LeakSemantic: Identifying Abnormal Sensitive Network Transmissions in Mobile Applications

Hao Fu*, Zizhan Zheng†, Somdutta Bose*, Matt Bishop*, Prasant Mohapatra*
*Department of Computer Science, University of California, Davis, USA.
†Department of Computer Science, Tulane University, New Orleans, USA.
{haofu, sombose, bishop, pmohapatra}@ucdavis.edu, zzheng3@tulane.edu

***Abstract*—Mobile applications (apps) often transmit sensitive data through network with various intentions. Some transmissions are needed to fulfill the app's functionalities. However, transmissions with malicious receivers may lead to privacy leakage and tend to behave stealthily to evade detection. The problem is twofold: how does one unveil sensitive transmissions in mobile apps, and given a sensitive transmission, how does one determine if it is legitimate?**

In this paper, we propose LeakSemantic, a framework that can automatically locate abnormal sensitive network transmissions from mobile apps. LeakSemantic consists of a hybrid program analysis component and a machine learning component. Our program analysis component combines static analysis and dynamic analysis to precisely identify sensitive transmissions. Compared to existing taint analysis approaches, LeakSemantic achieves better accuracy with fewer false positives and is able to collect runtime data such as network traffic for each transmission. Based on features derived from the runtime data, machine learning classifiers are built to further differentiate between the legal and illegal disclosures. Experiments show that LeakSemantic achieves 91% accuracy on 2279 sensitive connections from 1404 apps.

I. Introduction

The exponential growth of mobile devices has raised significant security concerns. Due to the large amount of sensitive data saved on these devices and the coarse-grained permission management in mobile systems, they are vulnerable to various privacy and malicious infringing behaviors, which is often hard to detect by mobile users themselves. One reason is that malicious apps have begun taking steps to avoid detection by introducing *logic bombs* [23]. For instance, an app can hide malicious transmissions by receiving certain commands from remote servers. Even if a sensitive network transmission is known, an end user often has trouble telling if it is necessary since the legitimacy of a sensitive transmission depends on its purpose. Therefore, it is critical to uncover security-sensitive behaviors and understand the *intention* of them to detect the abnormal ones.

In this paper, we focus on detecting abnormal sensitive network transmissions in Android apps. These transmissions either leak user private data to malicious servers, or collect sensitive information for purposes such as advertisements that do not contribute to fulfill the functionalities of the underlying apps. Despite the fast-growing literature on mobile device security and privacy, existing approaches are insufficient for identifying abnormal sensitive network behaviors. In particular, their ability is limited by the complexity of the Android API and runtime, which involves millions of lines of code. Moreover, they focus on detecting sensitive transmissions only and are often not able to distinguish between normal and abnormal sensitive transmissions.

To address these limitations, we propose **LeakSemantic**, a novel approach that combines program analysis and machine learning to identify abnormal sensitive network transmissions more accurately through a better understanding of network semantics. LeakSemantic adopts a hybrid static-dynamic analysis approach to uncover sensitive transmissions (both normal and abnormal). The hybrid approach not only produces better results than a purely static or dynamic analysis approach, but is also able to generate network traffic data in a proactive way, which provides a better characterization of network behavior than widely-used static program analysis based approaches [3, 4, 10, 14]. For example, the hostname of the malicious server in the `PJAPPS` malware family is encrypted as "`ax3mkl4mgele2guoo9f1hc3ohm`" and the real address (`xml.meego91.com`) is only revealed at runtime. Without running the code, static analysis methods fail to decrypt the malicious hostname, which is an important feature for detecting abnormal network transmissions. Instead, the dynamic execution used by LeakSemantic enables tracking runtime information including decrypted hostnames. The traffic data generated are then fed to the machine learning component to build classifiers for detecting abnormal transmissions. Note that our program analysis component can potentially gather more features than just network traffic, which can be useful to differentiate between normal and abnormal flows. We focus on network traffic in this work so that the learning model thus built can be applied even when the app code is not available, e.g., when it is integrated into a network-based intrusion detection system. Thus, LeakSemantic can be used in various settings. When deployed locally, it allows app market operators to identify privacy leakage in an app before releasing it to the market. Moreover, network administrators can benefit from the detection model constructed by LeakSemantic to protect users from unintended transmissions.

A major challenge of program analysis for mobile apps is how to achieve both accuracy and precision. Static program analysis examines the program dependencies in mobile apps without actually executing them. Because of its static nature, it cannot handle reflective calls whose target class or method name is concatenated at runtime, and loading code

dynamically is becoming more common [13]. Static analysis also introduces false alarms as an over-estimated method. In contrast, dynamic analysis chases the runtime behavior of apps and is applicable even when reflection is present. Unlike static analysis that explores all code paths including infeasible ones, dynamic analysis only proceeds to feasible paths and therefore introduces lower false positive rate. Moreover, it can obtain data that are not available in the static setting, such as network traffic data using encrypted URLs. However, by focusing only on the runtime behaviors, dynamic analysis suffers from insufficient coverage and hence false negatives.

