 (93.3%) detected data leaks are caused by 3rd-party advertising libraries. As previous research [31], [1], [2] has pointed out, 3rd-party advertising modules aggressively request application permissions to access various personal data. If an advertising library leaks data, it can potential affect lots of apps.

Meanwhile, hackers have started to exploit advertising libraries to spy on users [32]. We believe that privilege separation [33], [34], [35] and fine-grained privilege control will help to prevent the threats caused by these problematic libraries. From the perspective of app developers, AppAudit can help check their apps before publishing to the market,

Name	Component	Source	Venue	Privacy Policy	Installs (M for millions)
Texas Poker v4.0.1	App	Location	HTTP GET	×	10M-50M
Word Search v1.14	Mobfox	Location, IMEI	HTTP GET	~~app~~,~~lib~~	0.5M-1M
Speedtest v2.09	Mobfox	Location, IMEI	HTTP GET	app,lib	10M-50M
Brightest Flashlight v2.3.3	MDotm, Mobclix	IMEI, IMSI	HTTP GET	app,~~lib~~	50M-100M
Weather Underground v2.1.2	App, Mobclix	Location, IMEI	HTTP GET	app,lib	1M-5M
Fruit Ninja *(2 samples)*	Mobclix	IMEI	HTTP GET	app,lib	100M-500M
Angry Birds *(14 samples)*	Jumptap	IMSI	HTTP GET	app,lib	300M-900M
Bad Piggies *(3 samples)*	Jumptap	IMSI	HTTP GET	app,lib	10M-50M
Tap Tap Revenge v4.3.3 v4.4.5	Tapjoy	IMEI	HTTPS GET	×	0.1M
Logo Quiz v8.8	Tapjoy	IMEI	HTTPS GET	×	10M-50M
Trial Extreme v1.28 & v2.83	Tapjoy	IMEI	HTTPS GET	×	5M-10M
Big Win Basketball v2.0.4	Tapjoy	IMEI	HTTPS GET	app,~~lib~~	5M-10M
Solitaire v2.1.5	Tapjoy	IMEI	HTTPS GET	~~app~~,lib	50M-100M
Talking Tom 2 v2.0.3	Tapjoy	IMEI	HTTPS GET	app,lib	100M-500M

Table VIII: Free apps that spread certain personal information identified by AppAudit. For the "Privacy Policy" column, a "~~lib~~" means that the privacy policy does not cover the kind of data spread by advertising libraries.

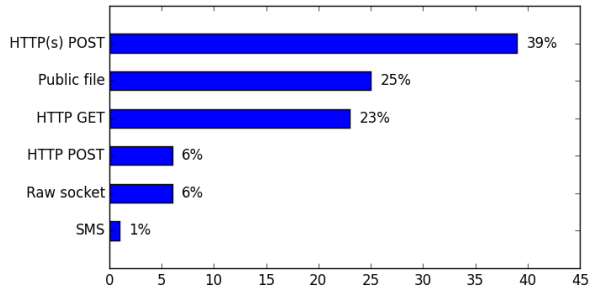

Figure 7: The venues of data leaking.

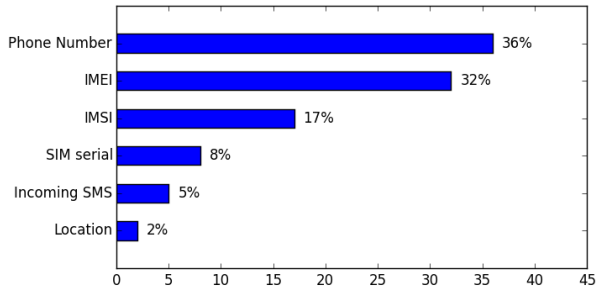

Figure 8: The types of leaked data.

which could effectively detect data leaks beforehand and avoid accidentally using data-leaking 3rd-party modules.

Finding 2: HTTP requests are the most prominent leaking venues. Figure 7 presents the leaking venues for all data-leaking cases in malware and free app datasets. HTTP(s) transmission turns out to be the most popular venue to leak data, since HTTP servers can be easily configured. This suggests that mobile application confinement tools [36], [34], [37], [38], [39] can focus on HTTP traffic to effectively confine data leaks.

Nevertheless, eight reported free apps transmit personal information in plain-text forms (HTTP GET requests). Consequently, some important personal information (locations and identity) can be easily obtained by traffic sniffing in the public. To make things worse, some of these report apps do not have a clear privacy policy statement, which makes users unaware of the potential risks.

Finding 3: Tracking is universal. Figure 8 presents the breakdown of leaked data found in the malware and free app datasets. We discover that the IMEI number and phone number is the most commonly leaked information. The phone number is commonly sent by malware for follow-up SMS phishing. IMEI serves as the phone identity and is widely used to track user for targeted advertising. Nowadays, each free app is bundled with a couple of advertising libraries [1] and the user interacts with a number of apps. IMEIs to mobile devices is what cookies are to web browsers. Cookies are bound to individual websites, i.e. one website cannot access the cookies of another. However, IMEI tracking is not bound to individual apps but to individual advertising libraries. Thus, if two apps use the same advertising library, the advertiser can accurately track user's transition from one app to another. If the data transmission between the library and server is unencrypted, the trace can be acquired and used to predict user habits and launch social engineering attacks.

