Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2019-IC-001

2019-IC-001

Execution Enhanced Static Detection of Android Privacy Leakage Hidden by Dynamic Class Loading

Yufei Yang, Wenbo Luo, Yu Pei, Minxue Pan, Tian Zhang

Annual Computer Software and Applications Conference 2019

2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

Execution Enhanced Static Detection of Android Privacy Leakage Hidden by Dynamic Class Loading

Yufei Yang†, Wenbo Luo†, Yu Pei§, Minxue Pan†*, Tian Zhang†*
†State Key Laboratory for Novel Software Technology, Nanjing University, China
§Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China
aliasyyf@gmail.com, 151220068@smail.nju.edu.cn, csypei@comp.polyu.edu.hk, {mxp,ztluck}@nju.edu.cn

***Abstract*—Mobile apps often need to collect and/or access sensitive user information to fulfill their purposes, but they may also leak such information either intentionally or accidentally, causing financial and/or emotional damages to users. In the past few years, researchers have developed various techniques to detect privacy leakage in mobile apps, however, such detection remains a challenging task when privacy leakage is implemented via dynamic class loading (DCL).**

In this work, we propose the DL^2 technique that enhances static analysis with dynamic app execution to effectively detect privacy leakage implemented via DCL in Android apps. To evaluate DL^2, we construct a benchmark of 88 subject apps with 2578 injected privacy leaks and apply DL^2 to the apps. DL^2 was able to detect 1073, or 42%, of the leaks, significantly outperforming existing state-of-the-art privacy leakage detection tools.

***Keywords*-Privacy Leakage Detection, Dynamic Class Loading, Taint Analysis, Constraint Solving**

I. Introduction

In the past few years, Android has become the most popular mobile operating system, taking over 80% of the market share [1], and the number of mobile apps targeting the Android platform has grown rapidly. Meanwhile, according to a recent study [2], more and more Android apps need to collect and/or access information like device id, location information, SMS messages, and contacts, to fulfill their purposes. While such apps greatly facilitate our work and daily life, serious concerns have been raised about the risks of them leaking that information.

As one of the security mechanisms, the permission-based framework employed on the Android platform can be used to ensure that the apps' access to certain resource/information is subject to the user's approval. The framework, however, is too coarse-grained in that it is only concerned with *whether*, but not *how*, an app uses the information. To complement the protection provided by such coarse-grained mechanism, various approaches to privacy leakage detection have been proposed and many of them [3]–[14] are based on taint analysis [15].

Certain features of the development language and the execution environment of mobile apps, however, have added to the difficulties in detecting privacy leakage. One important example of such features is dynamic class loading (DCL). DCL is a feature supported by the Android platform, which enables apps to extend their behaviors at runtime by using a class loader to load classes in an explicit fashion—in this paper, we refer to classes that are implicitly loaded during app execution as *internal* and those explicitly loaded through DCL as *external*. While DCL brings great flexibility to Android app development and has been used widely in developing frameworks and plug-ins, Google recommended to use it with caution [16], since classes from sources that are not verified "might be modified to include malicious behavior". In fact, a recent study [17] showed that DCL had been used by malwares to evade the security checks in vetting systems like Google Bouncer.

Based on whether the source or the sink of sensitive information is located in external classes, privacy leakage can be implemented via DCL in one of the following four schemes: *(i)* sensitive information is retrieved in internal code but leaked in external code; *(ii)* sensitive information is retrieved in external code but leaked in internal code; *(iii)* sensitive information is both retrieved and leaked in external code; *(iv)* sensitive information is both retrieved and leaked in internal code.

Existing techniques based on taint analysis do not perform well in discovering leaks implementing schemes *(i)–(iii)*: Since external classes are statically not part of the app under analysis, tools based on static analysis only have limited power in detecting those leaks [3]–[8], [18], [19]; Since most external classes are only loaded and executed along certain program paths, the chance is often low for those paths to be exercised during dynamic analysis [9]–[14], [20]–[24]; The DyDroid technique [25] conducts a privacy tracking analysis by combining static and dynamic analyses. It, however, handles only privacy leaks with both the source and the sink of sensitive information within external classes and cannot detect leaks implementing schemes *(i)* and *(ii)* listed above. In this work, we refer to privacy leaks implementing schemes *(i)–(iii)* as being *hidden* by DCL. Leaks implementing scheme *(iv)* are *not* hidden by DCL, since a conservative static taint analyzer always assuming the external code to propagate taint data can still detect the leaks.

To effectively detect privacy leakage hidden by DCL in Android apps, we propose the DL^2 technique that enhances static analysis with dynamic app execution. Given an Android app, DL^2 first applies static analysis to find paths in the app that lead to invocations to methods from external classes and

*Corresponding author.

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.00029

gather the corresponding path conditions on program variables. Next, the path conditions are solved by a constraint solver and the solutions are used to drive the dynamic execution of the app, during which DL^2 retrieves the external classes loaded and records information about the methods from those classes that are invoked. Finally, DL^2 applies static analysis to both the internal and external code and combines the results to detect privacy leaks hidden by DCL.

To evaluate the effectiveness and efficiency of DL^2, we constructed a benchmark of 88 subject apps based on 25 Android apps collected from the Google Play store and the F-droid repository[1]. In total, 2578 privacy leaks were injected into the subject apps under the guidance of a privacy leak model for auditing anti-malware tools [26]. DL^2 was able to detect 1073, or 42%, of the injected privacy leaks in the benchmark, which is 163%, and 200%, more than the state-of-the-art privacy leak detection tool TaintDroid and DyDroid, respectively. On average, it took DL^2 18.8 seconds to detect one privacy leak.

We make the following contributions in this work:

- We develop the DL^2 technique that enhances static analysis with dynamic app execution to effectively detect privacy leakage hidden by DCL, and implement the technique into an automated tool also named DL^2;
- We construct a benchmark of 88 subject apps with 2578 injected privacy leaks implemented via DCL;
- We experimentally evaluate DL^2 on the subject apps from the benchmark, and compare the performance of DL^2 with that of TaintDroid and DyDroid on the same subjects.

