

Ripple: Reflection Analysis for Android Apps in Incomplete Information Environments

Yifei Zhang, Tian Tan, Yue Li and Jingling Xue
School of Computer Science and Engineering, UNSW Australia

ABSTRACT

Despite its widespread use in Android apps, reflection poses graving problems for static security analysis. Currently, string inference is applied to handle reflection, resulting in significantly missed security vulnerabilities. In this paper, we bring forward the ubiquity of incomplete information environments (IIEs) for Android apps, where some critical data-flows are missing during static analysis, and the need for resolving reflective calls under IIEs. We present RIPPLe, the first IIE-aware static reflection analysis for Android apps that resolves reflective calls more soundly than string inference. Validation with 17 popular Android apps from Google Play demonstrates the effectiveness of RIPPLe in discovering reflective targets with a low false positive rate. As a result, RIPPLe enables FlowDroid to find hundreds of sensitive data leakages that would otherwise be missed.

Keywords

Android; Reflection Analysis; Pointer Analysis

1. INTRODUCTION

Static analysis is a fundamental tool for detecting security threats in Android apps. However, reflection poses graving problems for static analysis. According to a recent study [19], malware authors can use reflection to hide malicious behaviors from detection by all the 10 commercial static analyzers tested. Similarly, academic static analyzers either ignore reflection [7, 17] or handle it partially [3], resulting in also significantly missed program behaviors.

In Android apps, reflection serves a number of purposes, including (1) plug-in and external library support, (2) hidden API method invocation, (3) access to private API methods and fields, and (4) backward compatibility. Indeed, reflection is widely used in both benign and malicious Android apps. For a sample of 202 top-chart free apps from Google Play that we analyzed on 15 April 2016, we found that 92.6% of these apps use reflection. Elsewhere, in a malware sam-

ple consisting of 6,141 Android apps from the VriusShare project [1], 48.13% of them also use reflection.

Reflection can introduce implicitly many caller-callee edges into the call graph of the program. If some targets at a reflective call are ignored, their corresponding caller-callee edges will not be discovered. As a result, possible security vulnerabilities in the invisible part of the program, such as Obad [26] and FakeInstaller [20], may go undetected.

Therefore, the objective of reflection analysis is to discover the targets at reflective calls (e.g., objects created, methods called and fields accessed). For Android apps, regular string inference is currently performed to discover the string constants used as class/method/field names at reflective calls [4, 6, 9]. This is inadequate for framework-based and event-driven Android apps. In practice, reflection analysis is usually performed together with many other analyses, including pointer analysis [8, 10, 11, 13, 22], inter-component communication (ICC) analysis [16, 17], callback analysis [3, 8, 27], and library summary generation [2, 5, 8]. Soundness, which demands over-approximation, is often sacrificed in order to achieve efficiency and precision tradeoffs. As a result, class/method/field names used at reflective calls may be non-constant (either non-null but statically unknown or simply null). Similarly, the receiver objects at reflectively method call sites (i.e., the objects pointed to by `v` in `Method.invoke(v, ...)`) may be non-null with statically unknown types or simply null. All these can happen due to, for example, unsound library summaries, unmodeled Android services, code obfuscation, and unsound handling of hard-to-analyzed Android features such as ICC, callbacks and built-in containers. In this case, regular string inference, which keeps track of only constant strings, is ineffective, resulting in missed program behaviors.

In this paper, we bring forward the ubiquity of incomplete information environments (IIEs) for Android apps, where some critical data-flows are inevitably missing during static analysis. As discussed above, these include not only the case when class/method/field names are non-null but statically unknown, which is studied previously for Java programs [10, 11, 13, 22], but also the case when these string names are null, which is investigated for the first time for Android apps in this paper. We therefore emphasize the need for resolving reflective calls in Android apps under IIEs. To this end, we introduce RIPPLe, the first IIE-aware static reflection analysis for Android apps that can resolve reflective calls more soundly than string inference at a low false positive rate. We also demonstrate its effectiveness in improving the precision of an important security analysis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA

© 2017 ACM. ISBN 978-1-4503-4523-1/17/03...\$15.00

DOI: <http://dx.doi.org/10.1145/3029806.3029814>

For an extended version of this paper, we refer to [28]. In summary, this work makes the following contributions:

- We present (for the first time) an empirical study for IIEs in real-world Android apps, and examine some common sources of incomplete information, discuss their impact on reflection analysis, and motivate the need for developing an IIE-aware reflection analysis.
- We introduce RIPPLE, the first IIE-aware reflection analysis for Android apps, which performs type inference automatically (without requiring user annotations) and thus subsumes regular string inference.
- We have implemented RIPPLE in SOOT with RIPPLE working together with its pointer analysis, SPARK. We evaluate the soundness, precision, scalability and effectiveness of RIPPLE by using 17 popular real-world Android apps from Google Play. RIPPLE discovers 72 more (true) reflective targets than string inference in seconds at a low false positive rate of 21.9%. This translates into a total of 310 more sensitive data leakages detected by FLOWDROID [3].

2. IIES IN ANDROID APPS

There are two types of missing information under IIEs. In one case, the data flows needed for resolving reflective calls exist but are statically unknown. For example, class/method/field names used at reflective calls are non-null but unknown. We can resolve such reflective calls by performing type inference, as done for Java programs [10, 11, 22].

In the other case, the data flows needed for resolving reflective calls are completely missing, indicated by the presence of null. To understand this case, which has never been studied before, for Android apps, we have performed an empirical study on 45 Android apps, with 20 popular Android apps from Google Play and 25 malware samples from the VirusShare Project [1]. We discuss the four most common sources of incomplete information in IIEs: (1) undetermined intents, (2) behavior-unknown libraries, (3) unresolved built-in containers, and (4) unmodeled services. We delve into their bytecode to explain why regular string inference is inadequate, since it fails to enable FLOWDROID [3] to discover many data leaks from *sources* (API calls that inject sensitive information) to *sinks* (API calls that leak information). We also provide insights on why our IIE-aware RIPPLE can handle IIEs more effectively than string inference.

2.1 Undetermined Intents

ICC via intents is one of the most fundamental features in Android as it enables some components to process the data originating from other components. Thus, the components in an Android app function as building blocks for the entire system, enhancing intra- and inter-application code reuse.

