Pós-Graduação em Ciência da Computação

STATIC ANALYSIS OF IMPLICIT CONTROL FLOW: RESOLVING JAVA REFLECTION AND ANDROID INTENTS

By

PAULO DE BARROS E SILVA FILHO

M.Sc. Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE/2016

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

PAULO DE BARROS E SILVA FILHO

"STATIC ANALYSIS OF IMPLICIT CONTROL FLOW: RESOLVING JAVA REFLECTION AND ANDROID INTENTS"

A M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Marcelo d'Amorim*

RECIFE, 2016

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S586s Silva Filho, Paulo de Barros e.

Static analysis of implicit control flow: resolving Java reflection and Android intents / Paulo de Barros e Silva Filho. – 2016.
51 f.: il., fig., tab.

Orientador: Marcelo d'Amorim.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.
Inclui referências.

1. Engenharia de software. 2. Compiladores. 3. Segurança da informação. I. d'Amorim, Marcelo. (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2016-046

Paulo de Barros e Silva Filho

STATIC ANALYSIS OF IMPLICIT CONTROL FLOW: RESOLVING JAVA REFLECTION AND ANDROID INTENTS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 04/03/2016.

BANCA EXAMINADORA

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática / UFPE

Prof. Dr. Lucas Carvalho Cordeiro
Departamento de Eletrônica e Computação / UFAM

Prof. Dr. Marcelo Bezerra d'Amorim
Centro de Informática / UFPE
(Orientador)

Acknowledgements

I would like to express my thanks to everyone who helped me along my journey, notably:

- My advisor, Marcelo d'Amorim, for everything he teached me during the last three years. It was a great and memorable time of my life.
- My co-advisor, Michael D. Ernst, for every good piece of knowledge and advice he shared with me.
- My working colleagues, both from UFPE and University of Washington, for every shared moment of joy and pain.
- FACEPE, for funding my studies.
- My girlfriend, Lais, and friends, for being able to keep my mind away from studies once in a while.
- My parents, Paulo and Grace Anne, for all the support and unconditional love.

—THERE'S NO SUCH THING AS A FREE LUNCH. (Milton Friedman)

Resumo

Fluxo de controle implícito, ou indireto, permite que haja uma transferência de controle para um procedimento sem que esse procedimento seja invocado de forma explícita pelo programa. Fluxo de controle implícito é um padrão de projeto comum e bastante utilizado na prática, que adiciona flexibilidade no design de um sistema. Porém, é um desafio para uma análise estática ter que computar e verificar propriedades sobre um sistema que usa fluxos de controle implícito.

Quando uma análise estática encontra uma chamada a uma procedimento, geralmente a análise aproxima o comportamento da chamada de acordo com o sumário do método, generalizando de uma forma conservadora os efeitos da chamada ao procedimento. Em trabalho anterior, uma análise estática de segurança foi desenvolvida para aplicações Android, mas falhou em obter uma alta precisão na presença de fluxos de controle implícito.

Este trabalho apresenta uma análise estática para dois tipos de fluxos de controle implícito que aparecem frequentemente em aplicações Android: Java reflection e Android intents. Nas nossas análises, o sumário de um método é a assinatura do método. Nossas análises ajudam a descobrir para onde o controle flui e que dados estão sendo passados. Essa informação melhora a precisão de outras análises estáticas, que não precisam mais tomar medidas conservadoras na presença de fluxo de controle implícito.

Nós implementamos a nossa técnica em Java. Nós melhoramos uma análise de segurança existente através de um tratamento mais preciso em casos de reflection e intents. Em um estudo de caso envolvendo dez aplicações Android reais que usam reflection e intents, a precisão da análise de segurança aumentou em duas ordens de magnitude. A precisão de outras duas análises estáticas também foi melhorada.

Palavras-chave: Fluxo de controle implícito. Análise estática. Java reflection. Android intents. Análise de segurança.

Abstract

Implicit or indirect control flow allows a transfer of control to a procedure without having to call the procedure explicitly in the program. Implicit control flow is a staple design pattern that adds flexibility to system design. However, it is challenging for a static analysis to compute or verify properties about a system that uses implicit control flow.

When a static analysis encounters a procedure call, the analysis usually approximates the call's behavior by a summary, which conservatively generalizes the effects of any target of the call. In previous work, a static analysis that verifies security properties was developed for Android apps, but failed to achieve high precision in the presence of implicit control flow.

This work presents static analyses for two types of implicit control flow that frequently appear in Android apps: Java reflection and Android intents. In our analyses, the summary of a method is the method's signature. Our analyses help to resolve where control flows and what data is passed. This information improves the precision of downstream analyses, which no longer need to make conservative assumptions about implicit control flow, while maintaining the soundness.

We have implemented our techniques for Java. We enhanced an existing security analysis with a more precise treatment of reflection and intents. In a case study involving ten real-world Android apps that use both intents and reflection, the precision of the security analysis was increased on average by two orders of magnitude. The precision of two other downstream analyses was also improved.

Keywords: Implicit control flow. Static analysis. Java reflection. Android intents. Security analysis.

List of Acronyms

IFC	Information Flow Checker	36
CCP	Component Communication Pattern	28
RR	Reflection Resolution	37
INT	Intent Analysis	37
LOC	Lines of code	35

List of Figures

List of Tables

Contents

1

Introduction

Programs are easier to understand and analyze when they use explicit control flow: that is, each procedure call invokes just one target procedure. However, explicit control flow is insufficiently flexible for many important domains, so implicit control flow is a common programming paradigm. For example, in object-oriented dispatch a method call invokes one of multiple implementations at run time. Another common use of implicit control flow is in design patterns, many of which add a level of indirection in order to increase expressiveness. This indirection often makes the target of a procedure call more difficult to determine statically.

1.1 Problem Overview

Implicit control flow is a challenge for program analysis. When a static analysis encounters a procedure call, the analysis usually approximates the call's behavior by a summary, which conservatively generalizes the effects of any target of the call. If there is only one possible target (as with a normal procedure call) or a small number that share a common specification (as with object-oriented dispatch), the summary can be relatively precise. However, if the set of possible targets is large, then a conservative static analysis usually uses a very weak specification, causing it to yield an imprecise result. The imprecision is caused by a lack of information about possible call targets and about the types of data passed as arguments at each call.

The goal of this work is to provide a sound and sufficiently precise estimate of potential call targets and of the encapsulated data communicated in implicit invocations, in order to improve the precision of downstream program analyses. We have previously implemented a static analysis in the security domain (ERNST et al., 2014), and we noticed that the largest challenge to analyze Android apps is their use of reflection and intents, and this led us to our research on resolving implicit invocation. Therefore, our evaluation focuses on a particular domain — Android mobile apps — but the idea could be extended to other domains where implicit invocation may degrade the performance of downstream program analyses, such as iOS mobile apps or Java applications.

Reflection permits a program to examine and modify its own data or behavior (SMITH, 1982). Our interest is in use of reflection to invoke procedures. For example, in Java an object *m* of type `Method` represents a method in the running program; *m* can be constructed in a variety of ways, including by name lookup from arbitrary strings. Then, the Java program can call *m*`.invoke(...)` to invoke the method that *m* represents. Other programming languages provide similar functionality, including C#, Go, Haskell, JavaScript, ML, Objective-C, PHP, Perl, Python, R, Ruby, and Scala. For example, since (ERNST et al., 2014) is a security analysis, when it encounters a reflective call it conservatively assumes that any method could be called. This assumption reduces the precision of the analysis by a large factor.

Android intents are the standard inter-component communication mechanism in Android. They are used for communication within an app (an app may be made up of dozens of components), between apps, and with the Android system. An Android component can send or broadcast intents and can register interest in receiving intents. The Android architecture shares similarities with blackboard systems and other message-passing and distributed systems. As another example, when (ERNST et al., 2014) encounters an intent being sent, it conservatively assumes that this intent could be received by any component, reducing the precision of the analysis by a large factor.

It is crucial for a static analysis in the security domain to be sound. It should also have high precision in order to spare the time spent manually checking for false positives. We are not aware of a previous solution that handles reflection and intents soundly and with high precision, and this is what this work addresses.

1.2 Solution Overview

By default, a sound program analysis must treat reflection and intents conservatively — the analysis must assume that anything could happen at uses of reflection and intents, making its results imprecise. We have built a simple, conservative, and quite precise static analysis that models the effects of reflection and intents on program behavior. The key idea is to resolve implicit control and data flow first to improve the estimates of what procedures are being called and what data is being passed; as a result, those constructs introduce no more imprecision into a downstream analysis than a regular procedure call does.[1]

Both control flow and data flow are important. For reflection, our approach handles control flow by analyzing reflective calls to methods and constructors to estimate which classes and methods may be manipulated, and it handles data flow via an enhanced constant propagation. For intents, our approach handles control flow by using previous work (OCTEAU et al., 2013) to obtain component communication patterns, and it handles data flow by analyzing the payloads that are carried by intents.

[1] Our approach does not change the program's operations, either on disk or in memory in the compiler; see Section 3.2.

1.3 Contribution

We have implemented our approach for Java. We evaluated our implementation on open-source apps, in the context of three existing analyses, most notably an information flow type system for Android security (ERNST et al., 2014). Most Android apps use reflection and/or intents, so accurately handling reflection and intents is critical in this domain. Unsoundness is unacceptable because it would lead to security holes, and poor precision would make the technique unusable due to excessive false-positive alarms.

The reflection and intent analyses increased the precision of the information flow type system by two orders of magnitude, and they also improved the precision of the other two analyses. Furthermore, they are easy to use and fast to run. Our implementation is freely available in the SPARTA toolset (`http://types.cs.washington.edu/sparta/`), including source code and user manual, and the reflection analysis is also integrated into the Checker Framework (`http://checkerframework.org/`).

1.4 History of Publications

This work was developed in colaboration with René Just, Suzanne Millstein, Paul Vines, Michael D. Ernst (University of Washington), Werner Dietl (University of Waterloo) and Marcelo d'Amorim. Together, we published a paper in the 30th Annual International Conference on Automated Software Engineering (ASE 2015).

During that time, I was able to visit the University of Washington twice, where I stayed for 5 months during the first time and for 3 months during the second. We have also published the Information Flow Checker (ERNST et al., 2014) together, which was used to evaluate this work.

Aside from that, I also had the opportunity to publish SPLat (KIM et al., 2013), a technique for reducing the combinatorics problem when testing configurable systems. This was done in colaboration with Sabrina Souto (Marcelo's student at that time), a research group from the University of Texas led by Don Batory, and a research group from the University of Illinois led by Darko Marinov. I was responsible for performing experiments and fixing bugs.