Recent research efforts aim to combine static and dynamic program analysis to ameliorate the above problems [18, 21, 22]. We continue this line of research and propose a novel design of hybrid program analysis. LeakSemantic adopts light-weight static analysis to flag potential vulnerabilities, and creates an environment to dynamically confirm the suspicions. Our static analysis provides precise modeling of the call relationships inside an app component, which is crucial for the integrated dynamic analysis component. We introduce a new execution trace generation technique that enables LeakSemantic to uncover malicious behaviors on which previous studies would fail. As we will show in Section III-B, it is insufficient to simply identify code paths leading to targeted APIs. To this end, LeakSemantic dynamically spreads the code coverage and computes the appropriate traces to trigger stealthy behaviors. It also takes into account various sources of unknown variables with an effective handling of unknowns, which further reduces the number of false negatives.

To summarize, this paper presents the following contributions:

- We propose a novel hybrid static-dynamic program analysis technique to locate sensitive network transmissions in mobile applications. Our approach not only enables better accuracy and precision, but also helps derive more detailed features, e.g., traffic URLs, that are important for network behavior analysis.
- We present the design and implementation of LeakSemantic, a detection system that combines program analysis and machine learning to identify networking related abnormal behavioral patterns. Instead of classifying a whole app as malicious or not as most previous work does, our approach is able to distinguish malicious behavior from normal behavior within an app. We also show that network-level detection can benefit from the information provided by program analysis.
- We evaluate the effectiveness of LeakSemantic using two micro-benchmark suites and 1404 real-world apps. Our hybrid program analysis produces better results than any of the three state-of-the-art taint analysis tools used in evaluations. Experiments further show that LeakSemantic is fast and cheap, allowing it to identify true threats inside the real apps with high accuracy.

The rest of the paper is organized as follows. We highlight our system overview in Section II. The technical details are included in Section III. After presenting the system implementation in Section IV, we show the evaluation results in Section V. We discuss the limitations in Section VI and the related work in Section VII. Finally, Section VIII concludes the paper.

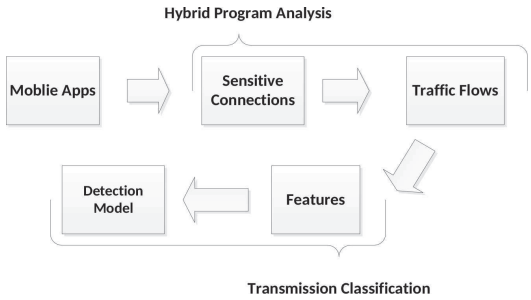

Fig. 1: System Architecture

II. Overview

Figure 1 depicts the architecture of LeakSemantic. From the datasets of authentic apps and malwares, our system proceeds in the following steps:

- **Hybrid Program Analysis:** The phase of hybrid program analysis precisely identifies and characterizes the leaking connections in the target app. We first perform static analysis to retrieve the call graphs of the corresponding app. To better model the lifecycles of app components and runtime events, we create `dummyMain()` for each component. The invocations of sensitive APIs (sources) that collect private data with their entry points are identified through traversal of the graphs. We then construct execution traces and run the program from the set of traces. The information flow analysis is performed during the execution. If a connection point (sink) is reached, we record the dynamic data of the communication. To achieve better coverage, we have designed methods to generate execution traces and handle unknowns encountered during runtime.
- **Transmission Classification:** Having extracted traffic information about the sensitive connections, we then derive a set of features that can be used by the anomaly detection system. Concretely, we concentrate on building machine learning classifiers using lexical features derived from URLs. Our novel design enables us to build models for both host-based and network-based detection.

III. LeakSemantic

To model the runtime behavior of apps while achieving good coverage, we use a hybrid program analysis that combines static analysis and dynamic analysis. In Android, a medium-sized app can contain dozens of components and thousands of methods. Dynamic traversal of all possible paths is expensive and infeasible in practice. Our approach leverages light-weight static analysis to locate invocations of sensitive APIs and the corresponding components. The output of static analysis will help guide dynamic analysis. Machine learning models are then constructed with the flows derived by dynamic analysis. It is crucial that LeakSemantic can generate sensitive flows with decrypted URLs. Finding 1 in Section V states that the

detection ratio decreases obviously if the training data does not cover sufficient characteristics of the malicious flows.

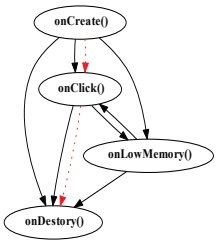

Fig. 2: The call graph of `Activity1` modeled by the corresponding `dummyMain()`. The solid lines indicate call relationships among the callbacks and the dashed lines specify one possible execution trace on the call graph.

A. Static Analysis

Static analysis is responsible for constructing the call graph of the target app, which guides the upcoming dynamic analysis. Unlike (desktop) programs written in `C` that contain a main function as the entry point of the execution, Android applications do not contain a single main method. Instead, they are composed of multiple components, where each `Activity` or `Service` component is a `Java` class and has its own lifecycle and event listeners. The lifecycle models transitions such as creation, pause, resume, and termination, between the states of a component. Event listeners allow applications to respond to various types of runtime events such as UI interactions or receiving SMS. The lifecycle and event listeners are constructed from the corresponding callback methods and every callback can be treated as an entry point because they are implicitly called by the Android framework.