The remainder of this paper is organized as follows: Section II presents a simple example demonstrating how dynamic class loading can be used to hide privacy leakage; Section III explains in detail how DL^2 works step by step; Section IV describes the experiments we conducted to evaluate the performance of DL^2 and reports on the experimental results; Section V reviews recent work related to privacy leak detection for Android apps; Section VI concludes the paper.

II. DCL and Privacy Leakage

In this section, we use a small example from the VirusShare[2] repository to introduce dynamic class loading on the Android platform and to demonstrate how DCL can be used to leak sensitive user information.

Dynamic class loading (DCL) is a mechanism that allows software systems to decide which classes to load during runtime, and it enables applications to be compiled separately from their dependencies and extended dynamically on-demand. DCL is supported on the Android platform, and Android apps can dynamically load classes from files of different formats.

The example shown in Listing 1 illustrates how DCL can be used to load and execute a class under certain conditions

[1]https://f-droid.org/en/

[2]https://virusshare.com/, MD5: 4217d6663656239a53d70e5e7e174adb.

```
1  public class SmsReceiver extends BroadcastReceiver{
2    ...
3    public void onReceive(...){
4      ...
5      String clsName = ..., methodName = ...;
6      SmsMessage msg = SmsMessage.createFromPdu(pdus[x]);
7      String phoneNbr = msg.getOriginatingAddress();
8      if(msg.getMessageBody().contains(KEYWORD)){
9        DexClassLoader loader=new DexClassLoader(...);
10       try {
11         Class cls = (Class)loader.loadClass(clsName);
12         Object instance = cls.newInstance();
13         cls.getMethod(methodName, Object.class)
14            .invoke(instance, phoneNbr);
15       } catch (Exception e) { ... }
16     }
17   }
18   ...
19 }
```

Listing 1: The code example of DCL

```
public class Sender{
  void sendMessage(String phoneNbr){
    SmsManager smsManager = SmsManager.getDefault();
    smsManager.sendMultipartTextMessage(phoneNbr, ...);
    ...
  }
  ...
}
```

Listing 2: The method invoked by DCL

during program execution. Class `SmsReceiver` is registered as a `BroadcastReceiver` for incoming text messages and its method `onReceive` is invoked upon receiving any message. In case the body of a message contains `KEYWORD` (Line 8), the method constructs a class loader (Line 9), loads a class named `clsName` (Line 11), creates an instance of the class (Line 12), and then calls a method with the name `methodName` on the new instance using the message sender's phone number as the argument (Lines 13 and 14).

Ultimately, whether such behavior causes privacy leakage depends on if the sender's phone number is considered sensitive, which class with the name `clsName` gets loaded, and how the phone number is utilized in `clsName.methodName`. In this work, we adopt a flexible design and allow users to decide which behaviors are considered causing privacy leakage. Particularly, a user may specify a list of *source* APIs and a list of *sink* APIs: *All information derived (directly or indirectly) from a source API is considered sensitive, and a piece of sensitive information is said to be leaked if it is used by any sink API.*

Continue with the above example. If method `SmsMessage.getOriginatingAddress()` is a source API, the method actually invoked on Line 14 is `Sender.sendMessage` shown in Listing 2, and method `SmsManager.sendMultipartTextMessage()` is a sink API, we have a privacy leak implemented via DCL. Existing techniques will have a hard time detecting the leak: Static

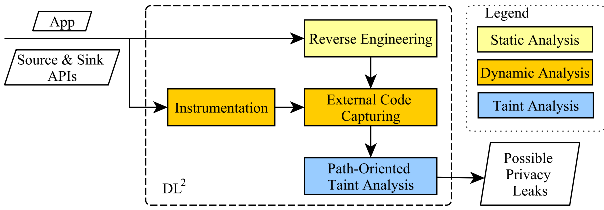


Fig. 1: An overview of DL^2.

analysis of the code in Listing 1 can label variable `phoneNbr` as sensitive, but would not find any suspicious sink for it; The chance for dynamic analysis to detect the leak is also slim, since the sensitive information is only used by a sink API when the condition on Line 8 is satisfied. The privacy leak in the example, therefore, can evade detectors like TaintDroid and DyDroid. In comparison, DL^2 uses several steps to detect privacy leaks in DCL effectively: static analysis aims to enhance dynamic execution of triggering DCL and DL^2 also combines the analysis results on both the internal classes and the external classes, so that it is able to successfully detect the leak. Section III elaborates on how DL^2 achieves that step by step.

III. The DL^2 Technique

An overview of DL^2 is depicted in Figure 1. Given an Android app in the form of an .apk file as well as a list of target source and sink APIs as the input, DL^2 first reverse-engineers the app to Java bytecode, then goes through the following steps to discover privacy leaks hidden by DCL:

1) DL^2 analyzes the bytecode of the app and collects execution paths that lead to reflective invocations to methods from the external code (Section III-A).
2) DL^2 drives the app to execute along the collected paths and records the external classes loaded and methods invoked from those classes (Section III-B).
3) DL^2 applies static taint analysis on both the internal and the external code of the app and combines the analysis results to find privacy leaks (Section III-C).

The following subsections explain the steps in detail.

A. Path Construction

Given an input app P, DL^2 first collects a set of execution paths from P, each from an entry point to an intersection point. The beginning of a life-cycle method or an event handler of a component[3] within P defines an *entry point*—the component is called the *enclosing component* of the entry point; a location where a reflective method call is made on an instance of a dynamically loaded class is referred to as an *intersection point*.

All the following analyses of DL^2 are applied exclusively to these paths. Such design is reasonable, since, given the component-based and event-driven nature of Android apps, most behaviors of P, malicious or not, are triggered by state changes in, or events on, those components. Focusing on execution paths starting from entry points, rather than the beginning of the application entry method, also significantly shortens the paths that DL^2 needs to analyze.

At the implementation level, DL^2 first detects all components of the app by parsing the app's manifest file. Life-cycle methods of those components can then be easily identified by looking for methods with specific signatures in their corresponding classes, and event handlers by analyzing the parameters of calls to APIs that register handlers to event sources. Let $\boldsymbol{E}_P$ be the set of entry points defined by these life-cycle methods and event handlers in P.