1.5 Outline

The rest of this work is structured as follows. Chapter 2 presents a background of Android, reflection and the Checker Framework, together with two motivating examples. Sections 3 and 4 present our analyses that resolve reflection and intents. Chapter 5 evaluates how the reflection and intent analyses improve the precision of downstream analyses. Chapter 6 shows how the type inference rules reduce programmer effort. Chapter 7 discusses related work, and Chapter 8 concludes.

2

Background and Motivation

This chapter presents background material related to Android, Java reflection and the Checker Framework, which are important for the rest of this thesis, and details two motivating examples used throughout this thesis.

2.1 Android

Android apps are packaged in compressed .apk files, which contain code files, resource files (*e.g.*, image and audio files), and a manifest file providing meta-data such as required permissions and a list of system-wide events that the app is interested in listening from the environment.

The concept of a *component* is central in Android. An Android app consists of a set of components of the following types: Activities, Services, (Broadcast) Receivers, and (Content) Providers. An Activity is a component with a screen for users to interact. A Service is a component that can perform long-running operations in the background. Receivers are components that listen to broadcast messages sent by other components (not necessarily from the same app). Content providers manage access to app data.

2.1.1 Component communication

Android components communicate asynchronously through messages called *intents*. An intent encapsulates data, which serves to initialize a component. It is common to refer to a component that sends (/receives) an intent as a sending component (/receiving component). Conceptually, an intent is a map of string keys to arbitrary objects. We describe in the following the elements that comprise the intent payload.

Intent Extras. Intent extras are a set of key-value pairs used to communicate data between components. For example, a component can communicate the identity of a user to another component passing a string “extra” containing the corresponding username of that user.

Intent Filters. An intent filter specifies interest of a given component in particular (intent) messages. The fields *action*, *category*, and *data* of an intent are typically used to specify these filters. Although filters can be defined within apps, using dynamic (Broadcast)Receivers, they are most often declared in the manifest file.

Detailed material about Android can be found elsewhere (GOOGLE, 2008).

2.2 Reflection

Reflection is a metaprogramming mechanism that enhances the flexibility and expressiveness of a programming language. Its primary purpose is to enable a program to dynamically exhibit behavior that is not expressed by static dependencies in the source code.

For example, in Java an object *m* of type `Method` represents a method in the running program; *m* can be constructed in a variety of ways, including by name lookup from arbitrary strings. Then, the Java program can call *m*`.invoke(...)` to invoke the method that *m* represents.

Reflection is commonly used for the following four use cases, among others. (1) Provide backward compatibility by accessing an API method that may or may not exist at run time. The reflective code implements a fallback solution so the app can run even if a certain API method does not exist, e.g., on older devices. (2) Access private API methods and fields, which offer functionality beyond what is provided by the public API. (3) Implement design patterns such as duck typing. (4) Code obfuscation to make it harder to reverse-engineer the program, e.g., code that accesses premium features that require a separate purchase.

The Android developer documentation encourages the use of reflection to provide backward compatibility and for code obfuscation (cases 1 and 4 above), and 39% of apps in the F-Droid repository (F-DROID, 2010) use reflection.

2.3 Checker Framework

The Checker Framework (DIETL et al., 2011) enhances Java's type system, allowing compiler plug-ins ("checkers") that find bugs and verify their absence. It is an open source project that provides support for writing custom checkers.

We implemented our solutions as type-checkers for the Checker Framework. In fact, every type system mentioned in this thesis has an associated type-checker implementation available on the Checker Framework[1], except for the noninterference type system mentioned on the next section.

2.4 Motivating examples

Our work improves the precision of a downstream static analysis, by eliminating false positive warnings in cases of implicit control flows. Imprecision due to implicit control flow

[1] `http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html`

affects every static analysis. One example of static analysis is a type-checking procedure – a type-checker performs a static analysis to verify that all types are used correctly in a program. This is the procedure performed by the type-checkers on the Checker Framework. In general, on every (pseudo-)assignment a type-checker verifies that the type of the value on the right-hand-side is a subtype of the type of the variable on the left-hand-side of the assignment. For concreteness, consider a noninterference type system (VOLPANO; SMITH, 1997), which guarantees that the program does not leak sensitive data.

The noninterference type system distinguishes high-security-level values from low-security-level values; for brevity, `Secret` and `Public` values. The static property checked is that values in `Secret` variables are not assigned to `Public` variables, which could leak sensitive data. Variables and expressions marked `Secret` may hold a `Public` value at run time; this is also expressed as `Public` <: `Secret`, where the symbol "<:" denotes subtyping. To use this type system, a user annotates each type with `Secret` or `Public`, the default being `Public`. The type system is conservative: if it issues no warnings, then the program has no interference and running it does not leak any `Secret` data to `Public` contexts.

When run on the Android app `Aard Dictionary`[2], the noninterference type system issues false positive warnings due to its conservative handling of implicit control flows. When our reflection and intent analyses are integrated into it, the type system remains sound but no longer issues the false positive warnings. Below we show examples that use the noninterference type system and suffer false positives due to implicit control flow, but other type systems suffer similar false positives. Our reflection and intent analyses also help other downstream analyses, as demonstrated in Section 5.1.3.

2.4.1 Reflection

Some calls to `Method.invoke` may return a `Secret` value at run time. Thus, the signature of `Method.invoke` (line 15 of Figure 2.1) must have a `Secret` return type; any other return type in the summary would be unsound. But some calls to `Method.invoke` will return a `Public` value. The conservative signature of `Method.invoke` causes false positive warnings in such cases.

Figure 2.1 illustrates the problem in `Aard Dictionary`. The component `ArticleViewActivity` uses an `ActionBar`, which is a feature that was introduced in version 11 of the Android API. In order to prevent run-time errors for a user who has an older version of Android (and also to enable the app to compile when a developer is using an older version of the Android API), this app uses reflection to call methods related to the `ActionBar`. The noninterference type-checker issues a false positive due to the use of reflection; our reflection analysis (Chapter 3) eliminates the false positive warning.

[2] `http://aarddict.org/`

Figure 2.1: A noninterference type-checker produces a false positive warning on line 7, where the return type of `Method.invoke`, of type `Secret`, is assigned to variable `actionBar` which has declared type `Public`. The call on line 7 always returns a `Public` value at run time (even though other calls to `invoke` may in general return a `Secret` value), so the assignment is safe. When the noninterference type system is augmented by our reflection analysis, it no longer issues the false positive warning. Note that since `Public` is the default type for the noninterference type system, all `Public` annotations could be omitted.

```
 1 class ArticleViewActivity extends Activity {
 2   void onCreate(Bundle savedInstanceState) {
 3     if (android.os.Build.VERSION.SDK_INT >= 11) {
 4       // Android version 11 and later has Action Bar
 5       Method getActionBar =
 6         getClass().getMethod("getActionBar");
 7       @Public Object actionBar = getActionBar.invoke(this);
 8       ...
 9     }
10   }
11 }
12
13 // Library annotations:
14 class Method {
15   @Secret Object invoke(Object obj, Object... args) {...}
16 }
17 class Activity {
18   // Only exists in Android SDK 11 and above.
19   @Public ActionBar getActionBar() {...}
20 }
```

Source: Aardict Dictionary source code.

2.4.2 Android intents

An Android component might send a `Secret` value via an intent message to another component; therefore, the summary for methods that retrieve data from an intent (lines 26–27 of Figure 2.2) must conservatively assume that the data is a `Secret` value. This conservative summary may cause false positive warnings when the data is of type `Public` at run time.

Figure 2.2 shows another example from `Aard Dictionary`. The components `DictionaryMain` and `WordTranslator` use Android intents to communicate. Android intents are messages sent between Android components, and those messages might contain “extras”, which is a mapping of keys to objects. Component `DictionaryMain` creates an intent object `i`, adds `Public`-security extra data to `i`’s extras mapping, and on line 7 calls the Android library method `startActivity` to send the intent. The Android system then calls `WordTranslator.onCreate`, which is declared on line 12. The noninterference type-checker issues a false positive due to the use of intents; our intent analysis (Chapter 4) eliminates the false positive warning.

Figure 2.2: A noninterference type-checker produces false positive warnings on lines 14–16, where the return type of `get*Extra`, of type `Secret`, is assigned to variables with declared type `Public`. The calls on lines 14–16 always return a `Public` value at run time (even though other calls to `get*Extra` may in general return a `Secret` value), so the assignments are safe. When the noninterference type system is augmented by our intent analysis, it no longer issues the false positive warnings.

```
1  class DictionaryMain extends Activity {
2   void translateWord(int source, int target, String word){
3      Intent i = new Intent(this, WordTranslator.class);
4      i.putExtra("source", source);
5      i.putExtra("target", target);
6      i.putExtra("word", word);
7      startActivity(i);
8    }
9  }
10
11 class WordTranslator extends Activity {
12   void onCreate(Bundle savedInstanceState)
13     Intent i = getIntent();
14     @Public int source = i.getIntegerExtra("source");
15     @Public int target = i.getIntegerExtra("target");
16     @Public String word = i.getStringExtra("word");
17     showResult(translate(source, target, word));
18   }
19   String translate(int source, int target, String word) {...}
20   Intent getIntent() {...}
21   void showResult(String result) {...}
22 }
23
24 // Library annotations:
25 class Intent {
26   @Secret Integer getIntegerExtra(String key) {...}
27   @Secret String getStringExtra(String key) {...}
28 }
```

Source: Aardict Dictionary source code.

3

Reflection resolution

Not all uses of reflection can be statically resolved, but our experiments show that many of them can. Whenever the developer runs a code analysis, it is beneficial to the analysis if as much reflection as possible is resolved, in order to reduce false positive warnings. Obfuscation is not compromised, because analysis results, annotations, and other information that is used in-house by the developer need not be provided to users of the software.

Approach for reflection resolution

Without further information about which method is reflectively called, a static analysis must assume that a reflective call could invoke any arbitrary method. Such a conservative assumption increases the likelihood of false positive warnings.

At each call to `Method.invoke`, our analysis soundly estimates which methods might be invoked at runtime. Based on this estimate, our analysis statically resolves the `Method.invoke` call — that is, it provides type information about arguments and return types for a downstream analysis. The results are soundly determined solely based on information available at compile time.

The reflection resolution consists of the following parts:

1. *Reflection type system*: Tracks and infers the possible names of classes, methods, and constructors used by reflective calls. (Section 3.1)
2. *Reflection resolver*: Uses the reflection type system to estimate the signatures of methods or constructors that can be invoked by a reflective call. (Section 3.2)

3.1 Reflection type system

Our reflection type system refines the Java type system to provide more information about array, `Class`, `Method`, and `Constructor` values. In particular, it provides an estimate, for each expression of those types, of the values they might evaluate to at run time.