In practice, some inter-component control- and data-flows cannot be captured by ICC analysis [16, 17]. If a data flow from an intent into a reflective call is missing, the reflection call cannot be fully resolved. The code snippet in Figure 1 taken from the game *Angry Birds* illustrates this problem.

In this app, the class name `cName` is obtained from an intent (line 4) and then used in a call to `Class.forName()` (line 5) to create a class metaobject `clz`. Then, an object of this class is created reflectively (line 6) and assigned to `mmBase` after a downcast to `MMBaseActivity` is performed. Finally, `onCreate()` is invoked on this object (line 7).

```
Android App Name: Angry Birds
1 public class MMAActivity extends Activity {
2   protected void onCreate(Bundle savedInstanceState) {
3     Intent intent = getIntent();
4     String cName = intent.getStringExtra("class");
5     Class clz = Class.forName(cName);
6     MMBaseActivity mmBase = (MMBaseActivity)clz.newInstance();
7     mmBase.onCreate(savedInstanceState);
    ... } }
```

Figure 1: Undetermined intents. Here, \dashrightarrow denotes a missed data-flow and \rightarrow the post-dominating-cast-based type inference used in Ripple.

To discover what `cName` is, we applied IC3, a state-of-the-art ICC analysis [17], but to no avail. Thus, the data-flow for `cName`, denoted by \dashrightarrow , is missing, rendering `clz` to be a null pointer. In this case, string inference is ineffective. As a result, the reflectively allocated object in line 6 and the subsequent call on this object in line 7 are ignored.

RIPPLE is aware of the incomplete information caused by this undetermined intent, which manifests itself in the form of `cName = null`. By taking advantage of the post-dominant cast `MMBaseActivity` for `clz.newInstance()` in line 6, RIPPLE infers that `mmBase` may point to five objects with their types ranging over `MMBaseActivity` and its four subtypes, which are all confirmed to be possible by manual code inspection. As a result, RIPPLE discovers 3,928 caller-callee edges in lines 5 – 7 (directly or indirectly), thereby enabling FLOWDROID [3] to detect 49 new sensitive data leaks that will be all missed by string inference in this part of the app that has been made analyzable by RIPPLE.

2.2 Behavior-Unknown Libraries

To accelerate the analysis of an application, the side effects of a library on the application are often summarized. Library summaries are either written manually [8] or generated automatically [2, 5]. However, both approaches are error-prone and often fail to model all the side-effects of a library for all possible analyses. DroidSafe [8] provides the Android Device Implementation (ADI) to model the Android API and runtime manually, with about 1.3 MLOC for Android 4.4.3. However, as the Android framework evolves with both new features and undocumented code added, how to keep this ADI in sync can be a daunting task.

Therefore, unsound library summaries are an important source of incomplete information in IIEs. The code snippet in Figure 2 taken from the app *Twist* illustrates this issue.

```
Android App Name: Twist
1 private static void writeToLog(UnityAdsDeviceLogEntry entry) {
2   String mName = entry.getLogLevel().getReceivingMethodName();
3   Method logMtd = Log.class.getMethod(mName, String.class, String.class);
4   String tag = ...;
5   String msg = ...;
6   logMtd.invoke(null, tag, msg);
7   public class Log {
8     public static int i(String tag, String msg) {}
9     public static int d(String tag, String msg) {} Sink
10    public static int w(String tag, String msg) {} Calls
11    public static int e(String tag, String msg) {}
12    public static int v(String tag, String msg) {}
13    public static int wtf(String tag, String msg) {}
    ... }
```

Figure 2: Behavior-unknown libraries. \dashrightarrow marks the methods invoked at a reflective call, \rightarrow denotes sensitive data-flow, and \bullet denotes tainted data.

This code snippet is used to log messages at different verbosity levels. In line 2, a method name `mName` is retrieved.

In line 3, its method metaobject `logMtd` is created. In line 6, this method, which is static, is invoked reflectively.

If we apply FLOWDROID [3] to detect data leaks in this app, by relying on string inference to perform reflection analysis, then the reflective call `logMtd.invoke()` in line 6 will be ignored. In FLOWDROID, the behaviors of maps are not summarized. However, `entry` was retrieved from a `HashMap` and then passed to `writeToLog`. Thus, `mName = null`, rendering string inference to be ineffective.

RIPPLE is aware of unsound library summaries and thus attempts to infer the target methods at `logMtd.invoke()`. Based on the facts that (1) these methods are static (since the receiver object is null), declared in or inherited by class `android.util.Log`, (2) each target method has two formal parameters, and (3) each parameter has a type that is either `String` or its supertype or its subtype, RIPPLE concludes that the six target methods, `i()`, `d()`, `w()`, `e()`, `v()` and `wtf()`, as shown in class `android.util.Log` may be potentially invoked. According to FLOWDROID, these six methods are all sinks for sensitive data contained in `msg`. Thus, resolving `logMtd.invoke()` causes 12 data leaks from two different sensitive data sources to be reported (as $2 \text{ sources} \times 6 \text{ sinks} = 12 \text{ leaks}$). By manual code inspection, we found that the first four methods, `i()`, `d()`, `w()` and `e()`, shaded in class `android.util.Log` are true targets, implying that 8 data leaks will not be reported if string inference is used.

2.3 Unresolved Built-in Containers

Android apps can receive a variety of user inputs from, e.g., intents, databases, internet, GUI actions, and system events. These data are stored in different types of containers, such as `Bundle`, `SharedPreferences`, `ContentValues` and `JSONObject`, for different purposes. Unhandled user inputs represent an important source of incomplete information in IIEs. The code snippet in Figure 3 taken from a game named *Seven Knights* illustrates this problem.

```
Android App Name: Seven Knights
1 public static WXMediaMessage fromBundle(Bundle bundle) {
2   String cName = bundle.getString("_wxobject_identifier_");
3   Class clz = Class.forName(cName);
4   IMediaObject media = (IMediaObject) clz.newInstance();
5   media.unserialize(bundle); }
6 public class WXFileObject implements IMediaObject {
7   public void unserialize(Bundle bundle) {
8     Object s = bundle.getString("_wxfileobject_filePath");
9     ... } }
```

Figure 3: Unresolved Bundles.