Finding 4: Apps and advertising libraries are gaining awareness of user privacy. We find that, apps (Word Search and Speedtest) are gaining awareness of privacy by removing problematic advertising libraries. We believe that AppAudit, when integrated with IDEs, could well assist developers for this purpose. On the other hand, we discover advertising libraries are gaining privacy awareness as well. For example, a newer version of the Tapjoy advertising library hashes

IMEI before sending it to the advertising server. Given that hashing is cryptographically hard to invert, hashing effectively avoids leaking plain-text IMEIs. The newer versions of Trail Extreme and Big Win Basketball benefit from this simple hashing and no long leak data because of Tapjoy. These advancements witness the improved awareness of user privacy for both apps and advertising libraries.

VI. Related Work

In this section, we introduce the related work about analyzing mobile applications. The related approaches can be divided into two categories. Static analysis produces analysis results by statically analyzing various files associated with the application. Dynamic analysis runs with the application in real devices and reports problems when they happen. The synergy of static and dynamic analysis is exploited by AppAudit as well as by existing work for various purposes [40].

A. Static Analysis

We discuss existing techniques in terms of their analysis granularities.

Permission-based analysis. Android defines permissions for an application to access various resources and system services. Every application is required to declare the permissions it uses in its manifest file. Permissions can serve as an approximation of application behavior, which has been leveraged to identify malicious apps [41]. Kirin [42] checks application permission usage with a set of security policies. However, permission analysis cannot distinguish if the application is abusing permissions. For example, having the access to personal information and network capability may not lead to the conclusion that the application will leak personal information via the network. As a result, permission-based analysis normally faces high false positive when used to analyze personal information leakage [43].

API analysis. To complement permission analysis, existing approaches have made an attempt to analyze the API usage of applications. Stowaway [31] extracts the APIs used by an application and checks if the app is overdemanding permissions. RiskRanker [43] uses API analysis to quickly identify applications that have higher security and privacy risks. API analysis refines the analysis granularity of permission-based analysis. However, API analysis does not consider the information flows within the application, which fails to justify privacy leakage with detailed code path.

Dataflow analysis. Dataflow analysis is a classic program analysis, used for information flow validation and data reachability test. PiOS [5] performs reachability dataflow analysis on iOS apps to identify potential privacy leaks. ContentScope [6] applies dataflow analysis to detect unwanted information leakage from personal information databases to third-party Android applications. In general, dataflow analysis can provide more accurate and informative results than previous analysis. However, dataflow analysis can encounter difficulties due to the wide use of GUI and event-driven programming paradigms in mobile apps [3].

Symbolic Execution. Symbolic execution is an alternative code analysis to finding information leakage. Symbolic execution faces the fundamental challenge of path explosion, especially with event-driven GUI programs. AppIntent [3] aims to reduce the number of paths to be executed based on Android intent propagation rules to improve performance of symbolic execution. Nevertheless, AppIntent still requires minutes to hours to examine an app.

B. Dynamic Analysis

Dynamic analysis is implanted into the mobile operating systems and monitors applications at runtime. TaintDroid [8] applies dynamic taint analysis to various components of Android OS, which tracks sensitive information flow and reports to the user when sensitive information leaves the device. AppFence [44] retrofits the Android OS to provide fake sensitive information to the applications upon user's requests. VetDroid [7] dynamically records the permission usage of untrusted applications, which is then analyzed offline to reveal malicious behavior.

Compared with static analysis, dynamic analysis only reports suspicious behavior that occurs at runtime. This feature avoid false alarms and is appealing to the end user. However, for other use cases (market-level vetting, detailed code analysis), dynamic analysis can be limited by low code coverage.

C. Compiler Techniques

The approximations in AppAudit are largely inspired by analysis techniques used in just-in-time compilers. Many of the design decisions in our execution engine is inspired by improvements to symbolic execution, e.g. prefix symbolic execution engine [45], directed symbolic execution [46]. Also our object representation is inspired by [47].

VII. Conclusion

Mobile devices carry abundant personal information. Program analysis can effectively reveal data leaks in apps and protect user privacy. App auditing has three major use cases. First, app market operators require automatic tools to detect and remove data-leaking apps. Second, app developers need to perform self-check before publishing apps. Third, mobile users expect to know if an app is leaking data before installation.

In this paper, we design AppAudit, an efficient analysis framework that can deliver high detection accuracy with significantly less time and memory. AppAudit comprises a static API analysis that can effectively narrow down analysis scope and an innovative dynamic analysis which could efficiently execute application bytecode to prune false positive and confirm data leaks.

