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

Yufei Yang1, Wenbo Luo1, Yu Pei1, 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

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 DL² technique that enhances static analysis with dynamic app execution to effectively detect privacy leakage implemented via DCL in Android apps. To evaluate DL², we construct a benchmark of 88 subject apps with 2578 injected privacy leaks and apply DL² to the apps. DL² 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 DL² technique that enhances static analysis with dynamic app execution. Given an Android app, DL² first applies static analysis to find paths in the app that lead to invocations to methods from external classes and
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 antimalware 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.

```
Listing 1: The code example of DCL

DexClassLoader loader = DexClassLoader(...);
... 
public class SmsReceiver extends BroadcastReceiver{
  ...
  public void onReceive(...){
    ...
    String clsName = ..., methodName = ...
    SmsMessage msg = SmsMessage.createFromPdu(pdus[x]);
    String phoneNbr = msg.getOriginatingAddress();
    if(msg.getMessageBody().contains(KEYWORD)){
      try{
        Class cls = (Class)loader.loadClass(clsName);
        Object instance = cls.newInstance();
        cls.getMethod(methodName, Object.class)
          .invoke(instance, phoneNbr);
      }
    }
    ...
  }
}
```

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.getMethodOriginatingAddress() is a source API, the method actually invoked on Line 14 is SmsSender.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

```
Listing 2: The method invoked by DCL

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

1https://f-droid.org/en/
2https://virusshare.com/, MD5: 4217d6663656239a53d70e5e7e174adb.
analysis of the code in Listing 1 can label variable phoneNumber 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,

\(^3\)A component of an Android app is either an activity, a service, a broadcast receiver, or a content provider.

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 \(E_P\) 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 \(e \in E_P\) 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 \(N_f\), 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 \(e \in E_P\), a set \(P_e\) of paths that start from \(e\) and end at an intersection point. Let \(P = \bigcup_{e \in E_P} P_e\).

B. External Code Capturing

In this step, DL\(^2\) employs dynamic analysis to capture the external code executed along each path from \(P\). 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 \(p \in P_e\), 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 IntellijDroid [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

Fig. 1: An overview of DL\(^2\).
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, DL² 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, DL² also records the trace of the actual execution of \( m_p \); 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, DL² discards both the trace and \( p \).

At the end of this step, DL² has collected a set \( \mathbb{P} (P \subseteq P) \) of paths. Each path \( p \in \mathbb{P} \) starts from an entry point \( e_p \) and ends at an intersection point \( e_p \), where method \( M_p \) from the external class \( C_p \) is invoked through reflection.

### C. Path-Oriented Taint Analysis

In this step, DL² 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, DL² 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.

DL² 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 Implementation Details

We have implemented the DL² technique into a tool, also named DL². DL² 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, DL² leverages the open source ApkTool [4]. During path construction (Section III-A), DL² flattens the enclosing method of every entry point by inlining methods directly or indirectly invoked within \( N_f = 3 \) 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, DL² 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 DL² 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 DL² in detecting privacy leaks hidden by DCL?
- **RQ2**: How efficient is DL²?
- **RQ3**: How effective is DL² in triggering DCL?

In the experiments, Android apps were executed on a Google Nexus emulator running Android 4.3. Both DL² 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 DL² 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

---

TABLE I: Potential source and sink API categories.

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>ID</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>loc</td>
<td>LocationManager.getLastKnownLocation()</td>
</tr>
<tr>
<td>account</td>
<td>acc</td>
<td>AccountManager.getAccount()</td>
</tr>
<tr>
<td>contact</td>
<td>con</td>
<td>ContentResolver.query(...)</td>
</tr>
<tr>
<td>phone</td>
<td>phn</td>
<td>TelephonyManager.getDeviceId()</td>
</tr>
<tr>
<td>browser</td>
<td>bw</td>
<td>ContentResolver.query(...)</td>
</tr>
<tr>
<td>audio</td>
<td>aud</td>
<td>ContentResolver.query(...)</td>
</tr>
<tr>
<td>sms</td>
<td>sms</td>
<td>SmsMessage.getMessageBody()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sink</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>internet</td>
<td>net</td>
</tr>
<tr>
<td>file</td>
<td>fil</td>
</tr>
</tbody>
</table>

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

By dynamically modifying apps at runtime, 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 (i) cannot run on our experiment emulator, (ii) the probability of scheme (i), (ii), and (iii) being chosen is 0.125, 0.125, and 0.75, respectively, and (ii) randomly select a valid location \( l \) covered by a path from \( P_c \) for the injection. Note that each subject app constructed in this way contains multiple leaks. We conjecture that 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 ‘\( \ast \)’), 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 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
randomly generated by the Monkey 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 (#LEAKHID) and the number of leaks detected by each tool (#LEAKDL2, #LEAKTD, and #LEAKDD).

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

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

a) Template for the snippet to be injected.

\[
\text{Object info = $callSourceMethod$} \\
\]

c) Template for sourcing sensitive information.

\[
\text{$callSinkMethod(info)$;} \\
\]

d) Template for sinking sensitive information.

\[
\text{Fig. 2: Code templates used in privacy leak injection.}
\]

\[
^7\text{In this work, the effect size is calculated as the Vargha and Delaney’s \(\hat{A}_{12}\) statistic [33].}
\]
multiple privacy leaks. The time for DL\(^2\) to detect one privacy leak averages to 18.8 seconds, suggesting the efficiency of DL 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

![Fig. 3: Numbers of privacy leaks detected.](image-url)

![Fig. 4: Distribution of privacy leak detection time.](image-url)
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.

DL² can effectively trigger DCLs. On average, 2.87 input events are needed for DL² 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 DL² needs. 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, DL² 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. DL² 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, DL² 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 DL² successfully execute a significant amount of entry methods in our experiments.

### V. RELATED WORK

DL² 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 limited 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.

Boddien 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 explicit control dependence introduced by the callback mechanism.

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 DCL 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 DL2.

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 DL2 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 DL2 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