For arrays, the refined type indicates the length of the array: for example, `@ArrayLen({3,4})` indicates that the array will be of length 3 or 4. For expressions of type `Class`, there are two possible type qualifiers, `@ClassVal` and `@ClassBound`, representing either an exact `Class` value or an upper bound of the `Class` value. The list of possible values is expressed as an array of strings representing fully-qualified types; for example, `@ClassVal("java.util.HashMap")` indicates that the `Class` object represents the `java.util.HashMap` class. Alternatively, `@ClassBound("java.util.HashMap")` indicates that the `Class` object represents `java.util.HashMap` or a subclass of it.

For expressions of type `Method` and `Constructor`, the type qualifier indicates estimates for the class, method name, and number of parameters. For example,

```
@MethodVal(cn="java.util.HashMap",
           mn={"containsKey", "containsValue"},
           np=1)
```

indicates that the method represents either `HashMap.containsKey` or `HashMap.containsValue`, with exactly 1 parameter. Likewise, the `MethodVal` type may have more than one value for the class name or number of parameters. The represented methods are the Cartesian product of all possible class names, method names, and numbers of parameters. For a constructor, the method name is “`<init>`”, so no separate `@ConstructorVal` type qualifier is necessary.

The `MethodVal` type is imprecise in that it indicates the number of parameters that the method takes, but not their type. This means that the type system cannot distinguish methods in the uncommon and discouraged (BLOCH, 2001) case of method overloading. This was a conscious design decision that reduces the verbosity and complexity of the annotations, without any practical negative consequences. In our experiments with more than 300,000 lines of Java code from 35 Android apps, this imprecision in the type system never prevented a reflective call from being resolved.

Our implementation caps the size of a set of values at 10. This cap was never reached in our case studies. If a programmer writes, or the type system infers, a set of values of size larger than 10, then the type is widened to its respective top type. A top type indicates that the type system has no estimate for the expression: the type system’s estimate is that the run-time value could be any value that conforms to the Java type. The top type is the default, and it is represented in source code as the absence of any annotation.

3.1.1 Type checking

The reflection type system enforces standard type system guarantees, e.g. that the right-hand side of an assignment is a subtype of the left-hand side. These typing rules follow those of Java, they are standard for an object-oriented programming language, and they are familiar to programmers. Therefore, we do not detail them in this work. The reflection type system and our implementation are compatible with all Java features, including generics (type polymorphism).

3.1.2 Type inference

Programmers do not need to write type annotations within method bodies, because our system performs local type inference. More specifically, for local variables, casts, and

Figure 3.1: The type of `s` in `C.m2` will be inferred to `@T String` after the call to `m()`.

```
public class C {
  @T String m() { ... }

  void m2() {
   String s = m();
   // From here, type of s is: @T String
   ...
  }
}
```

Source: Made by the authors.

`instanceof` expressions, the absence of any annotation indicates that the type system should infer the most precise possible type from the context. For all other locations — notably fields, method signatures, and generic type arguments — a missing annotation is interpreted as the top type qualifier.

The local type inference is flow-sensitive. It takes advantage of expression typing rules that yield more precise types than standard Java type-checking would. Figure 3.1 shows an example where a method call expression refines the type of a local variable at an assignment.

3.1.2.1 Estimates for values of expressions

We have designed and implemented a dataflow analysis that infers and tracks types providing an estimate for the possible values of each expression. Our implementation goes beyond constant folding and propagation: it evaluates side-effect-free methods, it infers and tracks the length of each array, and it computes a set of values rather than just one. Figure 3.2 shows selected inference rules. Our implementation is available as the Constant Value Checker [1] on the Checker Framework. The reflection type system builds on top of this dataflow analysis.

3.1.2.2 Inference of `@ClassVal` and `@ClassBound`

The reflection type system infers the exact class name (`@ClassVal`) for a `Class` literal (`C.class`), and for a static method call (e.g., `Class.forName(arg)`, `ClassLoader.loadClass(arg)`, ...) if the argument has a sufficiently precise `@StringVal` estimate. In contrast, it infers an upper bound (`@ClassBound`) for instance method calls (e.g., `obj.getClass()`).

An exact class name is necessary to precisely resolve reflectively-invoked constructors since a constructor in a subclass does not override a constructor in its superclass. Either an exact class name or a bound is adequate to resolve reflectively-invoked methods because of the subtyping rules for overridden methods. Figure 3.3 shows selected inference rules.

[1] http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#constant-value-checker

Figure 3.2: Inference rules for `@StringVal`, `@IntVal`, and `@ArrayLen`.

$$\frac{e : \texttt{String} \quad val \text{ is the statically computable value of } e}{e : \texttt{@StringVal(}val\texttt{)}}$$

$$\frac{e : \texttt{int} \quad val \text{ is the statically computable value of } e}{e : \texttt{@IntVal(}val\texttt{)}}$$

$$\frac{e : \texttt{@IntVal(}\pi\texttt{)}}{\texttt{new C[}e\texttt{]} : \texttt{@ArrayLen(}\pi\texttt{)}}$$

$$\frac{}{\texttt{new C[]\{}e_1,\ldots,e_n\texttt{\}} : \texttt{@ArrayLen(}n\texttt{)}}$$

Source: Made by the authors.

Figure 3.3: Selected inference rules for the `@ClassVal`, `@ClassBound`, and `@MethodVal` annotations. Additional rules exist for expressions with similar semantics but that call methods with different names or signatures, and for fields/returns.

$$\frac{fqn \text{ is the fully-qualified class name of } \texttt{C}}{\texttt{C.class} : \texttt{@ClassVal(}fqn\texttt{)}}$$

$$\frac{s : \texttt{@StringVal(}\nu\texttt{)}}{\texttt{Class.forName(}s\texttt{)} : \texttt{@ClassVal(}\nu\texttt{)}}$$

$$\frac{fqn \text{ is the fully-qualified class name of the static type of } e}{e\texttt{.getClass()} : \texttt{@ClassBound(}fqn\texttt{)}}$$

$$\frac{\begin{array}{c}\big(e : \texttt{@ClassBound(}\nu\texttt{)} \ \lor\ e : \texttt{@ClassVal(}\nu\texttt{)}\big)\\ s : \texttt{@StringVal(}\mu\texttt{)} \quad p : \texttt{@ArrayLen(}\pi\texttt{)}\end{array}}{e\texttt{.getMethod(}s,p\texttt{)} : \texttt{@MethodVal(cn=}\nu\texttt{,mn=}\mu\texttt{,np=}\pi\texttt{)}}$$

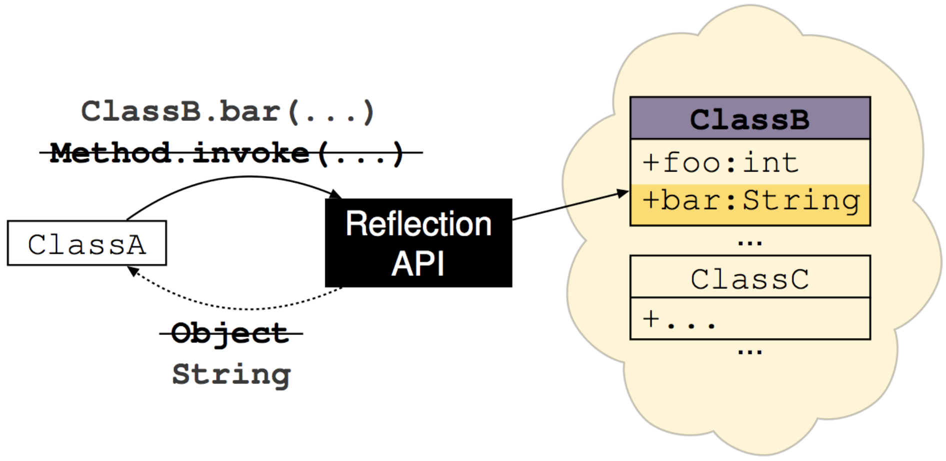
$$\frac{e : \texttt{@ClassVal(}\nu\texttt{)} \quad p : \texttt{@ArrayLen(}\pi\texttt{)}}{e\texttt{.getConstructor(}p\texttt{)} : \texttt{@MethodVal(cn=}\nu\texttt{,mn="<init>",np=}\pi\texttt{)}}$$

Source: Made by the authors.

3.1.2.3 Inference of `@MethodVal`

The reflection type system infers `MethodVal` types for methods and constructors that have been created via Java's Reflection API. A nonexhaustive list of examples includes calls to `Class .getMethod(String name, Class<?>... paramTypes)` and `Class.getConstructor(Class<?>... paramTypes)`. For example, the type inferred for variable `getActionBar` on line 5 of Figure 2.1 is `@MethodVal(cn="ArticleViewActivity", mn="getActionBar", np=0)`. Although Figure 2.1 uses raw (non-parameterized) types, our inference supplies the missing type argument information.

Figure 3.4: Reflection resolver: Procedure summary is narrowed based on the Reflection type system. Program and downstream analysis remains unchanged.



Source: Made by the authors.

3.1.2.4 Inference of field types

For private fields, our type inference collects the types of all assignments to the field, and sets the field type to their least upper bound (lub). If the lub is not a subtype of the declared type, this step is skipped and a type-checking error will be issued at some assignment. The same mechanism works for non-private fields, but the entire program has to be scanned for assignments. At the end of type-checking, the type-checker outputs a suggestion about the field types. The user may accept these suggestions and re-run type-checking to obtain more precise results; we did so in our experiments. Field type inference works for every type system, not just those related to reflection.

3.1.2.5 Method signature inference

Similarly to field type inference, private method parameters are set to the lub of the types of the corresponding arguments, and private method return types are set to the lub of the types of all returned expressions, when those are consistent with the declared types. For non-private methods, the entire program is scanned for calls/overriding and the type-checker outputs suggestions.

Figure 3.3 shows selected inference rules for the reflection type system.

3.2 Reflection resolver

Prior work (see Chapter 7) commonly re-writes the source code or changes the AST within the program analysis tool, changing a call to `Method.invoke` into a call to the method that is reflectively invoked before analyzing the program. This approach interferes with the toolchain, preventing the code from being compiled or run in certain environments. This approach is also at odds with the very purpose of reflection: the program no longer adapts to its run-time environment and loses properties of obfuscation. A final problem is that an analysis may discover facts that cannot be expressed in source code form.

Figure 3.4 shows that our reflection resolver operates differently: it leaves the program unmodified but narrows the procedure summary — the specification of parameter and return types used during modular analysis — for that particular call site only. When the downstream analysis requests the summary at a call to `Method.invoke`, it receives the more precise information rather than the conservative summary that is written in the library source code. This transparent integration means that the downstream analysis does not need to be changed at all to be integrated with the reflection analysis.