DL^2 then constructs a set of finite execution paths, with each path starting from an entry point $e \in \boldsymbol{E}_P$ and covering one execution of e's enclosing method m_e. That is, each path starts from e, or the beginning of m_e, and ends with an explicit or implicit **return** from m_e. During path construction, DL^2 inlines every method directly or indirectly invoked by m_e up to a certain level N_I, constructs a control flow graph (CFG) for the resultant method m'_e, and then enumerates all finite execution paths of m'_e based on its CFG by unrolling each loop for at most N_R times.

Next, DL^2 applies a lightweight static analysis to identify all intersection points covered by the collected paths. The analysis considers a reflective method invocation on path p as defining an intersection point if the target object used in the invocation is instantiated from a class dynamically loaded during the execution along p. By removing segments after intersection points, DL^2 gathers, for each entry point $e \in \boldsymbol{E}_P$, a set $\boldsymbol{P}_e$ of paths that start from e and end at an intersection point. Let $\boldsymbol{P} = \cup_{e \in \boldsymbol{E}_P} \boldsymbol{P}_e$.

B. External Code Capturing

In this step, DL^2 employs dynamic analysis to capture the external code executed along each path from $\boldsymbol{P}$. For that purpose, DL^2 first instruments P. Particularly, it inserts code right before each explicit loading of class so that all the external classes used during an execution are downloaded and stored for the following analysis. It also inserts code to record the dynamic type of the receiver object as well as the signature of each method that is reflectively called.

To actually drive P to execute along a path $p \in \boldsymbol{P}_e$, DL^2 needs to not only trigger the execution of method m_e, but also do that when P is in a properly prepared state. To find out the variable values required by that specific state, DL^2 utilizes the IntelliDroid [27] static analysis tool to gather the path condition c_p of p, encodes c_p into a constraint, and employs the Z3 solver to generate solutions to the constraint.

Next, DL^2 instantiates an instance of e's enclosing component directly from the instrumented app, sets the state of the component, prepares an event object based on the solution to c_p, and then executes method m_e by firing the corresponding event on the component. In this step, both component instantiation and event triggering are easily achieved via Android Debug Bridge (ADB), but in general, we cannot

[3]A component of an Android app is either an activity, a service, a broadcast receiver, or a content provider.

modify the state of a component or an event object by directly assigning to their fields. To prepare the component and the event object as required, DL^2 first employs IntelliDroid to collect `public` methods from their classes that manipulate the fields, and then explicitly invokes those methods with the desired values as arguments. Since there is no guarantee such method invocations can achieve the intended modifications without causing undesirable side-effects, DL^2 also records the trace of the actual execution of m_e: If the trace matches with p, we confirm p represents a feasible execution path and the external classes downloaded during p's execution will be used together with the internal code of P for privacy leak detection in the next step. Otherwise, DL^2 discards both the trace and p.

At the end of this step, DL^2 has collected a set $\mathbb{P}$ ($\mathbb{P} \subseteq \boldsymbol{P}$) of paths. Each path $p \in \mathbb{P}$ starts from an entry point e_p and ends at an intersection point i_p, where method M_p from the external class C_p is invoked through reflection.

C. Path-Oriented Taint Analysis

In this step, DL^2 applies a three-phase analysis to each path $p \in \mathbb{P}$ to determine if an execution along p may cause privacy leakage implemented in schemes *(i)*, *(ii)*, or *(iii)* described in Section I.

In phase one, the analysis aims to find out if any sensitive information is propagated via p to the intersection point i_p and used to call M_p. It first marks all values directly returned by source APIs along p as tainted, then iterates through every statement on p to propagate the taint, as done in common static taint analysis [15]. The analysis in phase two mainly focuses on M_p. It checks 1) whether M_p implements any instance of privacy leak by itself, 2) whether the taint in M_p's formal arguments will propagate to the invocation to any sink APIs, and, when applicable, 3) whether the return value of M_p is tainted. If condition 1) is true, DL^2 has detected a privacy leak implemented in scheme *(iii)*; If condition 2) holds and the values used in calling M_p are tainted at i_p according to phase-one analysis, we have a privacy leak implementing scheme *(i)*; If condition 3) is satisfied, a phase three analysis is applied to find out if the propagation of the taint in M_p's return value to an invocation of sink API is feasible in the internal code of the app. An instance of privacy leak implementing scheme *(ii)* is detected, if a path realizing such propagation is found.

DL^2 employs the FlowDroid tool [4] to carry out all the taint analysis in this step. FlowDroid is a state-of-the-art taint analysis tool and it has been successfully used for data leak detection. The original implementation of FlowDroid analyzes paths starting from the launch-point of an app, i.e., the `onCreate` method of the app's launcher activity. We modified FlowDroid so that it analyzes paths starting from any location of the app and accepts as the input a list of target source APIs—all information returned by the APIs is tainted by definition.

D. Other Implemention Details

We have implemented the DL^2 technique into a tool, also named DL^2. DL^2 runs on apps packaged in APK files. Since no source code of the app is needed, the technique can be applied to a wide range of apps and benefit users in different scenarios. To unpack bytecode files, manifest file, and layout files from APK files, DL^2 leverages the open source ApkTool[4]. During path construction (Section III-A), DL^2 flattens the enclosing method of every entry point by inlining methods directly or indirectly invoked within $N_I = 3$ levels, and unrolling loops for at most $N_R = 5$ times. Such design is motivated by the "small-scope hypothesis" [28], which states that many defects can be triggered using short executions. As explained in Section III-B, DL^2 also utilizes IntelliDroid to construct path conditions and the $Z3$ constraint solver [29] to find solutions to the path conditions. The Soot [30] bytecode manipulation and optimization framework is used to instrument the apps.

IV. Evaluation

We conducted experiments on DL^2 to evaluate its effectiveness and efficiency. This section reports on the experiments and findings.

Our evaluation aims to address the following research questions:

- **RQ1:** How effective is DL^2 in detecting privacy leaks hidden by DCL?
- **RQ2:** How efficient is DL^2?
- **RQ3:** How effective is DL^2 in triggerring DCL?

In the experiments, Android apps were executed on a Google Nexus emulator running Android 4.3. Both DL^2 and the emulator ran on a desktop PC running Windows 10 Professional on a 2.30GHz Intel Core i5-7360U processor and 8GB DDR3 memory.