To construct call graphs of applications, previous work typically creates one or more dummy main routines that are shared by multiple components. For example, FlowDroid [1] creates a single dummy main for the entire application and all components share that main. AppAudit [22] introduces a shared dummy main for all components of the same category (`Activity` or `Service`). However, analyses starting from a shared dummy main may include components that do not contribute to leakage. Moreover, a shared dummy main blurs the connections between event listeners and components. It is possible that an event listener may be linked to the wrong component so that the latter can directly invoke the former during the analysis, even though this would not happen in a real setting. Instead of constructing a shared dummy main, we let each component have its own dummy main to eliminate the confusion and alleviate the overhead of dynamic analysis. Each component thus has a call graph (an example is given in the Figure 2). The event listeners such as `onClick()` and `onLowMemory()` embedded with the component are registered after `onCreate()`. `onClick()` is a UI callback that is invoked once the corresponding buttons are clicked, whereas `onLowMemory()` is called once the available memory of the device is lower than a threshold value.

Listing 1: An example component

```
1  class Acvitity1 extends Activity {
2       String url = ""
3       String imei = "";
4       String tmp = "";
5
6       void onCreate() {
7         /* initiate the activity */
8         ...
9         url = "gongfu188.com";
10      }
11
12      void onClick() {
13        tmp = <get phone
              manager>.getDeviceId(); // source
14      }
15
16      void onLowMemory() {
17        url = url.concat(imei);
18        URLConnection conn = new
              URL(url).openConnection(); // sink
19        imei = tmp; // tainted
20      }
21
22      void onDestroy() {
23        /* finish the activity */
24        ...
25      }
26 }
```

A **source** is an invocation of an API provided by the Android framework to retrieve the sensitive information from the underlying device. We use the list from Susi [17] to locate the sources. An example source is the invocation of `getDeviceID()` at line 13 shown in Listing 1. The program is inspired by `EventOrdering1` in DroidBench [1]. For each source, the corresponding entry point of the component in the call graph is extracted with applying a graph traversal algorithm on the call graph. For instance, the entry point `onClick()` of the component `Activity1` in Listing 1 is located through breadth-first search beginning with `getDeviceID()` on the call graph. The entries with relevant call graphs serve as the starting points of dynamic analysis. We will explain this in detail in the next subsection.

B. Dynamic Analysis

The dynamic analysis component of LeakSemantic consists of an executor with a taint analysis module and a simulation of the Android runtime. The executor is our own version of the Dalvik virtual machine. It is able to directly unpack Android package files and execute the bytecode instructions. We feed a set of traces to the executor. The execution traces are derived not only from the results of the static analysis, but also from the execution procedure itself. The novel design enables capturing the misbehavior missed by state-of-the-art approaches, which we will discuss in detail later. During the execution, whenever a sensitive source API is invoked, the taint analysis module starts to track the propagation of sensitive values associated with the source API. When one or more sensitive values reach a network connection API call (a **sink**) such as `URL/openConnection()` in line 18 in Listing 1, which implies that the transmission is **sensitive**,

the corresponding runtime information such as the network traffic data is recorded. We adopt general taint policies used in previous work [5, 22] to specify the propagation procedure. For example, one rule set x is tainted as long as one of the operands in the instruction "$x = y$ binop z" is tainted. To improve the accuracy of the data flow analysis, we have further developed libraries to emulate the fundamental behaviors of the Android runtime. The implementation details are described in Section IV. In the following, we discuss how LeakSemantic constructs execution traces and how it handles unknown values during the analysis.

1) Execution Trace Generation: We leverage the outcomes of the static analysis phase to derive a set of *basic* execution traces, where each trace is a sequence of specific API calls beginning with a lifecycle callback and ending with an APT call where a source is triggered. For instance, for the entry point `onClick()` in `Activity1`, LeakSemantic builds an execution trace `onCreate()` → `onClick()` that informs the executor to invoke `onClick()` after calling `onCreate()`. The execution trace is generated by applying depth-first search to find a path from `onCreate()` to `onClick()` in the call graph (Figure 2). The default values of global variables are normally initialized at the lifecycle callbacks such as `onCreate()` and `onStart()`. We choose to execute from these callbacks to reduce unknown variables, which in turn reduces unknown branches that need to be explored and improves the efficiency of dynamic analysis. Properly modeling the unknowns is challenging in general and will be discussed in more detail in the following subsections. In addition to reducing unknowns, our approach also enables LeakSemantic to generate more complete URLs, which is important for building accurate classifiers (see Section III-C). As we can see in Listing 1, the connection in line 18 can only be correctly triggered if `url` is properly assigned with the hostname in line 9.

The *de facto* hybrid analysis approaches such as AppAudit, Harvester [18] and IntelliDroid [21] only use code paths with certain code locations (e.g., a sink) and terminate the analysis once one such location is reached. However, reachability alone does not necessarily imply the exposure of true malicious behavior. Reconsider the code snippet shown in Listing 1. A direct invocation of `onLowMemory()` does not lead to a leakage since the argument of the sink in line 18 may have an empty `imei`. Given that `tmp` is tainted in `onClick()`, the correct order to trigger a real leakage is to invoke `onLowMemory()` twice. The corresponding execution sequence can be represented as `onCreate()` → `onClick()` → `onLowMemory()` → `onLowMemory()`.

