The Pennsylvania State University
The Graduate School

ANALYSIS OF INTER-COMPONENT COMMUNICATION IN MOBILE APPLICATIONS THROUGH RETARGETING

A Dissertation in
Computer Science and Engineering
by
Damien Octeau

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2014

The dissertation of Damien Octeau was reviewed and approved* by the following:

Patrick D. McDaniel
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Trent R. Jaeger
Professor of Computer Science and Engineering

John Hannan
Associate Professor of Computer Science and Engineering

Stephen G. Simpson
Professor of Mathematics

Somesh Jha
Professor of Computer Sciences, University of Wisconsin
Special Member

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Smart phones and tablets are becoming ubiquitous. The functionality of these smart devices can be extended through the use of third-party applications. These applications rely on sophisticated communication mechanisms to share functionality and data. This information sharing gives rise to emerging threats. Unfortunately, the distribution model of mobile applications offers little to no security guarantees. Security decisions are delegated to the end user, who often has insufficient information to make informed choices. In particular, users have no visibility over inter-application communication, which forces them to blindly trust the applications they use not to misbehave.

In this work, we review our first steps in analyzing Inter-Component Communication (ICC) in mobile applications. Our method consists in first converting applications to a well-known representation to enable subsequent analysis. The result of that effort is a retargeting tool called Dare that can perform this conversion with very high accuracy and performance. The second part of our work consists in a formal model of ICC. We reduce the study of ICC to an instance of an Interprocedural Distributive Environment problem. After doing so, we are able to find a solution to the ICC problem using well-studied algorithms. We subsequently revise this ICC model by generalizing the inference of ICC specifications to a novel type of constant propagation for complex objects. Such constant propagation problems are specified using a propagation language. Using this approach, we are able to model all of ICC in the Android platform. We end this work by showing a first visualization-based study of inter-component communication. This methodology to determine and represent ICC links enables all kinds of analyses that need to consider multiple application components or multiple applications.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank all those without whom I could not have completed this dissertation. First, I wish to express my deepest gratitude towards my advisor Dr. Patrick McDaniel. He has always given me the right amount of guidance and freedom that I needed to perform my research. When I first arrived and needed time to learn basic skills he let me get up to speed without pressuring me. He also taught me to write research papers and even had the patience to rewrite parts of my first papers. In the later year of graduate school he gave me the freedom to implement my ideas without asking for immediate results. I cannot think of a way that would have stimulated my creativity more. He was always supportive when I needed it, including for simple things like reassuring me before my first conference talks.

The work I present in this thesis would not have been possible without Dr. Somesh Jha. He has provided me with very precious insight on issues related to program analysis. He has invited me to the University of Wisconsin twice and every time I have felt like I learned something new and that we made significant progress in our project. Our long email conversations also made my understanding of basic principles of program analysis deeper.

I would also like to thank the members of the SIIS Lab. When I started as a Master's student, Will Enck provided me with very valuable advice on technical aspects of my research. I received assistance from Tom Moyer when I had workstation or administrative issues. Stephen McLaughlin provided me with very useful feedback on paper drafts. I learned a lot about dedication and hard work from Steve, Hayawardh Vijayakumar and Sandra Rueda since they were consistently in the office late when I also happened to be there. They also were always there for interesting conversations, jokes or other bon mots. I also thank Matt Dering for his help in implementing parts of the last project described in this dissertation.

I thank Somesh's group for their very warm welcome both times I visited Somesh. In particular I thank Drew Davidson, Matt Fredrikson and Bill Harris for their constructive reviews of paper drafts. I am also thankful to Daniel Luchaup for his technical contributions to my research.

I am very grateful for the support I have received from my friends and family in France and in the United States. Despite my long stay abroad, they still make me feel

like I never left whenever I come back. I wish to thank my brother Vivien for inspiring me to find a path in the sciences. I am also deeply grateful to have parents who consistently told me that anything is possible if I persevere enough. Finally, I have spent the second half of my graduate studies in the company of my girlfriend Başak. I am very fortunate to have been able to count on her love and support in the good times and the bad times during this journey.