A. Data Set

A dataset often used to evaluate malware detection tools is Drebin [31]. We, however, concluded that Drebin is not suitable to be used to evaluate DL^2 after examining 50 apps randomly selected from the dataset: 23 apps out of the 50 failed to launch successfully; Although 24 others use DCL in their implementations, manual inspection of the code reveals that the mechanism is mainly used to load advertisements and no privacy leak is involved. We, therefore, constructed our own benchmark by injecting privacy leaks into existing apps.

The construction of privacy leaks was guided by the malware meta-model summarized in the work of Mystique-S [26]—the meta-model was used by Mystique-S to modularize common attack behaviors. Based on the model, Mystique-S first selects attacks according to the user scenario, then the selected attacks will be used to guide the model-driven generation of malicious code for the server side, and in the end, the malicious code is delivered to the user device and loaded

[4]http://ibotpeaches.github.io/Apktool/

TABLE I: Potential source and sink API categories.

	CATEGORY	ID	EXAMPLE
Source	location	loc	`LocationManager.getLastKnownLocation()`
	account	acc	`AccountManager.getAccounts()`
	contact	con	`ContentResolver.query(...)`*
	phone state	phn	`TelephonyManager.getDeviceId()`
	browser	brw	`ContentResolver.query(...)`*
	audio	aud	`ContentResolver.query(...)`*
	sms	sms	`SmsMessage.getMessageBody()`
Sink	internet	net	`HttpClient.execute()`
	file	fil	`OutputStream.write(...)`

* Different information is accessed depending on the actual arguments used to call the method.

dynamically by the application. Through DCL, Mystique-S was able to modify apps at runtime to execute different malicious behaviors. Since Mystique-S is not available for download, we followed its idea and applied Soot [30] to inject privacy leaks into real-world apps based on the same malware model automatically, as described below.

We first download the top 50 most popular apps from the Google Play store and 40 apps from the F-Droid repository that have more than 500 stars each. From these apps, we prune out the ones that 1) cannot run on our experiment emulator successfully after instrument because of signature protection, 2) do not have the required permissions to access any sensitive information, according to their manifest files, or 3) are games[5]. In the end, we are left with 25 Android apps. Table II lists for each original app the name (APP), the ID (ID), the top-level package (PACKAGE), the version (VER), and the size in numbers of classes (#C), methods (#M), entry points (#EP), and line of code (LOC).

Next, we analyze the permissions granted to each selected app and identify categories of source and sink APIs as the following: source APIs are those that access various types of sensitive information, while sink APIs are those that send information via internet or store information to external storage. Table I lists the relevant API categories gathered from the original apps (CATEGORY), their IDs (ID), and examples (EXAMPLE).

We then inject privacy leaks into the apps based on the permissions they receive. Given an app P, with X and Y being the set of source and sink API categories that P is permitted to use, respectively, we produce a set $S_P = \{s_P^{\langle x,y\rangle} | x \in X \wedge y \in Y\}$ of subject apps with injected leaks, where each element corresponds to one feasible source-sink pattern $\langle x, y\rangle$ for P. Table II also lists for each original app the categories of source (SRC) and sink (SINK) APIs and the number of source-sink patterns (#P). Based on the 25 apps, we produce in total 88 subjects injected with leaks.

To construct each subject $s_P^{\langle x,y\rangle}$, we first randomly select (with probability 0.8) a subset of the entry points in P, then for each selected entry point e, we 1) randomly decide which of the three schemes given in Section I the injected leak should implement (the probability of scheme *(i)*, *(ii)*, and *(iii)* being chosen is 0.125, 0.125, and 0.75, respectively), and 2) randomly select a valid location l covered by a path from $\boldsymbol{P}_e$ for the injection. Note that each subject app constructed in this way contains multiple leaks. We conjecture the probabilities used in subject app construction would not affect the experimental results significantly, partly because the effectiveness of DL^2 depends mostly on its capability to steer a subject app to exercise the DCL involved in privacy leakage, as shown by the results presented in Section IV-D. We leave a proper investigation into the impact of such choices via systematic experiments for future work.

The actual injection of a privacy leak into P involves two more tasks: 1) injecting a snippet at location l to load an external class and invoke a method of the class at runtime; 2) placing the properly constructed external class file at the right location so that it can be successfully loaded by the code injected in task 1). Figure 2 shows the templates that DL^2 uses to generate the code implementing privacy leaks. The snippet to be injected is instantiated from the template shown in Figure 2.a, while the external classes are instantiated from the template shown in Figure 2.b. Among the placeholders used in the templates (i.e., strings surrounded by a pair of '`$`'s), one of the two source action placeholders is to be replaced with code retrieving sensitive information (in the form shown in Figure 2.c) and the other with an empty string. Similarly, one of the sink action placeholders is to be replaced with code leaking the sensitive information (in the form shown in Figure 2.d) and the other with an empty string. Placeholders used in templates from Figures 2.c and 2.d are to be replaced with actual calls to APIs from categories x and y, respectively. Column #LEAK$\backslash$INJ of Table II gives the number of privacy leaks injected into each app.

B. Comparative Techniques

To put the performance of DL^2 in perspective, we also apply two state-of-the-art privacy leakage detection tools to the same set of subject apps: TaintDroid [9] is a sophisticated dynamic taint-tracking and analysis system which can detect privacy leaks hidden by DCL; DyDroid [25] is a tool to detect malicious behaviors in dynamically loaded classes. Stadyna [32] is yet another tool for privacy leak detection, but it uses a manual approach to trigger DCL events. To avoid the bias introduced by manual inputs, we exclude Stadyna from the comparison.

C. Experimental Protocol

In the experiments, we apply each privacy leak detection technique to the 88 subject apps and record the numbers of leaks each technique detects. We also record the time DL^2 spends on each of the three steps described in Section III. To answer RQ3, we also keep track of the number of classes downloaded during external class capturing.

Both TaintDroid and DyDroid are driven by user events

[5]Gaming apps are often built on top of frameworks or engines, and specific techniques are required to effectively analyze them.

randomly generated by the Monkey[6] Android app exerciser. We configure Monkey to generate N_e random events for each tool and empirically set N_e to be 6000 to strike a good balance between effectiveness and efficiency. To properly account for the randomness intrinsic to these techniques, we run TaintDroid and DyDroid for 20 times [33] on each subject app and use their best performance in comparisons with DL^2.