To correctly generate the set of execution traces that trigger the actual leakages (or other types of abnormal behavior), we parse the code of the executable callbacks to determine whether they contain statements that read the corresponding newly tainted variables. A new execution trace is then constructed by expanding the existing trace with relevant callbacks. For instance, after executing the trace `onCreate()` → `onClick()`, `onLowMemory()` is identified since it reads the value from the tainted variable `tmp`. A new execution trace `onCreate()` → `onClick()` → `onLowMemory()` is created. Similarly, LeakSemantics constructs `onCreate()` → `onClick()` → `onLowMemory()` → `onLowMemory()` once finishing running `onCreate()` → `onClick()` → `onLowMemory()`. We can set a threshold on the number of execution traces to save analysis time in practice.

Listing 2: A logic bomb

```
1  String mRun = getSearchTask(); // commands
2  void doSearchTask() {
3     if (mRun == null) {
4        reportState(1);
5        if (mRun != null) {
6           runPackage(mPkgName); // leak
7        } else {
8          ...
9        }
10    } else {
11       ...
12    }
13 }
```

2) Sources of Unknowns: During the execution, the dynamic analysis may encounter unknown variables that have no explicit assigned value to the executor. As mentioned earlier, running from `onCreate()` alleviates the issue through initializing the component as completely as possible.

In addition to the above mentioned unknowns, we observe that there are many cases where the accurate value of a variable is dependent on the runtime context, which can be categorized as follows:

- User input: input from end users during the interactions with the user interface;
- Device status: the real time status, such as WiFi on/off and the power level, of the underlying device;
- Natural environment: e.g., current temperature, coordinate and time;
- Incoming information: the content of the SMS and the network responses received while using the app.

Malicious apps may hide their behavior by leveraging some of the factors mentioned above to create malicious code that is only triggered under certain circumstances. For instance, `RCSAndroid` waits for incoming SMS messages and checks whether these messages contain specific commands and then decides whether to transmit the user data [6], and the `DroidDream` malware family triggers its malicious payload only at night [23]. As another example, consider the code shown in Listing 2, which comes from a malware sample of the `DroidKunfu1` family. In line 1 the program contacts a remote control server and retrieves the commands into `mRun`. `reportState()` is responsible for collecting user private data and it is only triggered when the malicious server replies with certain characters. In other words, the dynamic context causes the executor to generate different outcomes even for the same input trace. To detect such malicious behavior, it is therefore important to treat those variables whose values vary over the context as unknowns.

3) Handling of Unknowns: To represent the set of variables with unknown values, we maintain a symbolic state σ that maps variables to symbolic expressions, and a symbolic path

constraint PC, which is a quantifier-free first-order formula over symbolic expressions. Both σ and PC are updated during the course of execution.

A conditional statement such as `if` inside the target program may contain unknown values in its conditions. Unknown branches during the execution interrupt the execution since the executor does not know which direction to explore. Instead of always following one path, which increases false negatives significantly, LeakSemantic adopts a depth-first search scheme while taking the symbolic path constraints of unknown variables into account to reduce the search space.

More specifically, whenever an unknown branch is encountered, LeakSemantic creates a snapshot to store the state of the executor and pushes the snapshot onto a stack *SnapStack*. The snapshot consists of a copy of the current running context including the program counter and the values in the stack and the heap, which enables the executor to restore the environment after the unknown branch is processed and continue the analysis where it was left off. The executor then explores each direction under the branch one by one, while using *SnapStack* to save and restore the environment.

Consider again the code shown in Listing 2. The execution starts with an empty symbolic state and a symbolic path constraint $true$. As a result, $\sigma = mRun \mapsto mRun_0$, where $mRun_0$ is an initially unconstrained symbolic value. At every unknown conditional statement *if* (e) *then* S1 *else* S2, PC is updated to $PC \wedge \sigma(e)$ for the *then* branch and $PC \wedge \neg\sigma(e)$ for the *else* branch. For instance, at the unknown condition in line 3, a snapshot of the executor is saved. The executor first updates the PC to $mRun_0 \neq null$ and explores the *else* branch of the condition. Once the execution terminates, it restores the status from the snapshot and proceeds to the *then* branch of the condition in line 3 with PC updated to $mRun_0 = null$. The branch consists of a method `reportState()` that stealthily exposes user's private data, and another unknown condition (line 5). The procedure to handle the second unknown condition is similar to the first one. In this case, however, the *then* branch has the path constraint $mRun_0 = null \wedge mRun_0 \neq null$ leading to an infeasible path. Therefore, the executor ignores the *then* branch and only explores the *else* branch.

Code containing loops or recursion may result in an infinite number of paths to be explored if the termination condition for the loop or recursion is symbolic. Consider the code snippet shown below:

```
1       String[] x = getHttpResponse();
2       int i = 0;
3       while (!x[i].equals("")) {
4         i++;
5       }
6
7       if (i > 3 && i < 10) {
8         transmit(longitude, latitude);
9       }
```

Since we do not know exactly how the server will respond in line 1, the content and the length of string array `x` should be treated as unknown, leading to an infinite number of code paths. To address this problem, previous studies [18, 22] simply set thresholds on analysis time or the number of visited instructions. However, these approaches may lead to an incorrect value of `i` after the loop, which should be equal to the actual length of `x`. Importantly, the value of `i` is used to determine whether to trigger the leakage in line 7.