3.2.1 Example

Recall the example of Figure 2.1. When the noninterference type system analyzes `getActionBar.invoke(this)` on line 7, it uses a method summary (like a declaration) to indicate the requirements and effects of the call. Ordinarily, it would use the following conservative declaration for `Method.invoke`:

```
@Secret Object invoke(Object recv, Object ... args)
```

However, the reflection type system inferred that the type of variable `getActionBar` is `@MethodVal(cn="ArticleViewActivity", mn="getActionBar", np=0)`. In other words, at run time, the invoked method will be the following one from class `ArticleViewActivity`:

```
@Public ActionBar getActionBar ()
```

Thus, the noninterference type system has a precise type, `Public`, for the result of the `invoke` call. The reflection resolver automatically provides the following precise procedure summary to the downstream analysis, for this call site only:

```
@Public Object invoke(Object recv, Object ... args)
```

As a result, the type system does not issue a false positive warning about the assignment to variable `actionBar` on line 7.

The summary contains not just refined procedure return types as shown above, but also refined parameter types, enabling a downstream analysis to warn about clients that pass arguments that are not legal for the reflectively-invoked method. It would be possible to refine the Java types as well as the type qualifiers (for instance, to warn about possible run-time type cast errors or to optimize method dispatch), but our implementation does not do so.

If the reflectively-called method or constructor cannot be resolved uniquely, the reflection resolver determines the least upper bound of all return values and the greatest lower bound of all parameter and receiver types.

4

Android intent analysis

An Android app is organized as a collection of components that roughly correspond to different screens of an application and to background services.[1] Some apps consist of dozens of components. Intents are used for inter-component communication, both within an app and among different apps. Intents are similar to messages, communicated asynchronously across components. Sending an Android intent implicitly invokes a method on the receiving component, just as making a reflective procedure call implicitly invokes a method. The use of intents is prevalent in Android apps: all top 50 popular paid apps and top 50 popular free apps from the Google Play store use intents (CHIN et al., 2011), the top 838 most popular apps contain a total of 58,989 inter-component communication locations (OCTEAU et al., 2013), and intents are a potential target for attackers to introduce malware (CHIN et al., 2011).

Intents present two challenges to static analyses: (i) control flow analysis, or determining which components communicate with one another, and (ii) data flow analysis, or determining what data is communicated. Both parts are important. An existing analysis, Epicc (OCTEAU et al., 2013), partially solves the control flow challenge. Section 4.1 describes how our implementation uses Epicc to compute component communication. Our key research contribution here is to address the data flow challenge, which has resisted previous researchers. Section 4.2 presents a novel static analysis that estimates the data passed in an Android intent.

The structure of Android intents

In addition to attributes that specify which components may receive the intent, an intent contains a map from strings to arbitrary data, called “extras”. The extras map is used to pass additional information that is needed to perform an action. For example, an intent used to play a song contains the song’s title and artist as extras. An invocation of the `putExtra` method adds a key–value entry to the intent map, which can be looked up via the `getExtra` method call. Without loss of generality, we will consider that every intent attribute (including the action name, categories and data) is an entry in the map of extras. The use of extras is prevalent in Android: of the 1,052 apps in the F-Droid repository (F-DROID, 2010), 69% of the apps has at least one

[1] Activity, Service, BroadcastReceiver, and ContentProvider are the four kinds of Android components. See `http://developer.android.com/guide/components/fundamentals.html#Components`.

intent with extra data. Figure 2.2 shows the common use case of an Android app sending and receiving an intent containing extras.

4.1 Component communication patterns

For simplicity, this work abstracts all methods that send intents as the method `sendIntent`, and all methods that receive an intent as the method `onReceive`. For example, in Figure 2.2, `startActivity()`, called on line 7, is an example of a `sendIntent` method, and the method `getIntent()`, declared on line 20, is an example of an `onReceive` method.

To precisely analyze the types of data sent through intents, our analysis requires `sendIntent` calls to be matched to the declarations of `onReceive` methods they implicitly invoke. We express this matching as Component Communication Pattern (CCP): a set of pairs of the form $\langle \texttt{sendIntent}(a,i),\ \texttt{onReceive}(b,j) \rangle$. Each pair in the CCP indicates that components a and b, possibly from different apps, may communicate through intents i and j, which intuitively denote the actual arguments and formal parameters of the implicit invocation.

To precompute an approximated CCP, our current implementation uses APKParser (APKParser, 2011), Dare (OCTEAU; JHA; MCDANIEL, 2012), and Epicc (OCTEAU et al., 2013). Our implementation inherits Epicc's limitations. Note, however, that Epicc's limitations are not inherent to our intent analysis, and they would disappear if we used a better analysis to compute CCP. As better CCP techniques become available, they can be plugged into our implementation. IC3 (OCTEAU et al., 2015) is Epicc's successor, created by the same research group. We attempted to use IC3, but we discovered a soundness bug: dynamically-registered Broadcast Receivers were not being analyzed. The IC3 authors have confirmed but not fixed the bug[2], so we used Epicc instead. We now discuss sources of imprecision and unsoundness due to Epicc.

Epicc's sources of imprecision. Epicc's lack of support for URIs leads to imprecision since intents with the same action and category but different URIs are conservatively considered equal. As expected of a static analysis, Epicc also cannot handle cases where dynamic inputs determine the identity of receiver components. Epicc also handles this conservatively: all components are considered possible receivers. Furthermore, the points-to and string analyses used by Epicc are also sources of imprecision due to limitations on their implementation. Even with these limitations, all mentioned in (OCTEAU et al., 2013), Epicc reports 91% precision in a case study with 348 apps (OCTEAU et al., 2013).

Epicc's sources of unsoundness. Epicc unsoundly assumes that Android apps use no reflection. We used the type system of Chapter 3 to circumvent this limitation; see Chapter 5. Epicc also unsoundly assumes that Android apps use no native calls, a standard limitation of static analysis that is shared by IC3. We do not circumvent this limitation. Another unsoundness is the closed-world assumption; that is, Epicc assumes that it knows all the apps installed on a

[2] `https://github.com/siis/ic3/issues/1`

device. Our work shares this assumption. Compatibility with Epicc's analysis could be checked whenever an app is installed.

Recall that while finding CCP is necessary, it is not sufficient. Since acceptable solutions exist for finding CCP, the focus of our intent analysis is on the unsolved problem of estimating the payloads of intents, which is discussed below.

4.2 Intent type system

This section presents a type system for Android intents. The type system verifies that the type of data stored within an intent conforms to the declared type of the intent, even in the presence of implicit invocation via intents.

The type system verifies that for any `sendIntent` method call and any `onReceive` method declaration that can be invoked by the call site, the intent type of the argument in the `sendIntent` call is compatible with the intent type of the parameter declared in the `onReceive` method signature.

4.2.1 Intent types

We introduce intent types, which hold key–type pairs that limit the values that can be mapped by a key.

Syntax of intent types. This work uses the following syntax for an intent map type:

```
@Intent("K1" → t1, ..., "Kn" → tn) Intent i = ...;
```

where { *"K1"*,. . . ,*"Kn"*} is a set of literal strings and {*t1*,. . . ,*tn*} is a set of types. The type of variable `i` above consists of a type qualifier `@Intent(...)` and a Java type `Intent`. The regular Java type system verifies the Java type, and our intent type system verifies the type qualifier.

The actual Java syntax used by our implementation is slightly more verbose than that in this work:

```
@Intent(@Entry(key="K1", type="t1"), ...,
        @Entry(key="Kn", type="tn")) Intent i = ...;
```

Semantics of intent types. If variable `i` is declared to have an intent type T, then two constraints hold.

- (C1) The keys of `i` that are accessed must be a subset of T's keys. It is permitted for the run-time value of variable `i` to have more keys than those listed in T, but they may not be accessed. It is also permitted for the run-time value of variable `i` to have fewer keys than those listed in T; any access to a missing key will return `null`.

- (C2) For every key k in T, either k is missing from the run-time key set of `i`, or the value mapped by k in the run-time value of `i` has the type mapped by k in T. This can be more concisely expressed as $\forall k \in \mathit{domain}(T)\,.\,\texttt{i}[k] : T[k]$, where ":" indicates typing and `null` is a value of every non-primitive type.

Example. The example below illustrates the declaration and use of intent types. The symbols `@A`, `@B`, and `@C` denote type qualifiers, such as `@Secret` and `@Public` of the noninterference type system. On the left is the type hierarchy of these type qualifiers. (C1) and (C2) are the two constraints described above.

```
             @Intent("akey" → @C) Intent i = ...
  @A         @A int e1 = i.getIntExtra("akey");     // legal
  / \        @C int e2 = i.getIntExtra("akey");     // legal
 @B  @C      @B int e3 = i.getIntExtra("akey");     // violates (C2)
             i.getIntExtra("otherKey");             // violates (C1)
```

The first line is a creation of an Intent object, which is valid. According to the `@Intent` type of `i`, the type of `i.getIntExtra("akey")` is `@C`. Therefore, on the second line there is an assignment where the type `@C` is on the right-hand-side of the assignment and the type `@A` is on the left-hand-side of the assignment. This is a legal assignment since `@C` is a subtype of `@A`. The third line is also legal since `@C` is a subtype of itself. The fourth line is illegal because it is trying to assign a value of type `@C` on the right-hand-side to a variable of type `@B` on the left-hand-side, but `@B` and `@C` are not related to each other according to the type hierarchy, therefore this assignment violates (C2). The call `i.getIntExtra("otherKey")` violates (C1) since `"otherKey"` is not present in the `@Intent` type of `i`.

4.2.2 Type system rules

Figure 4.1 shows the typing rules for the intent type system. These rules are organized into three categories, according to their purpose. Subtyping rules define a subtyping relation for intent types, well-formedness rules define which constructions are acceptable, and typing judgment rules define the types associated with different language expressions.