D. Experimental Results

Table II also reports on the results of the experiments. In particular, the table lists for each original app the numbers of leaks detected (#LEAK) by DL^2 (DL2), TaintDroid (TD), and DyDroid (DD), and the average (mean) time DL^2 spent on path construction (PC), external code capturing (ECC), path-oriented taint analysis (PTA), and the whole privacy leak detection (TOT). Table III reports for each source-sink pattern the total number of hidden leaks injected (#LEAK$_{\mathrm{INJ}}$) and the number of leaks detected by each tool (#LEAK$_{\mathrm{DL2}}$, #LEAK$_{\mathrm{TD}}$, and #LEAK$_{\mathrm{DD}}$).

1) RQ1: Privacy Leak Detection: DL^2 detected in total 1073, or 42%, of the 2578 injected leaks, with the mean and median detection rates across subject apps being 50% and 52%, respectively, which suggests that DL^2 is overall effective in detecting privacy leaks hidden by DCL. There are two extreme cases among the apps. The detection rate of DL^2 was 100% on app A7, a barcode scanner application. A closer look at the app revealed that the app has a relatively simple GUI compared with other apps, and the injected DCLs were also easy to trigger. DL^2, however, failed to detect any leak injected in app A9, an open street map application. Since IntelliDroid failed to find any path leading to an intersection point, DL^2 terminated prematurely after path construction.

Compared with DL^2, TaintDroid and DyDroid detected only 408 and 349, or 16% and 14%, of the injected leaks, respectively. A main reason for the relatively low detection rate of TaintDroid is that it detects only leaks that are triggered on the paths it exercises. The most important reason for DyDroid's relatively low effectiveness is that it detects just privacy leaks completely implemented in external classes, but not cases where only the source or the sink of a leak is within the external code. In particular, DyDroid cannot detect the example leak shown in Section II.

Figure 3 plots the distribution of privacy leak detection rate on the subject apps by each of the three tools. It is clear from the figure that DL^2 detects more leaks than the other two. A two-tailed pair-wise Mann-Whitney U test [34] confirms the difference between DL^2 and TaintDroid is significant ($p = 0.002$, effect size $= 0.89$) and so is that between DL^2 and DyDroid ($p = 0.003$, effect size $= 0.87$)[7].

To better understand the limitations of the technique, we manually examined the privacy leaks that DL^2 failed to detect and found out that the main reason for such ineffectiveness is in our limited capability in steering the apps to load and execute the external classes. First, although IntelliDroid can often find a path from a given entry point to an intersection point and construct the constraint corresponding to the path, the Z3 solver may not be able to solve the constraint. Second, even when the path constraint can be solved successfully, DL^2 may not be able to realize an execution using the generated values. For instance, when calls to Android platform APIs are involved along a path, specific mechanisms like mocking need to be installed to make sure the calls return the expected values at runtime, but DL^2 does not support such mechanisms yet. In our experiments, these two reasons caused 776 and 726 injected privacy leaks to be missed by DL^2, respectively.

DL^2 is able to detect 42% of the injected privacy leaks in the benchmark, which is 163% and 200% more than the state-of-the-art privacy leak detection tools TaintDroid and DyDroid, respectively.

2) RQ2: Efficiency: Overall, DL^2 takes an average of 581.5 seconds to process one subject app, including 243.6 seconds for path construction, 107.6 seconds for external code capturing, and 230.3 seconds for path-oriented taint analysis.

Each subject app used in the experiments is injected with

[6]http://developer.android.com/guide/developing/tools/monkey.html

[7]In this work, the effect size is calculated as the Vargha and Delaney's $\hat{A}_{12}$ statistic [33].

```
try{
  Object info = null;
  $source-action1$
  Class cls = loadClass("tool.dl2.ExternalClass");
  Object instance = cls.newInstance();
  cls.getMethod("externalMethod", cls)
     .invoke(instance, info);
  $sink-action1$
}catch(Exception e){ ... }
```

a) Template for the snippet to be injected.

```
package tool.dl2;
class ExternalClass{
  public void externalMethod(Object info){
    $source-action2$
    $sink-action2$
  }
}
```

b) Template for the external class.

```
Object info = $callSourceMethod$
```

c) Template for sourcing sensitive information.

```
$callSinkMethod(info)$;
```

d) Template for sinking sensitive information.

Fig. 2: Code templates used in privacy leak injection.

TABLE II: Subject apps and experimental results.