According to our experiments on read-world malware and apps, AppAudit achieves a 99.3% true positive rate (comparable to static analysis) and no false positives. Most importantly AppAudit performs 8.3x faster than the state-of-the-art static analysis tools, which makes it the only solution viable for important use cases of app auditing.

VIII. Acknowledgment

This work was supported in part by the NSERC Discovery Grant 341823, Canada Foundation for Innovation (CFI) Leaders Opportunity Fund 23090. The testbed was supported in part by NSF China Grant 61272101 and the Shanghai Key Laboratory of Scalable Computing and Systems. The authors would like to thank Xinye Lin and our reviewers for the discussions and feedbacks.

References

[1] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12. New York, NY, USA: ACM, 2012, pp. 101–112.

[2] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 21–21.

[3] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1043–1054.

[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 the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2013, pp. 259–269.

[5] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *Proceedings of the 18th Network and Distributed System Security*, ser. NDSS '11, 2011.

[6] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security*, ser. NDSS '13, 2013.

[7] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 611–622.

[8] W. Enck, P. Gilbert, 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," in *Proceedings of the 9th USENIX Conference on Operating systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[9] K. Ali and O. Lhoták, "Application-only call graph construction," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 688–712.

[10] "Android callbacks (flowdroid project)," https://github.com/secure-software-engineering/soot-infoflow-android/blob/develop/AndroidCallbacks.txt.

[11] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, ser. ICISS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–25.

[12] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331.

[13] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices," in *Proceedings of the 10th International Conference on Security and Cryptography*, July 2013, pp. 461–467.

[14] "Scrubdroid/antitaintdroid project," http://gsbabil.github.io/AntiTaintDroid/.

[15] "dex2jar: Tools to work with android .dex and java .class files," https://code.google.com/p/dex2jar/.

[16] "Xml apk parser," https://code.google.com/p/xml-apk-parser/.

[17] "Android dynamic java virtual machine: Bytecode instruction set," https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[18] "Droidbench, an open test suite for evaluating the effectiveness of taint-analysis," https://github.com/secure-software-engineering/DroidBench.

[19] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.16

[20] "Lookout mobile security," https://www.lookout.com/.

[21] "Avg mobile security," http://www.avgmobilation.com/.

[22] "Sophos security," http://www.sophos.com/en-us.aspx.

[23] "Nviso apkscan," http://apkscan.nviso.be/.

[24] "Semantec mobile security," http://www.symantec.com/mobile-security.

[25] "Fortiguard virus encyclopedia," http://www.fortiguard.com/encyclopedia/.

[26] "Trendlabs security intelligence blog," http://blog.trendmicro.com/trendlabs-security-intelligence/.

[27] "An evaluation of the application verification service in android 4.2," http://www.cs.ncsu.edu/faculty/jiang/appverify/.

[28] "Virustotal: Free online virus, malware and url scanner," https://www.virustotal.com/.

[29] X. Jiang, "Security alert: New beanbot sms trojan discovered," http://www.csc.ncsu.edu/faculty/jiang/BeanBot/.

[30] M. X. Lu Gong, "Beanbot analysis report," https://github.com/mingyuan-xia/AppAudit/wiki/BeanBot-analysis-report.

[31] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638.

[32] "Free apps used to spy on millions of phones: Flashlight program can be used to secretly record location of phone and content of text messages," http://www.dailymail.co.uk/news/article-2808007/Free-apps-used-spy-millions-phones-Flashlight-program-used-secretly-record-location-phone-content-text-messages.html.

[33] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '12. New York, NY, USA: ACM, 2012, pp. 71–72.

[34] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov, "πbox: A platform for privacy-preserving apps," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '13, 2013, pp. 501–514.

[35] X. Zhang, A. Ahlawat, and W. Du, "Aframe: Isolating advertisements from mobile applications in android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13. New York, NY, USA: ACM, 2013, pp. 9–18.

[36] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark, "Firedroid: Hardening security in almost-stock android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13. New York, NY, USA: ACM, 2013, pp. 319–328.

[37] Y. Huang, P. Chapman, and D. Evans, "Privacy-preserving applications on smartphones," in *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, ser. HotSec'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 4–4.

[38] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. SEC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 27–27.

[39] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 131–146.

[40] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, Portland, OR, May 9, 2003, pp. 24–27.

[41] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proceedings of the 19th Network and Distributed System Security*, ser. NDSS '12, 2012.

[42] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 235–245.

[43] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th International Conference on Mobile systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 281–294.

[44] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 639–652.

[45] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Softw. Pract. Exper.*, vol. 30, no. 7, pp. 775–802, Jun. 2000.

[46] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proceedings of the 18th International Conference on Static Analysis*, ser. SAS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 95–111.

[47] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle, "Alias analysis for optimization of dynamic languages," in *Proceedings of the 6th Symposium on Dynamic Languages*, ser. DLS '10. New York, NY, USA: ACM, 2010, pp. 27–42.