In this code snippet, different types of objects are created (line 4) to handle different types of media data according to their unique identifiers stored in a `Bundle` (line 2), which is constructed according to the types of media introduced by third-party apps. `IMediaObject` is an interface implemented by eight types (i.e., classes) of media, with only `WXFileObject` shown partially. Therefore, `cName` represents the name of one of these eight classes. In line 4, `media` points to a reflectively created object of the class identified by `cName`. In line 5, a call is made to `unserialize()` on the receiver object pointed to by `media` with `bundle` as its argument.

If we apply again FLOWDROID to detect data leaks in this app, by relying on string inference to resolve the reflective calls in the app, then `cName` will be null, since the behaviors of bundles are not modeled in FLOWDROID. As a result, the reflectively created object in line 4 and the subsequent call on this object in line 5 will be ignored.

By being IIE-aware, RIPPLE will infer the inputs retrieved from `bundle` to resolve the call to `clz.newInstance()` in line 4. By taking advantage of the post-dominant cast `IMediaObject` for this reflective call, RIPPLE deduces that `media` points to potentially eight objects with their types ranging over all the eight classes implementing `IMediaObject`, confirmed by manual code inspection. As a result, a total of 37 caller-callee edges, together with 16 sensitive data sources, which would otherwise be missed by string inference, are discovered in lines 3 – 5 directly or indirectly. Currently, these 16 sensitive data sources do not flow to any sinks but may do so in a future app release. The resulting leaks will be then detected by FLOWDROID, assisted by RIPPLE.

2.4 Unmodeled Services

The Android framework provides an abstraction of abundant services for a mobile device, such as obtaining the device status, making phone calls, and sending text messages, which are all related to critical program behaviors. These services are usually initialized during system startup and subsequently used by calling the factory methods in the Android framework with often reflective calls involved. Unsound modeling for Android’s system-wide services can be an important source of incomplete information in IIEs. The code snippet in Figure 4 taken from a text message management app named *GO SMS Pro* illustrates this issue.

```
Android App Name: Go SMS Pro
1 public void getSubscriberId() {
2   Class clz = Class.forName("android.telephony.TelephonyManager");
3   Method getDefMtd = clz.getMethod("getDefault");
4   Object telephonyManager = getDefMtd.invoke(null);
5   Method getSubIdMtd = clz.getMethod("getSubscriberId");
6   String id = (String) getSubIdMtd.invoke(telephonyManager);
7   ... }
```




Figure 4: Unmodeled services.

In line 2, `clz` represents a class metaobject for `android.telephony.TelephonyManager`. In line 3, `getDefMtd` represents a method metaobject for a static method named `getDefault` in `clz`. In line 4, this method is invoked reflectively, with its returned object, an instance of `clz`, assigned to `telephonyManager`. In lines 5 – 6, a method metaobject, `getSubIdMtd`, for an instance method named `getSubscriberId` in `clz` is created and then invoked reflectively on the receiver object pointed to by `telephonyManager`.

In this code snippet, all the class and method names are string constants. Thus, regular string inference can resolve precisely the reflective targets at all the reflective calls shown. However, this still does not enable the target methods invoked in line 6 to be analyzed, because the `getDefault` method invoked in line 4 is part of the hidden API and thus not available for analysis. Thus, `telephonyManager` is null, causing the reflective call in line 6 to be skipped.

RIPPLE is aware of the existence of unmodeled services. By examining the class type in the `getSubIdMtd` metaobject, RIPPLE concludes that `telephonyManager` points to an object of type `android.telephony.TelephonyManager`. As a result, the reflective call in line 6 can be resolved, resulting in the target method `getSubscriberId` to be discovered. For this app, FLOWDROID is unscalable. Otherwise, the potential data link as shown will be detected automatically.

Finally, if the `getDefault` method is native with its method body unmodeled but available for analysis, then RIPPLE will be able to infer in line 4 that `telephonyManager` may point

to an object of type `android.telephony.TelephonyManager`. As a result, the reflective call in line 6 can also be resolved.

3. METHODOLOGY

Figure 5 depicts an overview of RIPPLE, an IIE-aware reflection analysis introduced in this paper for Android apps. To handle IIEs effectively, RIPPLE resolves reflective calls in the presence of incomplete information about these calls, so that their induced caller-callee edges can be discovered.

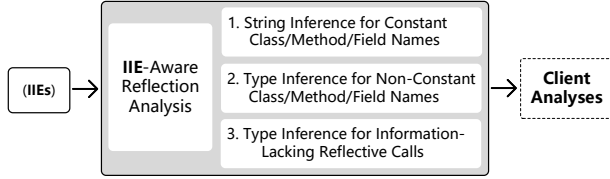


Figure 5: An overview of RIPPLE.

We have developed RIPPLE by leveraging recent advances on reflection analysis for Java [10, 11]. Conceptually, RIPPLE performs reflection analysis by distinguishing three cases:

- **Case 1. String Inference for Constant Strings.** If class/method/field names used at reflective calls are string constants, regular string inference is conducted.
- **Case 2. Type Inference for Unknown Strings.** If class/method/field names are non-constant but non-null strings, which may be read from configuration files or command lines, then type inference that was previously introduced for Java programs [10, 11, 13, 22] can be leveraged.
- **Case 3. Type Inference for Information-Missing Reflective Calls.** Again, type inference is performed to infer the missing information at reflective calls in the following three categories, as reviewed in Section 2:
 - **Null-Name.** Class, method or field names are null, as illustrated in Figure 1 (with `cName = null` for undetermined intents), Figure 2 (with `mName = null` for behavior-unknown libraries), and Figure 3 (with `cName = null` for unmodeled bundles). We will replace a null object by an unknown string and then go back to Case 2.
 - **Missing-RecvObj.** Given a call to `mtd.invoke(y, ...)`, a target method pointed to by `mtd` does not have a corresponding receiver object pointed to by `y`, as illustrated in Figure 4 for the `getSubscriberId` method invoked in line 6, where `telephonyManager = null`. If we know the class type of the target method pointed to by `mtd`, its corresponding receiver object can be inferred.
 - **Missing-RetObj.** Given `x = mtd.invoke(...)`, a target method pointed to by `mtd` is available for analysis but its method body is unmodeled. This can happen to `telephonyManager = getDefMt.invoke(null)` in Figure 4, as discussed in Section 2, when the `getDefault` method is hypothetically assumed to be an unmodeled native method. In general, if we know the return type of the target method pointed to by `mtd`, then objects of this type or its subtypes are created and assigned to `x`.