APP	ID	PACKAGE	VER	SIZE				PATTERN			#LEAK				T			
				#C	#M	#EP	LOC	SRC	SINK	#P	INJ	DL2	TD	DD	PC	ECC	PTA	TOT
Lightning	A1	acr.browser.lightning	4.5.1	400	1392	6	21110	loc,brw	net,fil	4	20	18	19	15	34.2	13.3	60.1	107.6
OpenKeychain	A2	org.sufficientlysecure.keychain	5.1.4	1233	5569	67	85816	acc,con	net,fil	4	252	111	47	40	194.1	81.7	508	783.8
AnkiDroid	A3	com.ichi2.anki	2.8.4	538	3075	33	66667	aud	net,fil	2	43	31	18	16	103.3	66.2	302.8	472.3
Wikipedia	A4	org.wikipedia	r/2.7.239	1494	6174	44	66655	loc,acc	net,fil	4	111	28	17	15	137.6	18.6	61	217.2
RingDroid	A5	com.ringdroid	2.7.3	74	268	3	6689	con,aud	net,fil	4	12	8	6	6	28.8	21	42.6	92.4
Afreerdp	A6	com.freerdp.afreerdp	2.0.0-rc1	95	696	12	11220	aud	net,fil	2	19	11	6	4	37.2	33.1	85.5	155.8
Barcode Scanner	A7	com.google.zxing.client.android	4.7.7	342	1789	9	40917	con,brw	net,fil	4	28	28	21	17	27.6	23.2	90.4	141.2
Materialistic	A8	io.github.hidroh.materialistic	3.2	475	1906	31	20871	acc	net,fil	2	46	33	22	17	68.1	53.3	285.4	406.8
Osmdroid	A9	org.osmdroid	6.0.1	581	3039	16	48863	loc	net,fil	2	26	0	13	11	78	0	0	78
Silence	A10	org.smssecure.smssecure	0.16.9	772	4034	50	53441	con,phn,sms	net,fil	6	124	52	13	12	103.4	111.5	131.9	346.8
Tasks	A11	org.dmfs.tasks	1.1.13	726	2960	17	43957	acc,con	fil	2	25	19	15	12	48.2	43.8	199.5	291.5
MPDroid	A12	com.namelessdev.mpdroid	1.07.2	198	1112	19	18691	phn	net,fil	2	25	15	7	7	86.1	25.7	111	222.8
Letgo	A13	com.abtnprojects.ambatana	2.1.5	3916	14021	62	159325	loc,acc	net,fil	4	218	138	27	22	657.3	298.5	548	1503.8
Bitmoji	A14	com.bitstrips.imoji	10.11.279	531	2154	36	24409	con,phn,aud	net,fil	6	161	79	26	23	475.3	152.8	330.8	958.9
Message lite	A15	com.facebook.mlite	29.0.0.6.188	2213	7509	47	148916	phn,aud,sms	net,fil	6	94	13	11	9	72.3	11.1	25.7	109.1
iTranslate	A16	at.nk.tools.iTranslate	4.0.3	43	217	37	4635	acc,aud	net,fil	4	84	50	22	18	783.5	297.4	297.1	1378
Xender	A17	cn.xender	3.9.0612	1693	9822	60	155739	phn,sms	net,fil	4	88	66	25	22	106.1	117.1	226.7	449.9
Wish	A18	com.contextlogic.wish	4.8.0	3637	11536	56	135163	acc,con	net,fil	4	106	29	12	8	466.1	199.7	185.3	851.1
Remind	A19	com.remind101	7.7.1.15565	632	4292	78	48976	con,aud	net,fil	4	195	27	7	7	320.4	123.2	230.4	674
PayPal	A20	com.paypal.android.p2pmobile	6.12.1	1611	7615	146	68822	loc	net,fil	2	278	9	12	10	539.7	27.5	45.7	612.9
TextNow	A21	com.enflick.android.TextNow	5.14.1	806	3327	80	63769	phn,sms	net,fil	4	156	74	4	4	116.8	73.1	302.1	492
Lyft	A22	me.lyft.android	4.37.3	371	2231	18	26360	phn	net,fil	2	27	14	2	2	283.5	108.2	209.7	601.4
News Break	A23	com.particlenews.newsbreak	3.1.4	13	89	72	1136	loc,acc	net,fil	4	250	139	40	37	396.1	377.1	632.4	1405.6
Pinterest	A24	com.pinterest	6.22.0	4497	16471	31	253252	loc,con	net,fil	4	65	20	7	7	443	77.6	147.8	668.4
Venmo	A25	com.venmo	7.1.0	1296	4982	75	44228	loc	net,fil	2	125	61	9	8	98.5	242.4	564.3	905.2
Overall	–	–	–	28187	116280	1105	1619627	–	–	88	2578	1073	408	349	243.6	107.6	230.3	581.5

multiple privacy leaks. The time for DL^2 to detect one privacy leak averages to 18.8 seconds, suggesting the efficiency of DL^2 is comparable with most existing privacy leakage detection tools.

Figure 4 plots the distribution of detection time (in seconds) of DL^2 both across apps (4a) and across leaks (4b). According to the figure, the detection time is less than 20 seconds for around 75% of the leaks.

On average, it takes DL^2 581.5 seconds to process one subject app and 18.8 seconds to detect one privacy leak hidden by DCL.

3) RQ3: DCL Triggering: To evaluate whether DL^2 can trigger DCL effectively, we measure the average number of DCLs triggered by DL^2 and the other two tools across the subject apps. Since both TaintDroid and DyDroid use input events generated by Monkey to exercise the subject apps, we do not differentiate the two tools in answering this research question. In Table IV, the first row of data is for DL^2, and the remaining rows are for both TaintDroid and DyDroid. For TaintDroid and DyDroid, we report the average numbers

TABLE III: Privacy leak detection results by source-sink pattern.

SOURCE-SINK PATTERN	$\#LEAK_{INJ}$	$\#LEAK_{DL2}$	$\#LEAK_{TD}$	$\#LEAK_{DD}$
loc → fil	388	118	45	40
loc → net	374	119	45	39
con → fil	235	96	43	37
con → net	228	80	28	25
acc → fil	227	110	51	42
acc → net	214	106	43	33
phn → fil	285	149	41	35
phn → net	195	145	41	39
brw → fil	13	12	10	7
brw → net	12	12	10	8
aud → fil	152	58	28	23
aud → net	148	61	22	20
sms → fil	4	3	0	0
sms → net	3	3	0	0
Overall	2578	1073	408	349

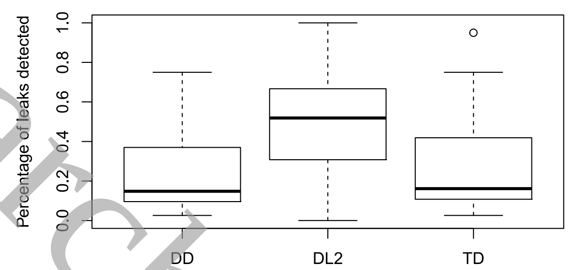
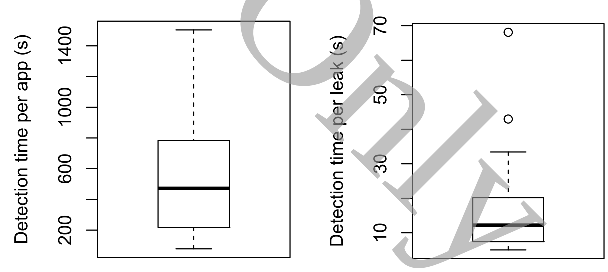


Fig. 3: Numbers of privacy leaks detected.



(a) Across apps. (b) Across leaks.

Fig. 4: Distribution of privacy leak detection time.

TABLE IV: Average Numbers of DCL Triggered.