Instead, we execute the block under the loop only once and mark all the variables that accept new values within the block. After exploration of the block, the tagged variables will be modeled symbolically for the rest of the execution. By treating `i` as a symbolic `Integer` with constraint $i > 3 \wedge i < 10$, the sensitive transmission in line 8 will be successfully reached. We also introduce some heuristics to further mitigate the issue of path explosion, which will be discussed in Section IV.

C. Transmission Classification

Using the traffic flows generated by the dynamic analysis component, we formulate the detection of abnormal sensitive transmissions as a classification problem. LeakSemantic uses a supervised learning approach to train classifiers that can be used by host-based or network-based intrusion detection systems. Specifically, we focus on **lexical features** derived from the set of URLs in the traffic traces. Lexical features often contain useful patterns to distinguish between suspicious and benign traces. URLs such as `gad.ju6666.com/GetAd?&lo=(.*)` and `api.openweathermap.org/forecast?&lon=(.*)`, in which `lo` or `lon` is an abbreviation of "longitude", have the user's location data embedded. The words `GetAd` and `forecast` further provide hints about the purposes of the transmissions: the former URL is sent as a request for advertisement while the latter is composed to retrieve the corresponding weather forecast. An effective detector should be able to report the ad request as suspicious and release the operational weather trace.

We utilize the simple yet powerful "bag-of-words" model [15] that is frequently used in spam detection to derive features inside URLs. LeakSemantic divides a URL into tokens by treating certain characters as separators. Each distinct token is then viewed as a separate feature and every data flow collected is then converted to a vector of binary values. Direct application of "bag-of-words" may produce a very large feature space, which results in a heavy computational cost. As stated in [19], one can limit the size of the feature set by removing tokens that seldom appear in the flows.

IV. Implementation

In this section, we provide further details about the implementation of LeakSemantic. LeakSemantic is mostly written in Java and consists of around 18,600 source lines of code.

LeakSemantic extends a part of FlowDroid for call graph generation. We implemented our own executor with taint analysis support to perform the dynamic analysis mentioned in Section III-B. The executor leverages `PATDroid`[1] to extract bytecode and then interprets each bytecode instruction one by one. During the execution, the sensitive data propagation

[1]https://github.com/mingyuan-xia/PATDroid

is tracked by the taint analysis plugin. Android applications invoke the APIs provided by the Android SDK to interact with the underlying operating system during runtime. However, the official Android SDK is missing critical parts of the Android runtime, which are filled with "stubs" used for compilation. The execution and taint analysis cannot proceed correctly without precisely modeling of the Android runtime. We therefore manually pad the incomplete Android SDK and emulate the core functionalities offered by Android. Our simulation of the Android system is similar to the Android Device Implementation (ADI) used in DroidSafe [9]. But their implementation is purely for static analysis and does not scale well to support our dynamic analysis.

LeakSemantic is currently using the `JaCoP`[2] to represent and update the path constraints. To alleviate the path explosion caused by unknown branches, we heuristically limit the number of unknown variables. We use the API `android.net.NetworkInfo/isConnected()` to illustrate the idea. `isConnected()` reveals the real time connection capability of the device, so that the return value reflects the device status. This should be treated as unknown in theory as mentioned in Section III-B. However, the transmission can be triggered only if the device is connected to the Internet. We therefore force the API call to always return *true* instead.

We also simulate some commonly used third-party libraries to reduce performance overhead. For instance, `com.squareup.picasso` is a widely used open-source package to support downloading and presenting images. Since no misbehavior in it has been detected, we do not check the subroutines called by the package during execution. Instead, we replace methods inside the official packages with our own methods during the execution.

V. Evaluation

We have conducted a comprehensive evaluation of LeakSemantic. In this section, we report the evaluation results and our findings. Our evaluation contains two steps. First, we leverage micro-benchmark suites to evaluate the leakage detection accuracy of our hybrid program analysis module. Second, we apply LeakSemantic to real-world apps and construct classifiers to detect illegitimate exposures for different settings.

A. Benchmark Suites and Quality of Program Analysis

We compared LeakSemantic with the following state-of-the-art taint analysis tools:

- Andrubis [13] is a dynamic analysis sandbox based on TaintDroid. It generates nearly 8,000 pseudo-random streams of external events and monitors the behavior of the target app for 240 seconds[3].
- FlowDroid is a flow-, field-, and object sensitive static program analysis framework. The original FlowDroid cannot track information flows across separate components. We integrated FlowDroid with Epicc [12] to partially support inter-component communications.
- AppAudit is a hybrid taint analysis approach similar to LeakSemantic. It also uses static analysis to mark potential leaking methods, and then prune candidate methods through dynamic analysis. But the way it generates call graphs and models the unknown variables is different from LeakSemantic.

We executed LeakSemantic on a computer with an Intel Core CPU E8500 @ 3.16GHz and 2GB of heap memory for the JVM. Since Andrubis has fixed analysis time and AppAudit does not provide installation package to run locally, it is hard to compare the running times of the set of tools directly. However, we observe that LeakSemantic exhibits good performance on the apps with short analysis time.