To the best of our knowledge, RIPPLE is the first automated reflection analysis (without relying on user annotations) for handling Cases 2 and 3 and also the first for handling Cases 1 – 3 in a unified framework for Android apps.

In RIPPLE, reflection analysis is performed together with pointer analysis mutually recursively, as effectively one single analysis. Once the entire analysis for an app is over, its call graph is readily available (at the same time).

4. FORMALISM

We formalize RIPPLE as a form of reflection analysis, performed together with a flow- and context-insensitive pointer analysis. We restrict ourselves to a small core of the Java reflection API. For simplicity, we consider only instance methods as static methods are handled similarly.

4.1 Notations

Figure 6 gives the domains used in our formalism. The abstract heap objects are labeled by their allocation sites. \mathbb{C} represents the set of class metaobjects and \mathbb{M} the set of method metaobjects. The class type of a class or method metaobject is identified by its superscript and the signature of a method metaobject, which consists of the method name and descriptor (i.e., return type and parameter types), is identified by its subscript. In particular, u indicates an unknown class type or an unknown method signature (with some parts of the signature being statically unknown).

class type	$t, u \in \mathbb{T}$
variable	$v \in \mathbb{V}$
abstract heap object	$o_1^t, o_2^t, o_-^t, \dots \in \mathbb{O}$
class metaobject	$c^t, c^u, c_-, \dots \in \mathbb{C}$
method metaobject	$m_s^t, m_s^u, m_u^t, m_u^u, m_-^t, \dots \in \mathbb{M} = \mathbb{T} \times \mathbb{S}$
method name	$n \in \mathbb{N}$
method parameter type	$p \in \mathbb{P} = \bigcup_{i=0}^{\infty} \mathbb{T}^i$
method signature	$s, u \in \mathbb{S} = \mathbb{T} \times \mathbb{N} \times \mathbb{P}$

Figure 6: Domains.

4.2 Pointer Analysis

Figure 7 gives a standard formulation of a flow- and context-insensitive Andersen’s pointer analysis. $pts(v)$ represents the points-to set of a pointer v . An array object is analyzed with its elements collapsed to a single field, say, `arr`.

$i : x = \text{new } t()$ [P-NEW]	$x = y$ [P-COPY]
$\frac{\{o_i^t\} \subseteq pts(x)}{x = y.f}$ [P-LOAD]	$\frac{pts(y) \subseteq pts(x)}{x.f = y}$ [P-STORE]
$\frac{o_i^t \in pts(y)}{pts(o_i^t.f) \subseteq pts(x)}$	$\frac{o_i^t \in pts(x)}{pts(y) \subseteq pts(o_i^t.f)}$
$x = y.m(arg_0, \dots, arg_{n-1})$ [P-CALL]	
$\frac{o_i^t \in pts(y) \quad m' = \text{dispatch}(o_i^t, m)}{\{o_i^t\} \subseteq pts(m'_{this}) \quad pts(m'_{ret}) \subseteq pts(x)}$	
$\forall 0 \leq k < n : pts(arg_k) \subseteq pts(m_{p_k})$	

Figure 7: Rules for pointer analysis.

In a reflection-free program, only five types of statements exist. In [P-NEW], o_i^t uniquely identifies the abstract object created as an instance of t at this allocation site, labeled by i . [P-COPY], [P-LOAD] and [P-STORE] are self-explanatory.

In [P-CALL] (for non-reflective calls), the function $\text{dispatch}(o_i^t, m)$ is used to resolve the virtual dispatch of method m on the receiver object o_i^t to be m' . We assume that m' has a formal parameter m'_{this} for the receiver object and $m'_{p_0}, \dots, m'_{p_{n-1}}$ for the remaining parameters, and a pseudo-variable m'_{ret} is used to hold the return value of m' .

$\text{Class } \text{clz} = \text{Class.forName}(\text{cName})$	[C12-FORNAME]
$\frac{o_{\text{String}} \in \text{pts}(\text{cName}) \quad c^- = \text{toClass}(o_{\text{String}})}{\text{pts}(\text{clz}) \supseteq \begin{cases} \{c^t\} & \text{if } c^- = c^t \\ \{c^u\} & \text{if } c^- = c^u \end{cases}}$	
$\text{Method } \text{mtd} = \text{clz.getMethod}(\text{mName}, _)$	[C12-GETMTD]
$\frac{o_{\text{String}} \in \text{pts}(\text{mName}) \quad c^- \in \text{pts}(\text{clz}) \quad s \in \text{toMtdSig}(c^-, o_{\text{String}})}{\text{pts}(\text{mtd}) \supseteq \begin{cases} \{m_s^t\} & \text{if } c^- = c^t \\ \{m_s^u\} & \text{if } c^- = c^u \end{cases}}$	
$i : x = (\text{T}) \text{clz.newInstance}()$	[C12-NEW]
$\frac{c^- \in \text{pts}(\text{clz})}{\text{pts}(x) \supseteq \begin{cases} \{o_i^t\} & \text{if } c^- = c^t \\ \{o_i^u \mid t <: T\} & \text{if } c^- = c^u \end{cases}}$	
$_ = \text{mtd.invoke}(y, _)$	[C12-INVTYPE]
$\frac{o_i^t \in \text{pts}(y) \quad m_s^u \in \text{pts}(\text{mtd})}{\text{pts}(\text{mtd}) \supseteq \{m_s^t\}}$	
$_ = (\text{T}) \text{mtd.invoke}(_, \text{args})$	[C12-INVSIG]
$\frac{m_s^u \in \text{pts}(\text{mtd}) \quad t \ll: T \quad p \in \text{toParaTys}(\text{args})}{\text{pts}(\text{mtd}) \supseteq \{m_s^- \mid s.\text{para} = p \wedge s.\text{ret} = t\}}$	

Figure 8: Rules for Cases 1 and 2 in Figure 5.

4.3 Reflection Analysis: Cases 1 and 2

Figure 8 gives the rules for resolving reflective calls for Cases 1 and 2 in Figure 5 simultaneously. In Case 1, regular string inference for constant class and method names is applied. In Case 2, type inference for non-constant but non-null class and method names is applied. Recall that $<$ denotes the standard subtyping relation.