TOOL	#EVENT	#DCL	T(SEC.)
DL^2	34.8	12.1	351.2
TaintDroid and DyDroid	2000	3.0	11.1
	4000	3.5	21.5
	6000	3.9	28.9

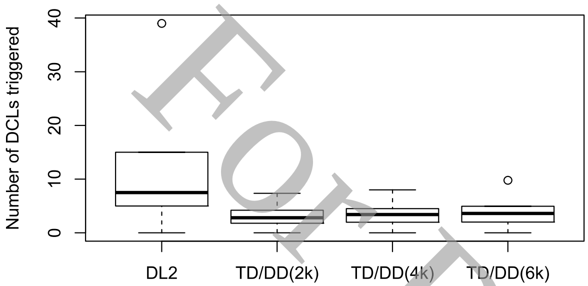


Fig. 5: Distribution of numbers of DCLs triggered on subject apps.

of DCL triggered and the average time cost when using 2000, 4000, and 6000 input events in the dynamic analysis, respectively. In each row, the table lists the average number of input events generated to exercise an app (#EVENT) and the average number of DCL triggered using the events (#DCL). Box plots in Figure 5 shows the distribution of numbers of DCL triggered in each subject app in this experiment.

DL^2 can effectively trigger DCLs. On average, 2.87 input events are needed for DL^2 to trigger one DCL during external code capturing. Compared with that, the average numbers of input events needed to trigger one DCL ranges between 666 and 1538 with TaintDroid and DyDroid, which is considerably more than what DL^2 needes. Moreover, the efficiency of TaintDroid and DyDroid decreases when more input events are used in experiments, as suggested by Figure 5, partly because the longer Monkey runs, the more duplicated events it tends to generate. In the experiments, DL^2 collected in total 175 unique path conditions, and it successfully solved 95 of them.

One reason for the big differences between the tools in triggering DCL is that most component life-cycle methods and event handlers in our subject apps are only executed after the user has successfully logged in, which however is nearly impossible for TaintDroid and DyDroid to achieve since all the input events they utilize are randomly generated by Monkey. DL^2 circumvents this requirement for valid credentials by directly triggering events on the corresponding components. For instance, to execute method `SmsReceiver.onReceive` as shown in Listing 1, DL^2 employs the ADB tool to directly trigger an `onReceive` event on an `SmsReceiver` object. Such technique has its own limitations, as discussed in Section IV-D1, but it is effective in activating simple event handlers and helps DL^2 successfully execute a significant amount of entry methods in our experiments.

On average, 2.87 input events are needed for DL^2 to trigger one DCL. Compared with that, the average numbers of input events needed to trigger one DCL ranges between 666 and 1538 with TaintDroid and DyDroid.

E. Threats to Validity

In this section, we outline possible threats to the validity of our findings.

Construct validity: Threats to construct validity are mainly concerned with whether the measurements used in the experiment reflect real-world situations. The effectiveness of privacy leak detection is often measured in terms of the number of leaks detected. However, since our focus is on privacy leakage in Android apps hidden by DCL, we measure and compare the effectiveness of privacy leakage detection techniques in numbers of hidden leaks detected.

Internal validity: Threats to internal validity are mainly concerned with the uncontrolled factors that may have also contributed to the experimental results.

The main threat to internal validity is in the possible faults in the implementation of our approach. To mitigate the threat, we review our code and experimental scripts to ensure their correctness before conducting the experiments.

External validity: Threats to external validity are mainly concerned with whether the findings in our experiment are generalizable for other situations.

Android apps and injected privacy leaks used as subjects in the experiments may pose threats to the external validity. First, the 25 apps selected from the Google Play store and F-Droid app market may not be representative of other Android apps. Second, since the focus of this work is to detect privacy leaks hidden by DCL, all the leaks injected into the apps implement one of schemes *(i)–(iii)* described in Section I, which introduces a major threat to the external validity of our findings in the experiments. Besides, the hidden privacy leaks injected into the apps all have relatively short propagation paths for tainted data. Actual privacy leaks in real-world apps, implemented using DCL or not, however may be more complex. To mitigate the threats, we plan to carry out more extensive experiments to evaluate DL^2 using leaks from real world apps in the future.

V. Related Work

DL^2 achieves effective privacy leakage detection by combining static and dynamic taint analysis, whose most important contributions we briefly review below. Since this work aims to address the challenges introduced by dynamic class loading to privacy leakage detection, we also discuss works related to dynamic class loading in software security.

A. Static Taint Analysis

Static analysis has been widely used in a number of research to detect privacy leakage in Android apps. Long et al. propose the CHEX [7] technique to detect component hijacking vulnerability in Android apps. The technique transforms apps in bytecode into a CHEX intermediate representation and applies the WALA framework to implement a data flow analysis that can also be used to detect privacy leakage. However, CHEX requires as input a complete model of the framework and is limit to 1-object-sensitivity. The AndroidLeaks [3] technique developed by Clint et al. is among the first few works on privacy leakage detection for Android apps. AndroidLeaks utilizes the WALA analysis framework to construct a context sensitive system dependence graph (SDG) and uses the SDG to inform a static taint analysis. Since the technique tracks data flow at the object-level, the precision of the analysis is limited. Yang et al. develop the LeakMiner [18] technique that employs the Soot [30] bytecode manipulation and optimization framework and and the points-to analysis framework to generate call-graphs and detect privacy leaks.

Bodden et al. present FlowDroid [4], which is one of the most sophisticated static analysis tools for Android. FlowDroid builds a precise and complete model of Android's lifecycle callbacks and leverages Soot [30] to build call-graphs, similar to LeakMiner. Then, it conducts a forward taint analysis and an on-demand backward alias analysis using IFDS [35] on Android apps to detect privacy leaks. The analyses are context, flow, field and object-sensitive, but it does not consider inter-component communication in the apps, which may also affect the data flow. In view of that, approaches like Amandroid [8], R-Droid [19], and IccTA [6] were proposed to detect leaks involving inter-component communication. Cao et al. propose EdgeMiner [5], a technique that performs backward data-flow analysis over the Android source code to process implicit control dependence introduced by the callback mechanism.