4.2.2.1 Subtyping (ST)

Intent type τ_1 is a subtype of intent type τ_2 if the key set of τ_2 is a subset of the key set of τ_1 and, for each key k in both τ_1 and τ_2, k is mapped to the same type.

```
@Intent("akey" → t, "anotherkey" → t) Intent i1 = ...;
@Intent("akey" → t) Intent i2 = ...;
@Intent("anotherkey" → t)) Intent i3 = ...;
i2 = i1; // legal
i1 = i3; // illegal
```

Figure 4.1: Type system for Android intents. Standard rules are omitted.

$$\boxed{\text{Subtyping}}$$

$$(\mathrm{ST})\qquad \frac{\forall\, k \in \mathrm{keys}(\tau_2).\ \ k \in \mathrm{keys}(\tau_1)\ \land\ \tau_1[k] = \tau_2[k]}{\tau_1 <: \tau_2}$$

$$(\mathrm{CP})\qquad \frac{\forall\, k \in \mathrm{keys}(\tau_2).\ \ k \in \mathrm{keys}(\tau_1)\ \land\ \tau_1[k] <: \tau_2[k]}{\tau_1 <_{\mathrm{copyable}} \tau_2}$$

$$\boxed{\text{Well-formedness}}$$

$$(\mathrm{OR})\qquad \frac{}{\texttt{void onReceive}(\tau\ \texttt{i})}\qquad \textit{No precondition}$$

$$\boxed{\text{Typing judgments}}$$

$$(\mathrm{SI})\qquad \frac{\begin{array}{c}\forall\ \texttt{onReceive}(b,j).\ \ \langle \texttt{sendIntent}(a,i), \texttt{onReceive}(b,j)\rangle \in\ \mathit{CCP} \\ i:\tau_i \qquad j:\tau_j \qquad \tau_i <_{\mathrm{copyable}} \tau_j\end{array}}{\mathrm{sendIntent}(a,i) : \mathit{int}}$$

$$(\mathrm{PE1})\qquad \frac{e:\tau \qquad v:\tau[k] \qquad k \in \mathrm{keys}(\tau) \qquad s:\texttt{@StringVal}(k)}{e.\mathrm{putExtra}(s,v):\tau}$$

$$(\mathrm{PE2})\qquad \frac{e:\tau \qquad k \notin \mathrm{keys}(\tau) \quad e \text{ is unaliased} \qquad s:\texttt{@StringVal}(k)}{e.\mathrm{putExtra}(s,v):\tau}$$

$$(\mathrm{GE})\qquad \frac{e:\tau \qquad k \in \mathrm{keys}(\tau) \qquad s:\texttt{@StringVal}(k)}{e.\mathrm{getExtra}(s):\tau[k]}$$

Source: Made by the authors.

The mapped types must be exactly the same; use of a subtyping requirement $\tau_1[k] <: \tau_2[k]$ instead of equality $\tau_1[k] = \tau_2[k]$ would lead to unsoundness in the presence of aliasing. The example below illustrates this problem. (On the left is the type qualifier hierarchy.)

```
            @C String c;
  @A        @Intent("akey" → @B) Intent i1;
  / \       @Intent("akey" → @A) Intent i2;
@B  @C      i2 = i1;      // illegal
            i2.putExtra("akey", c);
```

It would be incorrect to allow the assignment `i2 = i1` in this example, even though the assignment is valid according to standard object-oriented typing. In this case, the call to `putExtra` would store, in the object pointed by `i1`, a value of incorrect type at key `akey`. This happens because the references `i1` and `i2` are aliased to the same intent object.

4.2.2.2 Copyable (CP)

Copyable is a subtyping-like relationship with the weaker requirement $\tau_1[k] <: \tau_2[k]$. It may be used only when aliasing is not possible, which occurs when `onReceive` is invoked by the Android system, as explained in the (SI) rule below.

4.2.2.3 Declarations of `onReceive` (OR)

A declaration of `onReceive` always type-checks. The standard Java overriding rules do not apply to declarations of `onReceive`: the intent type of the formal parameter of `onReceive` is not restricted by the type of the parameter in the overridden declaration. This is allowable because by convention `onReceive` is never called directly but rather is only called by the Android system. The type-checker prohibits direct calls to `onReceive` methods; this constraint is omitted from Figure 4.1 for brevity.

4.2.2.4 Calls to `sendIntent` (SI)

A `sendIntent` call can be viewed as an invocation of one or more `onReceive` methods. A `sendIntent` call type-checks if its intent argument is copyable to the formal parameter of each corresponding `onReceive` method. CCP (see Section 4.1) is used to determine each `onReceive` method of a `sendIntent` call. The type comparison uses the copyable relation, not subtyping. This is sound because the Android system passes a copy of the intent argument to `onReceive`, so aliasing is not a concern.

4.2.2.5 Calls to putExtra (PE)

If the receiver of a `putExtra` call might have aliases, then the argument's type must be a subtype of the type with the specified key in the map. This prevents an alias from modifying an intent in such a way that it violates the type of another alias. For example:

```
@Intent("akey" → @Public) Intent a = new Intent();
@Intent() Intent b = a;
@Secret String hs = ...;
b.putExtra("akey", hs);       // does not type-check
a.getExtra("akey");
```

If the receiver has no aliases, then the key is permitted to be missing from the map type.

4.2.2.6 Calls to getExtra (GE)

The rule for `getExtra` is straightforward.

For both the PE and GE rules, the call (`putExtra` or `getExtra`) type-checks only if the key is a statically computable expression, according to the dataflow analysis of Section 3.1.2. For all 1,052 apps in the F-Droid repository, 93% of all keys could be statically computed.

Figure 4.2: Flow-sensitive type inference rules for intent types: the conclusion shows the type of e after the call to `putExtra`. Standard rules are omitted.

$$\frac{\begin{array}{l} e.\text{putExtra}(s,v) \quad e:\tau \quad v:\sigma \quad k \notin \text{keys}(\tau) \\ e \text{ is unaliased} \qquad\qquad s : \texttt{@StringVal}(k) \end{array}}{e:\tau\cup\{k\to\sigma\}}$$

$$\frac{\begin{array}{l} e.\text{putExtra}(s,v) \quad e:\tau\cup\{k\to\_\} \quad v:\sigma \\ e \text{ is unaliased} \qquad\qquad s : \texttt{@StringVal}(k) \end{array}}{e:\tau\cup\{k\to\sigma\}}$$

Source: Made by the authors.

4.3 Type inference

Annotations are rarely required within method bodies, because the intent type system performs flow-sensitive local type inference. Consider the following example:

```
@Intent Intent i = new Intent(); // i has type @Intent()
i.putExtra("akey", h); // i now has type @Intent("akey"→@Secret)
i.putExtra("akey", l); // i now has type @Intent("akey"→@Public)
```

Because the receiver expression of these `putExtra` calls is an unaliased local variable, its type can be refined by adding the key–type pair from the `putExtra` call. We implemented a modular aliasing analysis that determines whether an expression is unaliased.

Figure **??** shows two cases for the `putExtra` type inference rules for intent types. For both cases, the key argument of the `putExtra` call must be a statically computable expression (Section 3.1.2) and the receiver expression must be unaliased. For the first case, if the intent type of the receiver expression does not have a key–type pair with the same key passed as an argument, then the intent type is refined with the new key mapping to the type of the value passed as argument. For the second case, if the intent type already has a key–type pair with the same key, then the type in this key–type pair is replaced by the type of the value passed as an argument. A further standard condition (omitted from Figure **??** for brevity) is that the new intent type must be a subtype of the declared type.

4.4 Example

Recall the example of Figure 2.2. A noninterference type-checker would report false-positive warnings on lines 14–16 because the type system is unable to deduce that all extra data from the corresponding intent is of type `Public`. A developer can express this intended design by manually annotating the method `WordTranslator.getIntent` (inherited from class `Activity`):

```
@Override
@Intent("source" → @Public, "target" → @Public, "word" → @Public)
Intent getIntent() { return super.getIntent(); }
```

The `startActivity(i)` statement on line 7 still type-checks after this change because the type-checker refines the type of `i` to `@Intent("source" → @Public, "target" → @Public, "word" → @Public)` as a result of the `putExtra` calls on lines 4–6. Our implementation only has type inference for sent intents, therefore receiving intents that retrieve extras require manual annotations.

In the method `DictionaryMain.translateWord()`, the copyable typing rule enforces that the intent variable `i` has a compatible type with the return type of `WordTranslator.getIntent()`.

By extending the noninterference type system with our intent type system and manually adding the correct annotations to the return type of `WordTranslator.getIntent()`, the Aard Dictionary example type-checks and the developer is assured that the program does not contain security vulnerabilities that could leak private data. Note that any developer-written annotations in the program are checked, not trusted.

5

Improving a downstream analysis

We evaluated our work in two ways. First, this chapter reports how much our reflection and intent analyses improve the precision of a downstream analysis, which is their entire purpose. Then, it presents a formal proof of both analyses. Second, Chapter 6 measures how well our type inference rules reduce the programmer annotation burden.

The purpose of resolving reflection and intents is to improve the precision of a downstream analysis. Section 5.1.1 measures the improvement in precision, and Section 5.1.2 shows the programmer effort required to achieve the improved precision. Section 5.1.3 shows how our analyses can also improve other downstream analyses. Section 5.2 presents a formal analysis.

5.1 Subject programs and downstream analysis

We used open-source apps from the F-Droid repository (F-DROID, 2010) to evaluate our approach. F-Droid contains 1,052 apps that have an average size of 9,237 Lines of code (LOC)[1] and do not use third-party libraries.

415 out of 1,052 F-Droid apps (39%) use reflection, and each app that uses reflection has on average 11 reflective method or constructor invocations. 726 out of 1,052 F-Droid apps (69%) use intents with extra data, and each app that uses intents with extra data has on average 24 calls to `putExtra` or `getExtra`. 254 out of 1,052 F-Droid apps (24%) use both reflective calls *and* intents with extra data. These numbers support our motivation to pursue static analysis of reflection and intents.

We aimed to select subject apps of typical complexity. We excluded excessively simple apps: those with less than 2,000 LOC or that did not have at least one call to `putExtra`, `getExtra`, and `Method.invoke`. We also excluded excessively complex apps: those with more than 15,000 LOC or that used more than five Android permissions, which is the average number of permissions used by an F-Droid app. Overall, 40 apps satisfied our requirements, and we randomly sampled 10 apps, which are listed in Table 5.1. Each of the 10 apps contains on average 5,261 LOC, 3 reflective method or constructor invocations,[2] and 37 calls to `putExtra` or `getExtra`.

[1] Non-comment, non-blank lines of code, as reported by David A. Wheeler's SLOCCount. See `http://www.dwheeler.com/sloccount/`.

[2] This is smaller than the F-Droid average: most uses of reflection in F-Droid appear in a few huge apps (>500

Table 5.1: Selected subject apps from the F-Droid repository. The number of reflective invocatons is given for `Method`s and `Constructor`s, and intent uses count the number of `putExtra` and `getExtra` calls. The last three columns show the annotation overhead for the technique IFC+INT+RR. The column IFC shows the number of `@Source` and `@Sink` information flow annotations. The column refl shows the number of `@MethodVal` and `@ClassBound` annotations (no `@ClassVal` annotations were required). The column int shows the number of `@Intent` annotations.