[C12-FORNAME] handles a `Class.forName(cName)` call. For the auxiliary function $\text{toClass} : \mathbb{O} \rightarrow \mathbb{C}$, $\text{toClass}(o_{\text{String}})$ takes a string object o_{String} and returns its corresponding class metaobject. If o_{String} is a constant, then $\text{toClass}(o) = c^t$, where t is the class type named by `cName`. Otherwise, $\text{toClass}(o_{\text{String}}) = c^u$, since `cName` is a non-constant but no-null string. The missing type u may be inferred later.

[C12-GETMTD] handles a `clz.getMethod(mName, ...)` call analogously. For the auxiliary function, $\text{toMtdSig} : \mathbb{C} \times \mathbb{O} \rightarrow \mathcal{P}(\mathbb{S})$, $\text{toMtdSig}(c^-, o_{\text{String}})$ returns the set of method signatures for the methods declared in or inherited by the class c^- with their method name identified by `mName`. If $c^- = c^u$ but `mName` is a string constant, say, “foo”, then `foo` is recorded: $\text{toMtdSig}(c^-, o_{\text{String}}) = \{(u, \text{foo}, u)\}$. If `mName` is a non-constant but no-null string, then $\text{toMtdSig}(c^-, o_{\text{String}}) = \{(u)\}$. Therefore, this rule distinguishes two cases for every signature $s \in \text{toMtdSig}(c^-, o_{\text{String}})$. If $c^- = c^t$ is a statically known type t , a method metaobject m_s^t is created. Otherwise, a method metaobject m_s^u is created. In both cases, the missing information may be inferred later.

[C12-NEW] handles reflective object allocation at a call to $x = (\text{T}) \text{clz.newInstance}()$, where T symbolizes an intra-procedurally post-dominating type cast for the call if it exists or `java.lang.Object` otherwise. If $c^- = c^t$ is a statically known type t , then $x = \text{clz.newInstance}()$ degenerates into $x = \text{new } t()$ and can thus be handled as in [P-NEW]. Otherwise, $c^- = c^u$. If $\text{T} \neq \text{java.lang.Object}$, then u is inferred to be T or any of its subtypes.

There are two rules for handling reflective method invocation. To infer a target method invoked reflectively, we need to infer its class type, which is handled by [C12-INVTYPE], and its signature, which is handled by [C12-INVSIG].

[C12-INVTYPE] is simple. The class type of a method metaobject m_s^u is inferred to be m_s^t for every possible dynamic type t of every receiver object pointed to by y .

[C12-INVSIG] is more involved in handling $(\text{T}) \text{mtd.invoke}(_, \text{args})$, where T is defined identically as in [C12-NEW]. This rule attempts to infer the missing information in the signature s of a method from its `args` and its possible return type. We write $s.\text{para}$ and $s.\text{ret}$ to identify its parameter types and return type, respectively. The typing relation $\ll:$ is defined by distinguishing two cases. First, $u \ll:$ `java.lang.Object`. Second, if t is not `java.lang.Object`, then $t' \ll: t$ holds if and only if $t' <: t$ or $t <: t'$ holds. Therefore, $s.\text{ret}$ is deduced from a post-dominating cast T (which is not `java.lang.Object`). As for $s.\text{para}$, we infer it intra-procedurally from `args`. For the auxiliary function $\text{toParaTys} : \bigcup_{i=0}^{\infty} \mathbb{V}^i \rightarrow \mathcal{P}(\mathbb{S})$, $\text{toParaTys}(\text{args})$ returns the set of parameter types of a target method invoked with its argument list `args`, computed intra-procedurally for efficiency reasons. If `args` is not defined locally, $\text{toParaTys}(\text{args}) = \emptyset$. Otherwise, let D_i be the set of declared types of all possible variables assigned to the i -th argument $\text{args}[i]$. Let $P_i = \{t' \mid t \in D_i \wedge (t' <: t \vee t <: t')\}$. Then, $\text{toParaTys}(\text{args}) = P_0 \times \dots \times P_{n-1}$. With $s.\text{para}$ or $s.\text{ret}$ inferred for m_s^- , `mtd` is made to point to a new method metaobject m_s^- , where s contains the missing information in u deduced via inference.

4.4 Reflection Analysis: Case 3

Figure 9 gives the rules for resolving reflective calls for Case 3 in Figure 5. There are four rules for handling three categories of incomplete information, NULL-NAME, MISSING-RECVOBJ and MISSING-RETOBJ, as discussed in Section 3.

$\text{Class } \text{clz} = \text{Class.forName}(\text{cName})$	[C3-FORNAME]
$\frac{\text{pts}(\text{cName}) = \emptyset}{\text{pts}(\text{clz}) \supseteq \{c^u\}}$	
$\text{Method } \text{mtd} = \text{clz.getMethod}(\text{mName}, _)$	[C3-GETMTD]
$\frac{\text{pts}(\text{mName}) = \emptyset \quad c^- \in \text{pts}(\text{clz})}{\text{pts}(\text{mtd}) \supseteq \begin{cases} \{m_u^t\} & \text{if } c^- = c^t \\ \{m_u^u\} & \text{if } c^- = c^u \end{cases}}$	
$i : _ = \text{mtd.invoke}(y, _)$	[C3-INVREC]
$\frac{t'' \in (\{t \mid m_t^- \in \text{pts}(\text{mtd})\} \setminus \{o' \mid o' \in \text{pts}(y) \wedge t <: t'\}) \quad t'' \neq \text{java.lang.Object}}{\text{pts}(y) \supseteq \{o_i^{t''} \mid t'' \ll: t''\}}$	
$i : x = \text{mtd.invoke}(_, _)$	[C3-INVRET]
$\frac{m_s^- \in \text{pts}(\text{mtd}) \quad s.\text{ret} = t \quad t' <: t \quad \forall o_i^{t''} \in \text{pts}(x) : t'' \not<: t \quad t \neq \text{java.lang.Object}}{\text{pts}(x) \supseteq \{o_i^{t'}\}}$	

Figure 9: Rules for Case 3 in Figure 5.