Dedication

To my parents, for their unwavering support.

Chapter 1

Introduction

The rise of mobile computing is indisputable. Smart phones are now outselling feature phones [1] and tablets sales are on the verge of overtaking desktop and laptop computer sales [2]. Beyond portability, these devices offer many advantages to their users. Chiefly, the functionality of these devices can be seamlessly extended through the installation of applications, making mobile devices limited only by their hardware capabilities. The hardware limitations are constantly being pushed through the miniaturization and continuous improvement of components such as microprocessors, GPS receivers, accelerometers and cameras. This in turn encourages the development of even more innovative applications, rendering mobile devices indispensable to their users.

Millions of applications are now available to mobile users. The barrier of entry to distribute an application on a market is very low, usually only requiring at most a small fee. These applications are developed by a wide variety of organizations and individuals. Some are experienced developers who understand how to program for mobile devices, but others do not fully grasp the development model in their platform of choice. This can lead to vulnerabilities when the application programming interfaces (APIs) are not used properly. Developers' intentions are also very opaque: some may have the users' best interests at heart, while others steal data or money from their users. Unlike the desktop computing world, users have little to no visibility into this. They used to buy software from a relatively small number of companies whose reputation could easily be verified. Now they have to select applications mostly based on their description, from a very high number of developers.

To complicate the situation, there is often not a clear line between the harmless applications and the malicious ones. Markets offer a vast selection of "grayware", which

may not be outright malicious while still misusing users' personal data. It is often impossible for users to detect and avoid misuse of their personal data, as it is routinely collected by applications for legitimate purposes.

The dominant mobile operating system is Android. Unlike iOS, its main competitor, it can be used by any phone or tablet manufacturer. In fact, anyone can download the source code of Android, modify it, compile it and install the resulting system image to their device. Unsurprisingly, countless mobile terminals are thus using Android as their operating system.

Android applications are developed in *components*, which can communicate with each other using sophisticated platform-specific mechanisms. Inter-Component Communication (ICC) enables functionality reuse and data sharing between components. Components can communicate both within single applications and across applications. Unfortunately, properly using ICC in Android is all but simple. Users and developers alike can fall victim to its intricacies and ambiguities. Therefore, in practice, ICC increases the attack surface of applications in subtle ways [3, 4].

Inter-Component Communication is the source of many types of vulnerabilities. Data can be stolen by intercepting ICC messages containing private data. Additionally, privileged applications can be exploited when they do not appropriately protect their interfaces. One of the main facets of the Android security model is the notion that permissions protect access to sensitive resources such as the camera or the GPS receiver. Applications that are granted permissions can leak their privileges through the misuse of ICC [5]. Developers commonly misuse the communication primitives, rendering their own applications vulnerable. Finally, ICC itself can be the vehicle of attacks such as collusion attacks [6, 7], where different applications collaborate to achieve their malicious goals through ICC. Users have no visibility over which applications are interacting with each other and cannot make informed decisions to protect themselves.

It is worth asking which of the numerous mobile ecosystem stakeholders should be in charge of providing users with security guarantees. Cellular operators are not suited for this task. They provide network service and have no direct control over the platform or the applications. They only provide slight customizations to subsidized handsets. Phone and tablet manufacturers have some control over the platform, since they usually ship customized versions of Android with their phones, so they seem like a good candidate. However, they have difficulties providing security updates from the main distribution of Android to their customized versions [8]. Therefore, one cannot expect them to develop any significant additional security-related features. The Android developers

have influence over the security of the Android platform. However, modifications to the platform's security model are a considerable challenge, as compatibility with the current applications has to be maintained.

Application markets have incentives to provide secure applications. They want users to be comfortable downloading and installing applications, since they earn a percentage of application sales. However, they would face significant logistical hurdles if they tried to provide strong guarantees for all the applications they serve [9]. A more fundamental issue is that security is highly contextual. A given user may find an application secure, while another user may consider that it unacceptably violates her privacy. Thus, markets cannot provide meaningful guarantees when it comes to Inter-Component Communication.