We evaluated the detection accuracy of the above tools using the following two micro-benchmark sets. LeakSemantic spent 12.4s on average for each app and FlowDroid took an average of 13.2s per app:

1) DroidBench: DroidBench[4] is an open-source benchmark suite that contains a set of hand-crafted apps that exploit various characteristics of the programming language to bypass static taint analysis. It contains 118 apps in total, among which we excluded 10 apps with leakage types unsupported by both Andrubis and AppAudit, such as leaking user input passwords.

Table I summarizes the detection results over DroidBench. We observe that LeakSemantic achieves the best quality among the four taint analysis tools. Precise call graphs and the better handling of unknowns enable LeakSemantic to generate zero false alarms. Among the three baselines, Andrubis performs best and successfully report most leakages. This is because the dataset is originally designed to test static analysis tools and difficulties for static analysis are typically not hard for dynamic analysis. FlowDroid is able to locate more than 75% of leaks. But its over-approximation also leads to the worst precision. Also, FlowDroid is unable to generate runtime data such as traffic flow, and therefore cannot be directly used to build a traffic-based transmission classification model.

Since both AppAudit and LeakSemantic adopt hybrid program analysis, we conducted a more detailed comparison between them. LeakSemantic achieves better detection accuracy for several reasons. First, AppAudit terminates its execution once a sink is touched. As we discussed in Section III-B, reachability alone does not necessarily imply a sensitive transmission. Second, AppAudit does not consider some types of unknowns and always exploits one direction of an unknown branch, which introduces false negatives. Moreover, LeakSemantic provides a more complete implementation of dynamic analysis to support various mechanisms used in Android. In particular, LeakSemantic is able to locate event handlers registered in the layout configurations and track the communications among multiple components. AppAudit does not support any of these Android features. Last, as we mentioned in Section III-A, the inaccurate model of call graphs used by AppAudit increases its false positives.

[2]https://jacop.osolpro.com/

[3]The official Andrubis service is no longer available. We installed TaintDroid on a real device and composed scripts to create an environment similar to Andrubis.

[4]The up-to-date stable release is DroidBench 2.0 (https://github.com/secure-software-engineering/DroidBench/tree/master). We replaced all the sinks with network transmissions since Andrubis and AppAudit do not treat certain sinks as sensitive in some apps.

TABLE I: Detection results on DroidBench

Tools	Missed Flows	Accuracy	FP	Precision
Andrubis	15	84.2%	0	100%
FlowDroid	22	76.8%	10	56.6%
AppAudit	56	41.1%	2	91.3%
LeakSemantic	2	97.9%	0	100%

FP = False Positives

TABLE II: Accuracy on BombBench

Tools	Missed Flows	Accuracy
Andrubis	21	4.5%
FlowDroid	14	36.4%
AppAudit	12	45.5%
LeakSemantic	1	95.5%

LeakSemantic (and all the three baselines) misses two flows that involve inter-application communications, which requires modeling the behaviors across multiple apps. None of the existing taint analysis tools can detect this kind of collusion attack. Another unresolved challenge of LeakSemantic is *control flow dependent taints*, also a well acknowledged drawback in most taint analysis tools [22].

2) BombBench: BombBench[5] is another open-source benchmark that contains 22 apps to test taint analysis tools. Each app takes advantage of a kind of *logic* or *time* bomb inspired by previous studies [18, 23, 25] to conceal a sensitive flow. We show the results in Table II. LeakSemantic identifies most leaks among all the four tools. We can see the sharp decrease of accuracy in Andrubis, which indicates that current random-events based testing toolkit is not powerful enough to cover complicated program logic. Its limitation is fundamental and cannot be simply settled with extension of analysis time. For example, `DevInfo2` triggers its payload only under certain system language. Because, unlike LeakSemantic, they do not count as unknown the variables obtaining values from `Locale/getDisplayName()`, both Andrubis and AppAudit fail to capture the disclosure flow. We notice that FlowDroid also could not successfully mark this case, which may be caused by inaccurate modeling of system functions. LeakSemantic missed one flow because of a variable implicitly assigned by a user-driven event. Although we model variables who read the values from the UI-related API calls such as `EditText/getText()` as unknowns, currently we do not directly view the variables modified by the callbacks such as `onClick()` as unknowns even they are correlated with user interactions. We do this for performance concerns since there might be plenty of variables influenced by the callbacks in real apps. Excessive amount of unknowns leads to the exponential size of code paths needed to be explored.

B. Real Apps and Transmission Classification

We then applied LeakSemantic to build a traffic classification model using real apps. From the traffic generated by our hybrid analysis tool, it is possible that multiple code paths lead to the same connection, which results in separate transmissions with an identical URL. We merged these transmissions with the same URL into a single one within the target app.

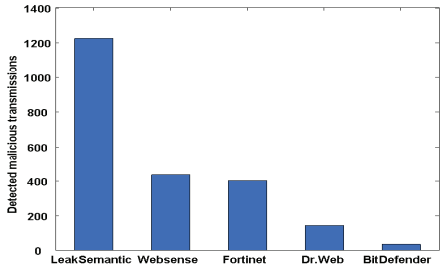

Fig. 3: Detected malicious sensitive transmissions.