[C3-FORNAME] and [C3-GETMTD] deal with NULL-NAME by treating null as a non-constant string and then resorting to [C12-NEW], [C12-INVTYPE] and [C12-INVSIG] to infer the missing information. [C3-FORNAME] handles a `Class.forName(cName)` call with `cName = null` identically as [C12-FORNAME] for handling a `Class.forName(cName)` call when `cName` is non-constant. [C3-GETMTD] handles a `clz.getMethod(mName, ...)` call with `mName = null` identically as [C12-GETMTD] for handling a `clz.getMethod(mName, ...)` call when `mName` is non-constant.

[C3-INVREC] handles MISSING-RECVOBJ by inferring the missing receiver objects pointed to by y from the known class types of all possibly invocable target methods, except that `java.lang.Object` is excluded for precision reasons. This rule covers an important special case when $\text{pts}(y) = \emptyset$.

[C3-INVRET] handles MISSING-RETOBJ by inferring the missing objects returned from a target method that is unmodeled (with its body missing) but available for analysis from the

return type $s.\text{ret}$ of its signature s . Objects of all possible subtypes of $s.\text{ret}$ are included in $pts(x)$, unless x already points to an object of one of these subtypes.

4.5 Transforming Reflective to Regular Calls

Fig. 10 shows how to transform a reflective into a regular call, which will be analyzed by pointer analysis.

$$\frac{\begin{array}{c} x = \text{mtd.invoke}(y, \text{args}) \quad \text{[T-Inv]} \\ \text{m}_s \in pts(\text{mtd}) \quad m' \in MTD(\text{m}_s) \quad o'_i \in pts(\text{args}) \\ o'_j \in pts(o'_i.\text{arr}) \quad t'' \text{ is declaring type of } m'_{p_k} \quad k \in [0, n-1] \quad t' <: t'' \\ \{o'_j\} \subseteq pts(\text{arg}_k) \quad x = y.m'(\text{arg}_0, \dots, \text{arg}_{n-1}) \end{array}}{} \quad \text{[P-CALL]}$$

Figure 10: Rule for *Transformation*.

For the auxiliary function $MTD : \mathbb{M} \rightarrow \mathcal{P}(M)$, where M is the set of methods in the program, $MTD(m_s^t)$ is the standard lookup function for finding the methods in M according to a declaring class t and a signature s for a method metaobject, except that (1) the return type in s is also considered and (2) any u in s is treated as a wild card.

As discussed earlier, args points to a 1-D array of type `Object[]`, with its elements collapsed to a single field arr during the pointer analysis. Let $\text{arg}_0, \dots, \text{arg}_{n-1}$ be the n freshly created arguments to be passed to each potential target method m' found by in $MTD(m_s^t)$. Let $m'_{p_0}, \dots, m'_{p_{n-1}}$ be the n parameters (excluding *this*) of m' , such that the declaring type of m'_{p_k} is t'' . We include o'_j to $pts(\text{arg}_k)$ only when $t' <: t''$ holds in order to filter out the objects that cannot be assigned to m'_{p_k} . Finally, the regular call obtained can be analyzed by [P-CALL] in Figure 7.

5. EVALUATION

We show that RIPPLE is more effective than string reference by addressing the four research questions:

- **RQ1.** Is RIPPLE capable of discovering more reflective targets, i.e., more sound than string inference?
- **RQ2.** Can RIPPLE achieve this with a good precision?
- **RQ3.** Does RIPPLE scale for real-world Android apps?
- **RQ4.** Is RIPPLE effective in enabling existing Android security analyses to detect security vulnerabilities?

To answer RQ1 – RQ3, we examine the reflective targets resolved by RIPPLE in Android apps. To answer RQ4, we investigate how RIPPLE enables FLOWDROID [3], a taint analysis for Android apps, to find more sensitive data leaks.

Real-World Android Apps. We examined the top-chart free apps from Google Play downloaded on 15 April 2016, which are the most popular apps in the official app store. A set of 17 apps is selected, such that they exhibit a wide range of incomplete information with null class and method names and are scalable under FLOWDROID within 2 hours.

State-of-the-Art Reflection Analysis. To resolve reflection for Android apps, there are only two existing static techniques, with both performing string inference, CHECKER [4, 6] and DROIDRA [9]. We cannot compare with CHECKER since its reflection analysis relies on user annotations. We cannot compare with DROIDRA either, since its latest open-source tool (released on 9 September 2016) is unstable [28]. Instead of comparing with CHECKER and DROIDRA directly, we compare RIPPLE with STRINF, which is a simplified RIPPLE that performs regular string inference in Case 1.

Implementation. We have implemented RIPPLE in SOOT, a static analysis framework for Android and Java programs. RIPPLE works with its SPARK, a flow- and context-insensitive pointer analysis, to resolve reflection and points-to information in a program. Based on the results of this joint analysis, the call graph of the program, on which many security analyses such as FLOWDROID operate, can be constructed.

Currently, RIPPLE handles a core part of the Java reflection API: `Class.forName()`, `Class.newInstance()`, `Method.invoke()`, and all four method-introspecting calls, `Method.getMethod()`, `Method.getDeclaredMethod()`, `Method.getMethods()`, and `Method.getDeclaredMethods()`.

Computing Platform. Our experiments are carried out on a Xeon E5-2650 2GHz machine with 64GB RAM running Ubuntu 14.04 LTS. The time measured for analyzing an app by a particular analysis is the average of 20 runs.

5.1 RQ1: More Soundness

Table 1 compares RIPPLE and STRINF in terms of the number of reflective targets discovered at all reflective calls to `Class.newInstance()` and `Method.invoke()`. For each of these two methods, only its calls reachable from the harness `main()` used during the analysis are included. We determine whether a target is true or not by manual code inspection.

By design, RIPPLE always finds every true target that STRINF does. In 11 out of the 17 apps, RIPPLE has successfully discovered more true targets than STRINF. This highlights the importance of making reflection analysis fully IIE-aware for Android apps, by handling not only Case 2 as for Java programs [10, 11, 13, 22] but also Case 3.

In the 17 apps, RIPPLE finds 64 and 168 but STRINF finds only 29 and 131 true targets for `Class.newInstance()` and `Method.invoke()`, respectively. Therefore, for both methods combined, RIPPLE finds 232 but STRINF finds only 160 true targets in total, yielding a net gain of 72 true targets and thus a 45% increase in soundness on reflection analysis.