App	LOC	Reflection		Intent uses		# of annotations		
		meth	cons	put	get	IFC	refl	int
AbstractArt	4,488	1	0	1	1	317	0	1
arXiv	3,643	14	0	70	17	130	0	13
Bluez IME	4,523	4	2	124	42	285	0	16
ComicsReader	6.612	6	0	1	2	381	1	6
MultiPicture	7,496	1	0	17	12	511	0	17
PrimitiveFTP	4,026	2	0	1	1	321	0	1
RemoteKeyboard	5,723	1	0	3	4	580	0	4
SuperGenPass	2,125	1	0	15	14	181	0	8
VimTouch	8,881	1	0	7	6	2,424	2	7
VLCRemote	5,097	1	0	12	21	453	0	22
Total	52,614	32	2	251	120	5,583	3	95

Our evaluation uses three downstream analyses. Sections 5.1.1–5.1.2 discuss the Information Flow Checker (IFC); Section 5.1.3 briefly discusses the other two case studies. IFC is a type system and corresponding type-checker that prevents unintended leakage of sensitive data from an application (ERNST et al., 2014). Given a program and an information-flow policy (a high-level specification of information flow, expressed as source–sink pairs), IFC guarantees that no other information flows occur in the program. IFC is sound: it issues a warning if the information flow type of any variable or expression does not appear in the information-flow policy. IFC is also conservative: if it issues a warning, then the program might or might not misbehave at run time.

We evaluated the effectiveness of our techniques by answering the following two research questions.

5.1.1 How much do our reflection and intent analyses improve the precision of IFC?

We measured the precision and recall of IFC's static estimate of possible information flows. To compute precision and recall, we manually determined the ground truth: the actual number of flows that could occur at run time in an app. This enormous manual effort is the reason we did not run the experiments on all 1,052 F-Droid apps. It would be easy to run our analyses on all the apps, but doing so would not indicate whether our analyses were useful. Precision is the number of ground-truth flows, divided by the total number of flows reported by the analysis. Recall is the number of real flows reported by the analysis, divided by the total number of ground-truth flows. We confirmed that IFC has 100% recall both with and without the

kLoC) that contain hundreds of reflective calls.

reflection and intent analyses, i.e., IFC is sound and misses no real flows.

To evaluate this research question, we compared the precision of the following techniques.

IFC-unsound makes optimistic assumptions about every reflective and intent-related call — it assumes that reflective and intent-related calls do not trigger new information flows. Its recall is only 95% — it unsoundly misses 5% of the information flows in the apps, which makes it unacceptable for use in the security domain. Its precision was 100%, for this set of apps.

IFC treats reflection and intents conservatively. Data in an intent may be from any source and may flow to any sink. Data used as an argument to a reflective invocation may flow to any sink, and data returned from a reflective invocation may be from any source. In the absence of reflection and intents, IFC is an effective analysis with high precision, as shown by IFC-unsound. However, for our subject programs, which use reflection and intents, IFC's precision is just 0.24%.

IFC+Reflection Resolution (RR) augments IFC with reflection resolution and can therefore treat data that is used in reflection precisely when the reflection can be resolved. Data in intents, however, is treated conservatively. Since all apps send intents, which may trigger the use of any permissions, reflection resolution alone does not help; the average precision remains 0.24%. In a (non-Android) program that does not use intents, IFC+RR would outperform IFC.

IFC+Intent Analysis (INT) augments IFC with intent analysis. It reports precise information flows for method calls involving intents. Differently from intent invocations, reflective calls are only allowed to use permissions listed in the app's manifest. Therefore, data passed to a reflective invocation is treated as flowing to any sink the app may access. Similarly, data returned from a reflective invocation is treated as if it could have come from all sources listed in the manifest. However, since Epicc generates CCP and unsoundly assumes that reflective calls do not invoke `sendIntent` methods, IFC+INT must issue a warning any time a method is reflectively invoked. For each such warning, the developer must manually verify that the reflective method does not invoke `sendIntent`. The average precision is 53%.

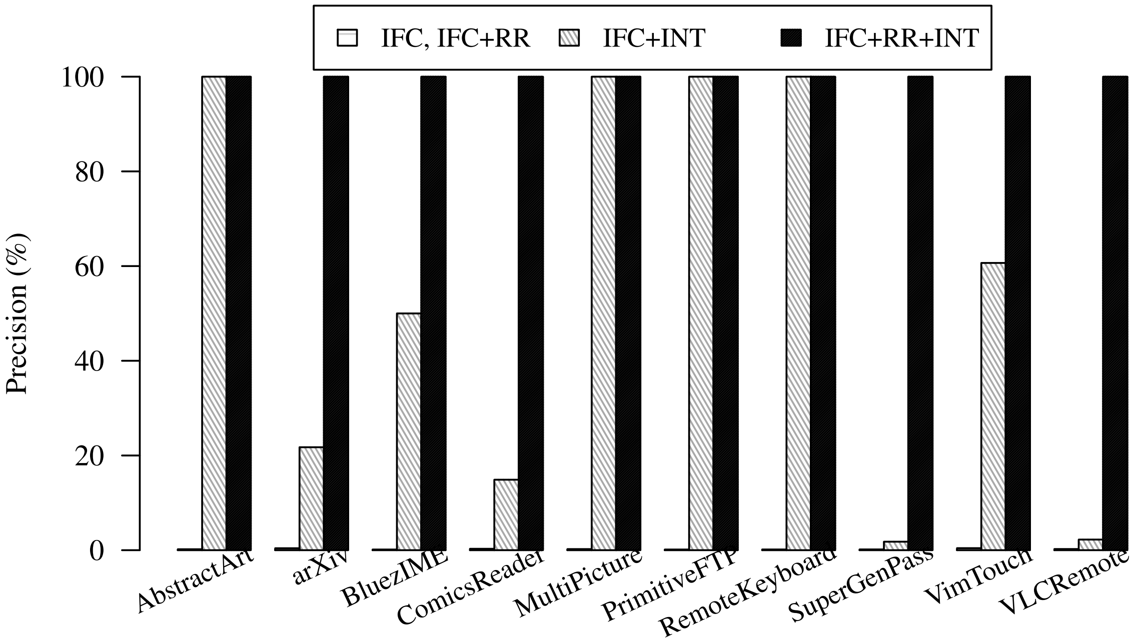
IFC+INT+RR augments IFC with both reflection resolution and intent analysis. When reflection resolution cannot resolve a method or when it resolves a method to `sendIntent`, it still issues a warning. The precision is 100% for each of the 10 randomly-chosen apps, but might be smaller for other apps.

Figure 5.1 plots the precision for the sound techniques.[3] Being the most basic technique, IFC has the worst precision among all approaches. At the other extreme, IFC+INT+RR has the highest precision for all cases. This occurs because this technique provides custom support for both reflective calls and intents. Such high precision is obtained at the expense of adding annotations in the code. Section 5.1.2 discusses the overhead associated with the annotation process.

For `AbstractArt`, `MultiPicture`, `PrimitiveFTP`, and `RemoteKeyboard` IFC+INT has perfect

[3] All sound techniques achieve 100% recall.

Figure 5.1: Comparison of precision among techniques.



Source: Made by the authors.

precision because these apps use reflection for control flow but not data flow — data returned from reflective calls is not sent to a sensitive sink and no sensitive information is passed as an argument to a reflective method call. For the other 6 apps, IFC+INT is more precise than IFC, but still reports flows that cannot happen at run time. For these apps, the reflection resolver is needed to reach 100% precision as reported by IFC+INT+RR. The results confirm that both techniques, reflection resolution and intent analysis, are necessary and that they are complementary and synergistic.

The 10 apps in Figure 5.1 use both reflection and intents with extras, like 24% of all apps in the F-Droid repository. For apps that use just one of the features, IFC+RR or IFC+INT would achieve the same precision as IFC+INT+RR.

All uses of reflection could be resolved except for one in the `RemoteKeyboard` app. For that case, the reflection resolver could determine the name of the invoked method (`createShell`) and the number of parameters (0), but the class name is obtained from preferences that the user can edit at run time. However, this method is not `sendIntent`, its returned object is not sent to any sink, and it takes no parameters; therefore, treating that call conservatively did not decrease the precision of IFC+INT+RR.

For the 10 apps IFC-unsound takes on average 4.5 minutes to run. IFC+RR+INT takes on average 6.7 minutes to run.

We attempted to compare our approach with IccTA (LI et al., 2015). IccTA crashed when run on 1 of the 10 apps. For the other 9 apps, IccTA outputted some static analysis data, but no data regarding information flows. We contacted the IccTA's first author, Li Li, about these issues but didn't hear back from him.

5.1.2 What is the annotation overhead for programmers?

Developers must write source code annotations in order to use our analyses. This is not extra work, since the alternative would be to spend time reviewing false-positive warnings.

Table 5.1 shows the annotations required to type-check each app. This table also includes the number of manual annotations required by IFC. These annotations are required by IFC regardless if our analyses are being used. Less than 2% as many annotations are required due to reflection and intents, compared to annotations related to information flow (the downstream analysis). If the programmer omits an annotation, or writes one that is inconsistent with the source code or with other annotations, then the analyses issue user-friendly warnings that pinpoint and explain the type inconsistency. The average time to add each annotation related to our analyses was roughly one minute. Thus, the annotation overhead is small in absolute and relative terms, especially considering the significant improvements in precision due to reflection and intent analysis.

Part of the need for annotations is because the downstream analysis is a modular analysis — a type-checker that verifies programmer-written types. If the downstream analysis were a whole-program analysis such as pointer analysis, type inference, or abstract interpretation, these would not be necessary. Other annotations are needed to express facts that no static analysis can infer; in these cases, human intervention is unavoidable.

5.1.3 Precision improvements for other downstream analyses

We demonstrated the generality of our approach by integrating our reflection and intent analyses with two other downstream analyses. The Nullness Checker (PAPI et al., 2008) verifies the absence of null pointer dereferences: if the Nullness Checker issues no errors for a given program, then that program is guaranteed to not throw a `NullPointerException` at runtime. The Interning Checker (PAPI et al., 2008) verifies equality tests: if the Interning Checker issues no errors for a given program, then all reference equality tests (i.e., `==`) are over canonicalized data, and thus are consistent with `.equals()`.

These analyses suffer false positives due to reflection and intents. Consider the Nullness Checker. Its library annotations must mark the return type of `Method.invoke` as `@Nullable`, for soundness. The reflection analysis can determine that some calls to `invoke` return a non-null value, and thus it eliminates false positives in the nullness analysis.

Reflection resolution improved the precision of the Nullness Checker for 3 of the 10 apps. There were no reference equality tests over values returned by a reflective method invocation, and therefore reflection resolution did not improve the precision of the Interning Checker for these 10 apps.

The intent analysis improved the precision of the Interning Checker for 2 of the 10 apps.

The intent analysis does not improve the precision of the Nullness Checker for any app, because `getExtra` can return `null` if a key does not exist in an intent map. The intent type system does not guarantee the existence of a key in an intent map — only that if it exists, it has a certain type.

5.2 Formal analysis

Our implementation works on Java code: it does not analyze native calls. For efficiency, it relies on trusted annotations for system libraries. These are standard limitations of a static analysis. Section 4.1 notes other limitations regarding the estimation of component communication patterns.