We first collected malicious sensitive transmission from the Android Malware Genome project, which contains 744 leaking malwares [26]. LeakSemantic extracted 1223 malicious sensitive transmissions and collected the corresponding traffic. We first observe that these malicious transmissions cannot be correctly identified by existing commercial anti-virus solutions, which motivates the need for a new detection approach. To this end, we uploaded the URLs of these transmissions to `VirusTotal`[6], a popular website that scans submitted URLs with latest 68 anti-virus engines. Surprisingly, 64 out of 68 engines did not report any alarms regarding the transmissions. Figure 3 presents the detection results by the rest 4 engines. `Websence` identified relatively more malicious URLs (436, or 35.7%), but the number found is still far from 50% of all malicious connections.

We then ran LeakSemantic on 660 apps crawled from the categories that have legal sharing functionalities in app markets[7]. Among them, LeakSemantic recognized 1056 sensitive transmissions. The average analysis time for each app is 135.3s, including the 744 malwares and the 660 authentic apps. For each flow collected, we examined the destination host name. If the host name belongs to an advertisement or analytics server, we marked the flow as illegal. We then checked the plain text content delivered through the flow to see whether the response sent by the server is related to the sent user data or not. There are cases in which the communication between the phone and the server are encrypted. We leveraged instrumentation and reverse engineering to block those flows. We reran the modified app to see how blocking influences the app. The flow was labeled as legal when the app's functionality is affected. Out of 1056 transmissions, 791 did not affect the app's functionality, so we labeled them as illegitimate. The other 265 operational sensitive transmissions were collected from 183 apps.

We used the labeled 2279 transmissions as training and testing data with ten-fold cross-validation [11], which is a standard approach for evaluating machine learning solutions. We applied **Decision Tree** as the learning classifier for LeakSemantic since it is commonly used in traffic classification [16, 19]. As mentioned earlier in Section I, LeakSemantic can be deployed as a host-based or network-based detection system.

[5]https://github.com/bombbench/BombBench

[6]https://www.virustotal.com/

[7]Google Play (https://play.google.com/store/apps) and Baidu App Market (http://shouji.baidu.com)

TABLE III: Host-based Classification Results

Class	TP Rate	FP Rate	Precision	F-measure
Illegal	0.938	0.063	0.974	0.956
Legal	0.937	0.062	0.856	0.895

TABLE IV: Network-based Classification Results

Class	TP Rate	FP Rate	Precision	F-measure
Illegal	0.915	0.095	0.916	0.915
Legal	0.905	0.085	0.904	0.904

TP = True Positive, FP = False Positive

We conducted two experiments that reflected the effectiveness of LeakSemanic in different scenarios. When LeakSemantic is configured in a single host system, it automatically finds the disclosure points and then picks the illegal instances based on the flows generated. The classifier at host-level involves only the flows of sensitive transmissions; the detection model at network-level should be able to filter out the innocent flows that do not carry any sensitive data.

1) Host-based Detection: Table III shows that LeakSemantic has high precision and F-measure in identifying illegal transmissions[8]. After manually inspecting the misidentified instances, we found that their URLs were very similar to the benign addresses. Also, they put the sensitive data into their body rather than the URL, which makes the URL-based detection more difficult to correctly label them. We note that LeakSemantic is able to collect more information than URLs. We plan to consider more features to further reduce the false negatives in the future.

2) Network-based Detection: Based on the sensitive transmissions we collected, we added the non-sensitive traffic flows to the legitimate class. This reflects the real environment of the network-based detection. Table IV summarizes our results. As we can see, the prediction incurs a slight loss in accuracy compared to the results of the host-based detection. This is expected as the addition of non-sensitive flows makes the learning task more challenging.

During the experiments, we also observed the following interesting phenomena:

Finding 1: Among the 1223 malicious leaking transmissions extracted from the malware dataset, we found that 69.7% of the transmissions used encryption to hide the hostnames. Malware leverages encryption to evade traditional signature-based detection approaches. As mentioned earlier, encryption also hinders pure static analysis from explicitly detecting the target behaviors. Without enough dynamic information, the intrusion detection systems failed to locate many malicious transmissions. To illustrate how important the decryption is, we conducted an experiment that trained a model based solely on unencrypted instances and tested the model on the instances with encrypted hostnames. Among the 806 encrypted instances, the model only recognized 578 (71.7%) of them. Compared to the prediction results (91%) shown previously, the accuracy decreased dramatically.

[8]Since the data is heavily skewed towards the illegal class, we used SMOTE [2] to over-sample the legitimate class.

Finding 2: LeakSemantic identified more than 1223 malicious transmissions in the malware dataset. However, it could not properly generate traffic flows for a few transmissions such as those from the `DroidKunfu4` malware family. We manually inspected the code and found that the hostnames of the transmissions are not embedded either in the code or in the resource files of the apps. Instead, the transmissions dynamically retrieve the hostnames from a remote server with the help of the command and control modules.

Finding 3: From the crawled apps, we noticed that 3 connections indirectly leak the private data. Instead of sending the user data directly to a tracing server, they first grab the user's coordinates and query a legitimate popular location server to get the corresponding description. They then transmit the description to a suspicious server. Such behavior suggests the need to track the influence of a connection even when the first connection contacts a legitimate server.

