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 antivirus solutions. In terms of AppVerify, we reference the results from an existing study [27].

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

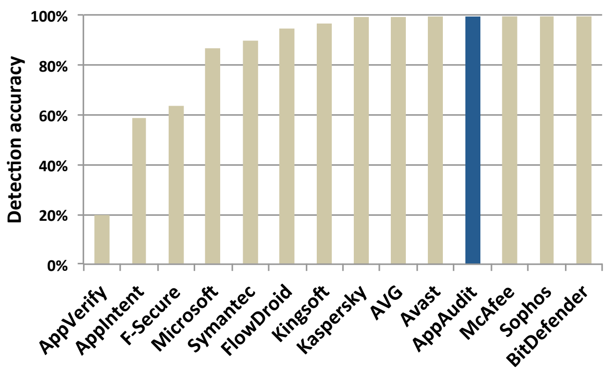

Figure 5: The overall true positives on Android malware genome dataset (99.3%).

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.

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.

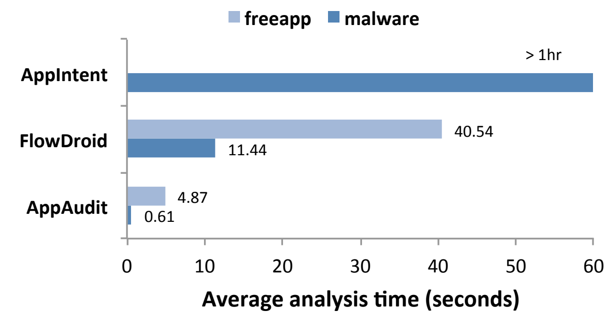

Figure 6: The average analysis time per app for AppAudit and two static analysis tools. Note that FlowDroid only finishes 61% of the samples (due to OutOfMemory exceptions and 10-minute timeout). Its average time only includes successful cases.

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,