B. Dynamic Taint Analysis

Static analysis often reports a large number of false positives, while dynamic analysis does not suffer from the issue. The TaintDroid tool [9], developed by Enck et al., is one of the most widely used Android dynamic taint-tracking and analysis system. TaintDroid is based on a modified Android system and can be used to collect taint information of sensitive data at runtime. The information can be analyzed later by users to discover leakage of privacy data. Many dynamic analysis tools for Android are built on top of TaintDroid: DroidBox [20] uses TaintDroid to build a sandbox for analyzing apps; AppFence [11] extends TaintDroid to not only detect privacy leaks in apps, but also prevent sensitive data leaks by adding a module for privacy access control to apps BayesDroid [21] is similar to TaintDroid, but it implements a Bayesian classification to improve the accuracy of analysis results. DroidScope [10] and CopperDroid [12] provide full system analysis and collect dynamic information for reconstructing malicious behaviors. VetDroid [22] is another dynamic taint analysis system that vets undesirable behaviors in Android apps by systematically analyze the usage of permissions granted to apps. TaintART [23] implements a tracking system for the latest Android runtime which TaintDroid does not support.

Although dynamic analysis techniques can produce more precise detection results than static ones, they may miss leaks that are not triggered during the analysis. Therefore, work has been done to combine the strengths of both static and dynamic analyses, e.g., by generating analysis results statically and verifying them dynamically. SmartDroid [24] takes advantage of a hybrid approach that statically generates paths leading to suspicious actions and dynamically executes the UI events which can flow through the paths. AppIntent [13] distinguishes user-intended and unintended data transmission and employs dynamic symbolic execution to determine whether a behavior is malicious. AppAudit [14] detects vulnerabilities statically and verifies the data leaks through dynamic analysis.

DL^2 combines static and dynamic taint analysis to effectively detect privacy leakage hidden by DCL. Taint analysis implemented in DL^2, however, is rather lightweight. For example, it is not context- or field-sensitive. In consequence, the analysis results are not precise enough in certain circumstances and may contain false positives. Besides, implicit flows [36] in apps are not considered in DL^2's taint analysis.

C. Dynamic Code Loading

DCL poses new challenges to various security analyses on mobile apps. Peoplau et al. conducted a large-scale study on the vulnerabilities in mobile apps due to DCL and summarized the findings based on apps collected from the Google Play store into a group of common patterns [17]. They also presented static analysis techniques to detect vulnerabilities based on the patterns, and modified the Android Dalvik virtual machine to prevent attacks due to DCL based on whitelists. Their techniques, however, do not analyze behaviors in the external code and thus cannot effectively detect privacy leaks that DL^2 targets at. Stadyna [32] executes DCL dynamically and uses static analysis to expand the method call graph by analyzing the loaded classes. Through the expanded call graph, it can identify suspicious behaviors more precisely using a permission map. DyDroid [25] also leverages static and dynamic analysis to detect DCL. It triggers DCL through fuzzy testing and intercepts dynamically loaded classes. Then it analyzes the loaded code to detect malicious behaviors or privacy leakage statically. Falsina et al. propose the Grab'n run system, which includes a code verification protocol and a serious of supporting libraries, APIs, and components [37], to address security issues related to the misuses of DCL. Systems like IntelliDroid [27] can be used to construct and realize paths between specified locations in apps. They serve more general purposes, and can be customized and extended, e.g., to conduct specific analysis on app behaviors involving DCL, as exemplified by DL^2.

VI. Conclusions

Privacy leakage detection for Android apps has always been an important task in the area of software security. In view that existing techniques offer only limited effectiveness in detecting leaks hidden by DCL, we propose in this paper DL^2 that enhances static analysis with dynamic app execution to effectively detect such leaks. We have implemented the technique into a tool with the same name. Experimental evaluation of the tool on 88 subjects apps injected with 2578 privacy leaks shows that DL^2 is both effective and efficient in detecting leaks implemented hidden by DCL.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation (Grant Nos. 61690204, 61472180 and 61502228) of China, and partly supported by the Hong Kong RGC General Research Fund (GRF) PolyU 152703/16E and The Hong Kong Polytechnic University internal fund 1-ZVJ1 and G-YBXU.

REFERENCES

[1] I. D. Corporation, "Worldwide smartphone os market share," https://www.idc.com/promo/smartphone-market-share/os.

[2] M. Zhang and H. Yin, "Efficient, context-aware privacy leakage confinement for android applications without firmware modding," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 259–270.

[3] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 291–307.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. Mcdaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM Sigplan Conference on Programming Language Design and Implementation*, 2014, pp. 259–269.

[5] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework." in *NDSS*, 2015.

[6] 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 *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.

[7] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 229–240.

[8] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[9] 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," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[10] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *USENIX security symposium*, 2012, pp. 569–584.

[11] 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*. ACM, 2011, pp. 639–652.

[12] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors." in *NDSS*, 2015.

[13] 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 SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1043–1054.

[14] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 899–914.

[15] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2017, pp. 1–16.

[16] Google, "Android security tips," http://developer.android.com/training/articles/security-tips.

[17] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications." in *NDSS*, vol. 14, 2014, pp. 23–26.

[18] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Software Engineering (WCSE), 2012 Third World Congress on*. IEEE, 2012, pp. 101–104.

[19] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer, "R-droid: Leveraging android app analysis with static slice optimization," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 129–140.

[20] A. Desnos and P. Lantz, "Droidbox: An android application sandbox for dynamic analysis (2011)," https://code.google.com/p/droidbox, 2014.

[21] O. Tripp and J. Rubin, "A bayesian approach to privacy enforcement in smartphones." in *USENIX Security Symposium*, 2014, pp. 175–190.

[22] 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 SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 611–622.

[23] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 331–342.

[24] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 93–104.

[25] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley, "Dydroid: Measuring dynamic code loading and its security implications in android applications," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 2017, pp. 415–426.

[26] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving android malware and dynamic loading technique." *IEEE Trans. Information Forensics and Security*, vol. 12, no. 7, pp. 1529–1544, 2017.

[27] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware." in *NDSS*, vol. 16, 2016, pp. 21–24.

[28] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the "small scope hypothesis"," 10 2002.

[29] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[30] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[31] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[32] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 37–48.

[33] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification, and Reliability*, vol. 24, no. 3, pp. 219–250, May 2014.

[34] G. Corder and D. Foreman, *Nonparametric Statistics: A Step-by-Step Approach*. Wiley, 2014.

[35] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[36] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *International Conference on Information Systems Security*. Springer, 2008, pp. 56–70.

[37] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi, "Grab'n run: Secure and practical dynamic code loading for android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 201–210.