Modulo these limitations, our analysis is sound. That is, if a program type-checks, then the type of any expression is a sound estimate of its possible run-time values.

For reflection, this means that the value for a `Class` or `Method` expression is contained within the set of possible values in its type, and likewise for array lengths.

For intents, this means that if an expression has a type with an intent key–type pair, then at run time the expression's value is an intent whose extra data maps the key to a value of that type, or the key does not appear in the map.

Equally importantly, the resolution preserves any soundness property for a downstream analysis. If the downstream analysis is sound when using the conservative library annotations, then it remains sound when using more precise summaries supplied by the reflection and intent resolvers.

It is possible to state formal type-correctness, progress, and preservation theorems for our type systems. The theorems are standard and their proofs would be straightforward.

5.3 Threats to validity

Our system is modular, except that a whole-program analysis computes inter-component communication. We assume that text/XML files don't change from analysis time to execution time. Aside from that we have the following threats to validity:

- Experiments. We manually determined the ground truth of information flows for each app. This enormous manual effort is the reason we did not run the experiments on all 1,052 F-Droid apps. It would be easy to run our analyses on all the apps, but doing so would not indicate whether our analyses were useful.
- Native calls. Our analyses assume that Android apps use no native calls, a standard limitation shared by Epicc as well.
- Manual annotations. We tried to reduce the number of manual annotations required per app to a minimum — one manual annotation required per 2kLOC— but this overhead still exists and we cannot say that our analysis is fully automatic.

6

Evaluation of type inference

As shown in Section 5.1.2, programmers have to write very few annotations to aid the reflection and intent analysis. This section explains why, by evaluating our type inference rules.

6.1 Reflection resolution

In addition to the 10 subject apps of Section 5.1, we arbitrarily selected 25 apps from F-Droid that use reflection. Using the entire set of 35 apps, we evaluated the reflection resolution by answering the following three research questions.

6.1.1 How is reflection used in practice?

The 35 apps contain 142 invocations of reflective methods or constructors. 81% are used to provide backward compatibility, 6% access a non-public API, and 13% are for other use cases.

6.1.2 How often can reflection be resolved at compile time?

Our reflection resolution resolved 93% of instances of reflective method or constructor invocations. It failed on the other 7% because the reflectively invoked method or constructor cannot be determined statically by any analysis. As an example, the `RemoteKeyboard` app uses reflection for extensibility and duck typing: the user can configure the class name for a shell implementation, and the app reflectively invokes a factory method on this class. Moreover, these shell implementations do not have a common interface that defines the factory method, rendering static reflection resolution impracticable.

6.1.3 How effective is type inference for reflection resolution?

To enable modular reflection resolution, a developer may have to write type annotations in a program. We evaluated the effectiveness of our type inference (see Section 3.1.2) that reduces the annotation burden. Specifically, we determined how many instances of reflection can be resolved without any developer-written annotation and whether the remaining instances require stronger inference or developer-written annotations.

For 52% of reflective invocations, our intra-procedural type inference (Section 3.1.2) enabled fully automated reflection resolution. This means that our type inference determined the exact method that is reflectively invoked without requiring a single annotation.

For 41% of reflective invocations, our inter-procedural, intra-class type inference determined the exact method that is reflectively invoked. A common example is the initialization of a private field of type `Class` or `Method`. These fields are only assigned once but are initialized within a method that provides exception handling. Another example is the use of a helper method that manipulates `String`s and returns an object of type `Method` that is used within the class.

We also implemented an inter-class inference, but it did not improve the results for the selected apps, beyond the intra-class analysis results.

The other 7% of reflection invocations cannot be resolved by any static analysis (for an example, see Section 6.1.2). Code inspection and developer intervention are required in those cases.

Table 5.1 gives the number of developer-written annotations that were required. Recall that all annotations in an app are checked, not trusted. Thus, use of developer-supplied annotations does not compromise the soundness of our approach.

6.1.4 Bug detection

Our reflection resolver revealed a bug in the `arXiv` app. The reflection resolver reported an unresolvable method even though it precisely inferred the class name, method name, and the number of parameters. The bug was a misspelled method name, and it prevented a menu from being updated. A menu was supposed to be updated every time a certain button was pressed. This bug prevented such update, since the name of the method that performs the update was misspelled. The developer confirmed the bug via email.

6.2 Intent type inference

Section 4.3 introduced rules to refine the type of an intent, which reduce the number of developer-written annotations required in a program. This section evaluates how effective they are in practice. We only implemented type refinement for sent intents. A limitation of our implementation is that declarations of `onReceive` methods must have a precise intent type, so `sendIntent` calls can be type-checked against these declarations. Therefore, we evaluated type refinement of sent intents (68% of all intents). We defer inferring intent types on declarations of `onReceive` methods to future work. We considered only intents with extras (51% of all sent intents), as an empty intent requires no developer-written annotation.

To measure the effectiveness of the intent type inference (Section 4.3), we used a similar approach as when measuring the reflection resolution type inference: we determined the number of sent intents with extras that required no annotations and compared it with the overall number of sent intents with extras.

For 67% of the cases, our intra-procedural inference determined that the sent intent had no aliases and precisely inferred the type of the sent intent. For those cases, developer-written annotations are not necessary.

For 21% of the cases, our inter-procedural inference correctly infers the type of the sent intent.

For 12% of the cases, the sent intent was stored in a field. Our alias analysis (Section 4.3) treated such intents as possibly-aliased, so the intent type cannot be refined using the `putExtra` rule. Manual annotations are required for these cases.

The 10 apps require a total of 7 developer-written annotations for sent intents with extras. Without intent type inference, the apps would have needed an additional 52 developer-written annotations in order to type-check.[1] This result shows that intent type inference greatly reduces the annotation burden.

[1] $88\% \neq 52/(7+52)$ because some developer-written annotations solve multiple cases where intent type inference does not succeed.

7

Related work

7.1 Reflection

The most common approach for improving precision of a static analysis in the presence of reflection is profiling from an observed set of executions, assuming that the observed program exercises all possible behaviors. Livshits (LIVSHITS; WHALEY; LAM, 2005) requires user annotations or dynamic information from casts to estimate reflection targets as part of static call graph construction. Tatsubori (TATSUBORI, 2004) earlier built a system with similar qualities. TamiFlex (BODDEN et al., 2011) performs unsound dynamic analysis of reflection and dynamic class loading. It replaces uses of reflection by standard method calls, and supplies the modified call graphs to existing static-analysis tools. In other words, an unsound analysis can be built on top of TamiFlex, just as a sound analysis can be built on top our our work. An example is that Averroes (ALI; LHOTáK, 2013) can use TamiFlex when building call graphs, to unsoundly improve precision over its conservative defaults. All of these approaches that use dynamic information are unsound. By contrast, our approach is sound: it makes conservative assumptions about any occurrence of reflection that it cannot handle.

In some special cases, reflection can be resolved based on assumptions about the run-time execution context. For example, Zhang's GUI error detection tool (ZHANG; LÜ; ERNST, 2012) builds reflection-aware call graphs for Android applications, enabling it to find more GUI errors than without. However, it only handles a particular scenario — it converts reflective calls into explicit constructor invocations based on the contents of configuration files at compile time. This approach is sound if the same configuration files will be installed at run time as at analysis time. This is the same assumption made by Epicc (OCTEAU et al., 2013) to handle inter-component communication, which our system uses.

A few static analyses partially handle reflection. Javari (TSCHANTZ; ERNST, 2005) introduces a new API to invoke reflection that does a single dynamic check of the method signature rather than of the object. Programs using that API can be soundly type-checked. Our approach could eliminate that special API and the run-time check. Li et al.(LI et al., 2014) developed an unsound self-inferencing reflection resolution to improve the precision of a pointer analysis for Java programs. They additionally analyzed how reflection is used in open-source

Java applications. In contrast, our approach is sound and our evaluation focuses on the use of reflection in Android apps.

7.2 Android

We evaluated our reflection and intent analyses in the context of detecting and preventing malicious behavior in mobile apps (FUCHS; CHAUDHURI; FOSTER, 2009; ENCK et al., 2010; HORNYACK et al., 2011; EGELE et al., 2011; GRACE et al., 2012; CHIN et al., 2011; GIBLER et al., 2012; MANN; STAROSTIN, 2012; GRACE et al., 2012; ZHOU et al., 2012; YAN; YIN, 2012; XU; SAÏDI; ANDERSON, 2012). We discuss some closely related work.

SCanDroid (FUCHS; CHAUDHURI; FOSTER, 2009) applies data flow analysis to check security properties in Android apps. It analyzes intra-component and inter-component information flows for vulnerabilities. The analysis cannot handle interactions between apps and provides limited support to handle intent extras, making no distinction between the flows of permissions that result from the entries of an intent. Several other techniques came after it (EGELE et al., 2011; GRACE et al., 2012; CHIN et al., 2011; GIBLER et al., 2012; MANN; STAROSTIN, 2012; GRACE et al., 2012; ZHOU et al., 2012; ARZT et al., 2014), improving precision and recall of reported warnings. However, to the best of our knowledge, no later technique has focused on handling the important aspect of data encapsulation in intents. Our technique is complementary to push-button static analysis techniques such as SCanDroid: our analysis requires a small number of annotations from the developer but requires less examination of false positives and provides stronger guarantees. It preserves soundness, achieves good precision, and remains easy to use.

FlowDroid (ARZT et al., 2014) is a technique that performs taint analysis on Android apps with the goal of finding security vulnerabilities. FlowDroid does not support Android's implicit intents nor reflection. In experiments, the tool achieved 83% precision and 93% recall for apps containing different types of vulnerabilities.

Our implementation currently relies on Epicc (OCTEAU et al., 2013) to approximate the set of component pairs that actually communicate. See Section 4.1 for a discussion.

Our implementation has been publicly available since December 12, 2013. In forthcoming work, IccTA (LI et al., 2015) adopts a similar approach that performs static taint analysis in the presence of inter-component communication. IccTA's reflection resolution is much more limited than ours: it only processes string constants. Although IccTA is applied to taint analysis, IccTA is neither sound nor complete; by contrast to our work, it provides no security guarantees to its user and is not applicable in the context of high-assurance app stores (ERNST et al., 2014). Even if the analysis flaws were addressed, IccTA would remain vulnerable because its taint model uses an insufficient set of sensitive sources and sinks. Another difference is the evaluation: we measured the precision and recall of our information-flow analysis on real Android apps and achieved 100% precision and recall, but IccTA was evaluated on 22 examples hand-crafted by its authors, where it achieved 96% precision and recall.