Let us revisit Figure 1. For the call to `clz.newInstance()` in line 6, RIPPLE infers five reflectively created objects, which are all true targets configured to provide different forms of advertisement. A similar pattern appears also in the app named *Dumb Ways to Die*. Let us consider now Figure 3. For the call to `clz.newInstance()` in line 4, RIPPLE infers 8 reflectively created objects, which are all true targets used for handling eight different types of media according to user inputs. All these targets are missed by STRINF, which relies only on a simple string analysis for string constants.

5.2 RQ2: Precision

Table 1 reveals also the false positive rates for STRINF and RIPPLE. RIPPLE finds a total of 297 reflective targets with 232 true targets, representing a false positive rate of 21.9%. STRINF finds a total of 167 reflective targets with 160 true targets, representing a false positive rate of 4.2%.

Due to 72 more true targets discovered, as discussed in Section 5.1, RIPPLE is regarded to exhibit a satisfactory precision for Android apps. For many security analyses such as security vetting and malware detection, and even debugging, it is important to analyze reflection-related program behaviors even if doing so may cause some false warnings to be triggered. Consider the app in Figure 2. For the call to `logMtd.invoke(null, tag, msg)` in line 6, RIPPLE infers its target to be the six methods, `i()`, `d()`, `w()`, `e()`, `v()` and

Table 1: Soundness and precision.

App (Package Name)	STRINF								RIPPLE							
	Class.newInstance()				Method.invoke()				Class.newInstance				Method.invoke()			
	#Calls		#Targets		#Calls		#Targets		#Calls		#Targets		#Calls		#Targets	
	Reachable	Resolved	Resolved	True	Reachable	Resolved	Resolved	True	Reachable	Resolved	Resolved	True	Reachable	Resolved	Resolved	True
com.facebook.orca	0	0	0	0	7	0	0	0	0	0	0	0	7	1	7	7
com.netmarble.sknightsgb	2	0	0	0	11	4	8	4	2	2	26	11	11	4	8	4
com.productmadness.hovmobile	1	0	0	0	7	5	31	29	1	1	11	1	7	5	31	29
com.facebook.moments	0	0	0	0	5	0	0	0	0	0	0	0	5	1	7	7
me.msgrd.android	0	0	0	0	11	4	4	4	0	0	0	0	11	4	4	4
com.nordcurrent.canteenhhd	3	0	0	0	16	5	6	5	4	3	27	10	20	5	6	5
com.ea.game.simcitymobile_row	0	0	0	0	6	2	2	2	0	0	0	0	6	2	2	2
com.imangi.templerun	0	0	0	0	7	1	1	1	0	0	0	0	7	1	1	1
com.rovio.angrybirds	3	1	1	1	10	3	3	3	3	2	6	6	14	8	13	11
com.sgn.pandapop_gp	0	0	0	0	13	3	3	3	0	0	0	0	13	3	3	3
com.gameloft.android.ANMP.GloftA8HM	1	1	16	16	9	0	0	0	1	1	16	16	9	0	0	0
com.appsorama.kleptocats	2	2	5	5	3	0	0	0	2	2	5	5	3	1	6	4
air.au.com.metro.DumbWaysToDie	1	0	0	0	18	8	34	34	1	1	5	5	21	11	37	37
com.ketchapp.twist	3	2	5	5	8	2	2	2	3	2	5	5	8	3	8	6
com.stupefix.legend	4	2	2	2	1	0	0	0	4	3	13	5	1	0	0	0
com.maxgames.stickwarlegacy	1	0	0	0	2	1	1	1	1	0	0	0	2	2	7	5
air.com.tutotoons.app	0	0	0	0	14	7	43	43	0	0	0	0	14	7	43	43
animalhairsalon2jungle.free																

`wtf()` in class `Log`, where the last two are false positives, enabling FLOWDROID to report 12 leaks via `msg` from two data sources, with 4 from `v()` and `wtf()` being false positives.

We can lift the precision of RIPPLE by improving the precision of its collaborating analyses, as discussed in [28].

Table 2: Efficiency and effectiveness.

App Package Name	STRINF			RIPPLE		
	CG Edges	Analysis Time (s)	Total Leaks	CG Edges	Analysis Time (s)	Total Leaks
com.facebook.orca	5598	2.1	15	5605	2.2	15
com.netmarble.sknightsgb	12148	8.0	96	12779	8.4	142
com.productmadness.hovmobile	6278	3.1	44	6480	3.2	48
com.facebook.moments	6647	2.6	10	6654	2.7	10
me.msgrd.android	11064	4.8	25	11064	4.9	25
com.nordcurrent.canteenhhd	16759	10	184	21625	12.8	289
com.ea.game.simcitymobile_row	8403	4.2	50	8403	4.4	50
com.imangi.templerun	10592	2.5	54	10592	2.6	54
com.rovio.angrybirds	13448	7.4	66	17384	10.2	120
com.sgn.pandapop_gp	10588	5.9	575	10590	5.9	575
com.gameloft.android.ANMP.GloftA8HM	16015	7.2	144	16016	6.6	144
com.appsorama.kleptocats	3707	2.8	39	3714	3.3	39
air.au.com.metro.DumbWaysToDie	10312	5.3	26	11679	5.9	103
com.ketchapp.twist	14144	8.7	97	14151	8.7	109
com.stupefix.legend	8852	3.4	47	9066	3.5	47
com.maxgames.stickwarlegacy	3831	2.5	5	3844	2.4	17
air.com.tutotoons.app						
animalhairsalon2jungle.free	12075	5.5	9	12075	5.7	9

5.3 RQ3: Scalability

Table 2 compares the analysis times of STRINF and RIPPLE. For all the 17 apps except `canteenhhd` and `angrybirds`, RIPPLE finishes in under 10 secs. For all the 17 apps, STRINF and RIPPLE spend 86.0 and 93.4 seconds, respectively.

5.4 RQ4: Security Analysis

Table 2 also compares STRINF and RIPPLE in terms of their effectiveness for enabling FLOWDROID to find sensitive data leaks in Android apps. For each analysis, FLOWDROID calls it iteratively to build a harness for an app (by modeling more and more callbacks discovered) until a fixed-point is reached. FLOWDROID will then perform a flow- and context-sensitive taint analysis on the inter-procedural CFG, which is constructed based on the call graph (CG) that is computed for the app with respect to the final harness obtained.