Finding 4: LeakSemantic found no sensitive HTTPS connections in the malwares. However, 27 illegitimate HTTPS transmissions were identified in the authentic apps and they were all built by third-party ads/analytics libraries. Although sensitive HTTPS connections are not popular at the current time, we foresee the necessity of inspecting HTTPS connections with the techniques such as `SSLsplit`[9] in the future.

Finding 5: We found that more than 60% of the 183 apps that have legitimate sharing connections also contain illegal transmissions inside for ad or analytics purposes. We also found a weather application that only transmits users' location data to ad servers. It is highly probable that the users of these apps will grant the app the permission to access sensitive resources without knowing their private data will be collected stealthily by unintended servers.

VI. Limitations

Our approach has the following limitations:

- If an adversary knows our approach, he could obfuscate the flows to match our criteria. We envision that more features need to be considered in the future.
- The technique most closely related to our dynamic analysis is *concolic testing* [8], which also leverages both concrete and symbolic values to proceed its execution. Our approach inherits its path explosion limitation; the size of code paths is exponential in the number of unknown branches. We currently remove most unnecessary unknowns with our specific preprocessing and we will look into more advanced relevant techniques soon.

VII. Related Work

Dynamic and static taint analysis track sensitive data flows in programs. TaintDroid [5] modifies the Dalvik virtual machine to monitor potential leaks at runtime. It only identifies leakage that is actually triggered during execution, thus requiring a driver with good code coverage. The static analysis tools FlowDroid [1] and DroidSafe [9] overcome the coarse-granularity through over-approximation. But they also suffer

[9]https://www.roe.ch/SSLsplit

from imprecision by visiting code paths that are not actually feasible. AppAudit [22] leverages hybrid static-dynamic analysis in order to keep the advantages and avoid the drawbacks of both. It only examines code paths determined statically and explores one path when it encounters an unknown branch. In contrast, our system dynamically extends the code coverage and explores as many paths as feasible when an unknown branch is found. ReCon [19] is a solely network-based detection that learns patterns from traffic traces, which is similar to the transmission classification used in LeakSemantic. Our program analysis approaches can further improve the performance of network-based detection. All above approaches treat any exposures of user data as illegitimate, which obscure the true threats through generating large number of false alarms.

AppIntent [24] first stresses the necessity to justify the sensitive transmissions in apps. Bayesdroid [20] proposes a solution by treating the transmissions that carry less accurate information as legal. However, a transmission could be very harmful even if it only contains coarse information since it can collude with others. FlowIntent [7] leverages front-page user interfaces to discriminate location-sharing communications. Its effectiveness depends on the content shown on the pages and its underlying random fuzzing based approach, which is similar to Andrubis [13], makes it hard to locate stealthy malicious payloads. AAPL [14] is a static app auditing tool that queries a commercial recommendation system to rank sensitive disclosures. But as shown in [7], being in the same category does not imply having the same functionality. Other static analysis approaches including AsDroid [10] and DroidJust [3] only treat connections that do not influence the user-observable phone states as malicious. But a flow can still be malicious even it leads to visible changes as it can also trigger the underlying malicious payload simultaneously. LeakSemantic looks beyond the mere surface of leaks by examining their intention based on the corresponding traffic flows.

VIII. Conclusion

In this work, we developed a prototype called LeakSemantic that can identify suspicious sensitive network transmissions from mobile apps automatically. Its hybrid program analysis component enables it to provide better accuracy and precision than other state-of-the-art taint analysis approaches. LeakSemantic further constructs machine learning classifiers to differentiate among the disclosures based on features derived from the program analysis. Our evaluation on 2279 sensitive connections collected from real-world 1404 apps shows that LeakSemantic achieves a detection accuracy of 91%.

IX. Acknowledgements

The effort described in this article was partially sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Contract Number W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon.

References

[1] 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 *PLDI*, 2014.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[3] X. Chen and S. Zhu. Droidjust: automated functionality-aware privacy leakage analysis for android applications. In *WiSec*, 2015.

[4] H. Choi, J. Kim, H. Hong, Y. Kim, J. Lee, and D. Han. Extractocol: Autoatic extraction of application-level protocol behaviors for android applications. In *SIGCOMM*, 2015.

[5] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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 *OSDI*, 2010.

[6] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *S&P*, 2016.

[7] H. Fu, Z. Zheng, A. K. Das, P. H. Pathak, P. Hu, and P. Mohapatra. Flowintent: Detecting privacy leakage from user intention to network traffic mapping. In *SECON*, 2016.

[8] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[9] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.

[10] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *ICSE*, 2014.

[11] R. Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, 1995.

[12] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *ICSE*, 2015.

[13] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *BADGERS*, 2014.

[14] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS*, 2015.

[15] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Beyond blacklists: Learning to detect malicious web sites from suspicious urls. In *KDD*, 2009.

[16] A. Raghuramu, H. Zang, and C.-N. Chuah. Uncovering the footprints of malicious traffic in cellular data networks. In *PAM*, 2015.

[17] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[18] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

[19] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *MobiSys*, 2016.

[20] O. Tripp and J. Rubin. A bayesian approach to privacy enforcement in smartphones. In *USENIX Security*, 2014.

[21] M. Y. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, 2016.

[22] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *S&P*, 2015.

[23] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *ICSE*, 2015.

[24] 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 *CCS*, 2013.

[25] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS*, 2014.

[26] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *S&P*, 2012.