7.3 Other

Xiao et al. (XIAO et al., 2012) proposed a semi-automatic approach to analyze TouchDevelop mobile app scripts for privacy. Their workflow is similar to ours: users annotate APIs and code, and the analyzer uses a dataflow analysis to check conformance of inferred flows against a specification of the app. However, their static analysis does not handle implicit control flows.

Google's Android NDK (NDK, 2008) allows parts of an app to be implemented using native-code languages such as C and C++. Our toolset does no analysis of native code: summaries for native methods are trusted. The Checker Framework, on which our implementation is built, treats unannotated methods conservatively.

Our work has some similarities to call graph construction in object-oriented programs (TIP; PALSBERG, 2000; LE; LHOTÁK; HENDREN, 2005). Dynamic dispatching can be viewed as an implicit control flow mechanism, much as Java reflection and Android intents can. Most call graph construction algorithms do whole-program pointer analysis. Our approach is modular but relies on user annotations. A whole-program type inference or pointer analysis could eliminate the need for programmers to write annotations.

8 Conclusions

We have presented novel analyses for two programming paradigms — Java reflection and Android intents — that are useful to programmers but challenging for static analysis. Our analyses statically resolve reflection targets and intent payloads. Though sound and conservative, they achieve high precision in practice, as confirmed by experiments on real-world Android apps. Our experiments also confirmed that the overhead to use our analyses is small in relative and absolute terms. Our implementations are publicly available as open source (FRAMEWORK, 2007), and they can be integrated with an arbitrary downstream analysis to improve its precision.

As future work, we would like to implement inference for intent types on declarations of `onReceive` methods. This would greatly reduce the number of manual annotations required by the Intent analysis. We would also like to integrate our approach with IC3, Epicc's successor, in order to achieve an even better precision. We would also like to perform experiments on a larger set of apps, and with others downstream analyses.

References

ALI, K.; LHOTáK, O. Averroes: whole-program analysis without the whole program. In: ECOOP 2013 — OBJECT-ORIENTED PROGRAMMING, 27TH EUROPEAN CONFERENCE, Montpellier, France. **Proceedings...** [S.l.: s.n.], 2013. p.378–400.

APKParser. Available in: <`https://code.google.com/p/xml-apk-parser/`>. Accessed on: 20 Mar. 2014.

ARZT, S. et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: PLDI 2014, PROCEEDINGS OF THE ACM SIGPLAN 2014 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, Edinburgh, UK. **Proceedings...** [S.l.: s.n.], 2014. p.259–269.

BLOCH, J. **Effective Java Programming Language Guide**. Boston, MA: Addison Wesley, 2001.

BODDEN, E. et al. Taming Reflection: aiding static analysis in the presence of reflection and custom class loaders. In: ICSE'11, PROCEEDINGS OF THE 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Waikiki, Hawaii, USA. **Proceedings...** [S.l.: s.n.], 2011. p.241–250.

CHIN, E. et al. Analyzing inter-application communication in Android. In: INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, 9., Bethesda, MD, USA. **Proceedings...** [S.l.: s.n.], 2011. p.239–252.

DIETL, W. et al. Building and using pluggable type-checkers. In: ICSE'11, PROCEEDINGS OF THE 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Waikiki, Hawaii, USA. **Proceedings...** [S.l.: s.n.], 2011. p.681–690.

EGELE, M. et al. PiOS: detecting privacy leaks in iOS applications. In: ANNUAL SYMPOSIUM ON NETWORK AND DISTRIBUTED SYSTEM SECURITY, 18., San Diego, CA, USA. **Proceedings...** [S.l.: s.n.], 2011.

ENCK, W. et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: USENIX 9TH SYMPOSIUM ON OS DESIGN AND IMPLEMENTATION, Vancouver, BC, Canada. **Proceedings...** [S.l.: s.n.], 2010.

ERNST, M. D. et al. Collaborative verification of information flow for a high-assurance app store. In: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY (CCS), 21., Scottsdale, AZ, USA. **Proceedings...** [S.l.: s.n.], 2014. p.1092–1104.

F-DROID. **F-Droid**: free and open source android app repository. Available in: <`http://f-droid.org`>. Accessed on: 20 Mar. 2015.

FRAMEWORK, C. **Checker Framework website**. Available in: <`http://types.cs.washington.edu/checker-framework/`>. Accessed on: 20 Mar. 2015.

FUCHS, A. P.; CHAUDHURI, A.; FOSTER, J. S. **SCanDroid**: automated security certification of Android applications. [S.l.]: University of Maryland, 2009. (CS-TR-4991).

GIBLER, C. et al. AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale. In: INTERNATIONAL CONFERENCE ON TRUST AND TRUSTWORTHY COMPUTING, 5., Vienna, Austria. **Proceedings...** [S.l.: s.n.], 2012. p.291–307.

GOOGLE. **Android Fundamentals**. Available in: <http://developer.android.com/guide/components/fundamentals.html>. Accessed on: 20 Mar. 2015.

GRACE, M. et al. Systematic detection of capability leaks in stock Android smartphones. In: ANNUAL SYMPOSIUM ON NETWORK AND DISTRIBUTED SYSTEM SECURITY, 18., San Diego, CA, USA. **Proceedings...** [S.l.: s.n.], 2012.

GRACE, M. et al. RiskRanker: scalable and accurate zero-day Android malware detection. In: INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, 10., Low Wood Bay, UK. **Proceedings...** [S.l.: s.n.], 2012. p.281–294.

HORNYACK, P. et al. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY (CCS), 18., Chicago, IL, USA. **Proceedings...** [S.l.: s.n.], 2011. p.639–652.

KIM, C. H. P. et al. SPLat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In: JOINT MEETING OF THE EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE'13, SAINT PETERSBURG, RUSSIAN FEDERATION, AUGUST 18-26, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.257–267.

LE, A.; LHOTÁK, O.; HENDREN, L. Using inter-procedural side-effect information in JIT optimizations. In: COMPILER CONSTRUCTION: 14TH INTERNATIONAL CONFERENCE, CC 2005, Edinburgh, Scotland. **Proceedings...** [S.l.: s.n.], 2005. p.287–304.

LI, L. et al. IccTA: detecting inter-component privacy leaks in android apps. In: IEEE/ACM INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE 2015, FLORENCE, ITALY, MAY 16-24, 2015, VOLUME 1, 37. **Proceedings...** IEEE, 2015. p.280–291.

LI, Y. et al. Self-inferencing Reflection Resolution for Java. In: ECOOP 2014 — OBJECT-ORIENTED PROGRAMMING, 28TH EUROPEAN CONFERENCE, Uppsala, Sweden. **Proceedings...** [S.l.: s.n.], 2014. p.27–53.

LIVSHITS, B.; WHALEY, J.; LAM, M. S. Reflection analysis for Java. In: THIRD ASIAN SYMPOSIUM ON PROGRAMMING LANGUAGES AND SYSTEMS, Tsukuba, Japan. **Proceedings...** [S.l.: s.n.], 2005. p.139–160.

MANN, C.; STAROSTIN, A. A framework for static detection of privacy leaks in Android applications. In: ACM SYMPOSIUM ON APPLIED COMPUTING, 2012., Trento, Italy. **Proceedings...** [S.l.: s.n.], 2012. p.1457–1462.

NDK, A. **Android NDK**. Available in: <http://developer.android.com/tools/sdk/ndk/index.html/>. Accessed on: 20 Mar. 2015.

OCTEAU, D. et al. Effective inter-component communication mapping in Android with Epicc: an essential step towards holistic security analysis. In: USENIX SECURITY SYMPOSIUM, 22., Washington, DC, USA. **Proceedings...** [S.l.: s.n.], 2013. p.543–558.

OCTEAU, D. et al. Composite constant propagation: application to Android inter-component communication analysis. In: ICSE'15, PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Florance, Italy. **Proceedings...** [S.l.: s.n.], 2015.

OCTEAU, D.; JHA, S.; MCDANIEL, P. Retargeting Android applications to Java bytecode. In: FSE 2012, PROCEEDINGS OF THE ACM SIGSOFT 20TH SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, Cary, NC, USA. **Proceedings...** [S.l.: s.n.], 2012. p.6:1–6:11.

PAPI, M. M. et al. Practical pluggable types for Java. In: ISSTA 2008, PROCEEDINGS OF THE 2008 INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, Seattle, WA, USA. **Proceedings...** [S.l.: s.n.], 2008. p.201–212.

SMITH, B. C. **Procedural Reflection in Programming Languages**. Cambridge, MA: MIT Laboratory for Computer Science, 1982. (MIT-LCS-TR-272).

TATSUBORI, M. Living with Reflection: towards coexistence of program transformation by middleware and reflection in Java applications. In: JSSST WORKSHOP ON PROGRAMMING AND PROGRAMMING LANGUAGES (PPL2004), 6., Gamagohri, Aichi, Japan. **Proceedings...** [S.l.: s.n.], 2004.

TIP, F.; PALSBERG, J. Scalable propagation-based call graph construction algorithms. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA 2000), Minneapolis, MN, USA. **Proceedings...** [S.l.: s.n.], 2000. p.281–293.

TSCHANTZ, M. S.; ERNST, M. D. Javari: adding reference immutability to Java. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA 2005), San Diego, CA, USA. **Proceedings...** [S.l.: s.n.], 2005. p.211–230.

VOLPANO, D. M.; SMITH, G. A type-based approach to program security. In: TAPSOFT '97: THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT, 7TH INTERNATIONAL JOINT CONFERENCE CAAP/FASE, Lille, France. **Proceedings...** [S.l.: s.n.], 1997. p.607–621.

XIAO, X. et al. User-aware privacy control via extended static-information-flow analysis. In: ASE 2012: PROCEEDINGS OF THE 27TH ANNUAL INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, Essen, Germany. **Proceedings...** [S.l.: s.n.], 2012. p.80–89.

XU, R.; SAÏDI, H.; ANDERSON, R. Aurasium: practical policy enforcement for Android applications. In: USENIX SECURITY SYMPOSIUM, 21., Bellevue, WA, USA. **Proceedings...** [S.l.: s.n.], 2012.

YAN, L. K.; YIN, H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: USENIX SECURITY SYMPOSIUM, 21., Bellevue, WA, USA. **Proceedings...** [S.l.: s.n.], 2012.

ZHANG, S.; LÜ, H.; ERNST, M. D. Finding errors in multithreaded GUI applications. In: ISSTA 2012, PROCEEDINGS OF THE 2012 INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, Minneapolis, MN, USA. **Proceedings. . .** [S.l.: s.n.], 2012. p.243–253.

ZHOU, Y. et al. Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets. In: ANNUAL SYMPOSIUM ON NETWORK AND DISTRIBUTED SYSTEM SECURITY, 18., San Diego, CA, USA. **Proceedings. . .** [S.l.: s.n.], 2012.