For each app, RIPPLE’s CG is a super-graph of STRINF’s and RIPPLE’s leak count is no smaller than STRINF’s. For 10 out of the 17 apps, FLOWDROID reports the same number of leaks for each app under both analyses. For the remaining 7 apps, FLOWDROID reports 310 more leaks under RIPPLE than STRINF (highlighted by the numbers in bold in Table 2). RIPPLE’s ability in finding more true reflective targets than STRINF, as shown in Table 1, has paid off.

Let us revisit two examples discussed in Section 2 to understand reflection-induced privacy violations. Consider Figure 1 first. Due to an undermined intent, `cName = null`. RIPPLE infers five reflectively created objects for `Class.newInstance()` in line 6 based on its post-dominant cast `MMBaseActivity`. As discussed in Section 2.1, all these five objects are true targets configured to provide different forms of advertisement, enabling FLOWDROID to detect 49 leaks that are missed by STRINF, on the methods called directly or indirectly on these objects. For this entire app, FLOWDROID finds 54 more leaks under RIPPLE than STRINF. A similar code pattern also appears in *Dumb Ways to Die* where FLOWDROID finds 77 more leaks under RIPPLE than STRINF.

Let us now consider Figure 2. RIPPLE infers that `i()`, `d()`, `w()`, `e()`, `v()` and `wtf()` in class `Log`, where the last two are false positives, are the potential targets invoked at `logMtd.invoke(null, tag, msg)` in line 6. Some sensitive data may be accidentally passed to `msg` and get written to log files, resulting in potential security vulnerabilities. Due to the six target methods discovered, FLOWDROID finds 12 data leaks (= 2 sources \times 6 sinks) from two sensitive sources, of which 4 from `v()` and `wtf()` are false positives.

6. RELATED WORK

Android Apps. Ernst et al. [6] presented CHECKER, a data-flow analysis for Android apps with reflective calls handled later [4]. As for the reflection resolution approach used, CHECKER performs regular string inference as DROIDRA for constant class and method names but requires user annotations to handle non-constant class and method names. Li et al. [9] introduced DROIDRA, a string inference analysis for resolving reflection in Android apps. Rasthofer et al. [18] developed HARVESTER, an approach for automatically extracting runtime values, such as some class and method names, from Android apps. Zhauniarovich et al. [29] introduced STADYNA, a system that interleaves static and dynamic analysis in order to reveal the program behaviors caused by dynamic code update techniques, such as reflection.

Java Programs. There are several reflection analysis techniques for Java programs [10, 11, 13, 22]. Earlier, Livshits et al. [13] suggested to discover reflective targets by tracking the flow of string constants representing class/method/field names and infer reflective targets based on post-dominating type casts for `Class.newInstance()` calls if their class names are statically unknown strings. Recently, Li. et al. intro-

duced ELF [10] and SOLAR [11] to apply sophisticated type inference to resolve reflective targets effectively. In particular, SOLAR is able to accurately identify where reflection is resolved unsoundly or imprecisely. In addition, it provides a mechanism to balance soundness, precision and scalability, representing a state-of-the-art solution for Java. A recent program slicing technique, called program tailoring [12], can also be leveraged to resolve reflection calls precisely. However, all the reflection analysis techniques proposed for Java cannot resolve a reflective call fully if the data-flows needed (e.g., class or method names) at the call are null.

Reflection analysis usually works together with pointer analysis to discover the targets at reflective calls. Many pointer analysis techniques for Java exist [14, 15, 21, 23–25].

7. CONCLUSION

In this paper, we introduce a reflection analysis for Android apps for discovering the behaviors of reflective calls, which can cause directly or indirectly security vulnerabilities. We advance the state-of-the-art reflection analysis for Android apps, by (1) bringing forward the ubiquity of IIEs for static analysis, (2) introducing RIPPLE, the first IIE-aware reflection analysis, and (3) demonstrating that RIPPLE can resolve reflection in real-world Android apps precisely and efficiently, and consequently, improve the effectiveness of downstream Android security analyses.

Acknowledgement

This research is supported by Australian Research Council grants, DP150102109 and DP170103956.

8. REFERENCES

- [1] Virusshare project. <http://virusshare.com/>.
- [2] S. Arzt and E. Bodden. Stubdroid: automatic inference of precise data-flow summaries for the Android framework. In *ICSE '16*, pages 725–735.
- [3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI '14*, pages 259–269.
- [4] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. D. Ernst, et al. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *ASE '15*, pages 669–679.
- [5] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *POPL '15*, pages 553–566.
- [6] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *CCS '14*, pages 1092–1104.
- [7] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in Android applications. In *S&P '16*, pages 377–396.
- [8] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *NDSS '15*.
- [9] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. DroidRA: taming reflection to support whole-program analysis of Android apps. In *ISSTA '16*, pages 318–329.
- [10] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *ECOOP '14*, pages 27–53.
- [11] Y. Li, T. Tan, and J. Xue. Effective soundness-guided reflection analysis. In *SAS '15*, pages 162–180.
- [12] Y. Li, T. Tan, Y. Zhang, and J. Xue. Program tailoring: slicing by sequential criteria. In *ECOOP '16*.
- [13] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *APLAS '05*, pages 139–160.
- [14] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. *CC '13*, pages 61–81.
- [15] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. *ISSTA '12*, pages 1–11.
- [16] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *POPL '16*, pages 469–484.
- [17] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE '15*, pages 77–88.
- [18] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *NDSS '16*.
- [19] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *CODASPY '13*.
- [20] F. Ruiz. “fakeisntaller” leads the attack on Android phones. *McAfee Labs Website*, Oct, 2012.
- [21] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO '12*, pages 264–274.
- [22] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer. More sound static handling of Java reflection. In *APLAS '15*, pages 485–503.
- [23] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. *POPL*, pages 17–30, 2011.
- [24] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. *PLDI*, pages 387–400, 2006.
- [25] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. *SAS '16*, pages 489–510.
- [26] E. Tinaztepe, D. Kurt, and A. Güleş. Android obad. *Technical Analysis Paper*, 2013.
- [27] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE '15*, pages 89–99.
- [28] Y. Zhang, T. Tan, Y. Li, and J. Xue. Ripple: Reflection analysis for Android apps in incomplete information environments. *arXiv preprint arXiv:1612.05343*, 2016.
- [29] 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 *CODASPY '15*, pages 37–48. ACM.