In the end, two stakeholders have a strong influence over the security of a mobile device as it relates to Inter-Component Communication: developers and users. Developers have control over the interfaces that their applications are exposing. For example, they can ensure that a component remains private to the application in which it is declared. They can also limit the applications with which a public component is allowed to interact. Users can choose which applications they install. They can potentially avoid vulnerable applications or problematic application combinations. However, when it comes to ICC they cannot make informed choices. In other words, security- and privacy-conscious users do not have the means to enforce the security policies that they deem reasonable. Before they can take steps to select applications based on how they interact with their environment, it is necessary to be able to analyze application ICC. That would enable users to make informed choices before installing applications.

1.1 Thesis Statement

The focus of this work is on statically inferring the late binding between ICC message sources and targets. The binding of ICC messages with their targets normally occurs at runtime. This hinders static analysis, since the potential control or data flows between components are not known. We attempt to accurately describe the specifications of ICC objects so that potential runtime links between application components can be found statically. This applies to communication within single applications and across different applications.

We first note that, while Android applications are developed in Java, they are compiled to a platform-specific format that cannot readily be analyzed by Java tools. In

order to be able to leverage the work that has been done for analyzing Java applications for almost two decades, we need to be able to convert Android applications to a Java representation. For many reasons, this *retargeting* process is challenging. However, we found that after transforming code to an intermediate representation that is typed according to Java rules, we can perform retargeting by using only a small number of generic code rewriting rules.

After we establish reliable retargeting techniques, we formulate a data flow model for Inter-Component Communication. Our model allows us to infer the values of ICC messages that we use to form links between components. The inter-component links that we infer represent potential control or data flows between application components. Our techniques are all evaluated using large-scale experiments. We show aggregate data and we prove that the processes we develop can scale to the large numbers of applications present on individual devices or even on application markets. We now formulate our thesis statement.

> *Program retargeting can be used to bridge the gap between new application formats and existing analysis tools to enable the analysis of applications from emerging platforms. Given a retargeted program and a specification of the data flows affecting inter-component messages, we can build a data flow model that is sufficient to infer inter-component communication.*

This thesis has two fundamental parts. In our work, application retargeting is an essential prerequisite to any other analysis. Therefore, we first develop reliable techniques to retarget Android applications to Java bytecode, which can already be analyzed using multiple existing analysis tools. The second thrust of this work consists in developing Java bytecode analyses to infer the potential runtime links between application components.

1.2 Contributions

In this dissertation, we make the following contributions:

- *We develop techniques to accurately retarget Android applications to Java bytecode.* The *ded* system serves to identify the challenges involved with retargeting. It also provides insight on how to tackle these challenges. The Dare method [10] subsequently formalizes the retargeting process and the measurement of its success. It is based on a translation to an intermediate representation that is optimized for the retargeting problem. In particular, the intermediate representation provides

adequate typing for local variables where needed. Dare handles malformed input bytecode and rewrites it in order to make the output bytecode valid. The rewriting process preserves the runtime semantics of the program. We evaluate Dare on 1100 applications and find that it is successful in 99.99% of cases.

- *We introduce a static analysis to infer the values of ICC objects.* The Epicc tool [11] performs static data flow analysis of retargeted Android applications. It is based on a reduction to an Interprocedural Distributive Environment (IDE) problem. This class of problems can be solved using existing approaches in an efficient manner. We evaluate Epicc on 1200 applications and compare it to an existing research tool for finding potential ICC vulnerabilities. Results indicate that Epicc generates fewer false positives than the other tool.

- *We introduce a generic IDE model for finding the value of complex Java objects and we apply this model to ICC object inference.* In order to enable complete coverage of ICC methods, we develop the COAL language to express constant propagation problems in a textual manner. This results in simpler specification of problems such as the one solved by Epicc. The COAL solver uses a generic IDE model and a textual COAL specification to solve ICC problems. Using COAL, we build the IC3 tool for inferring ICC objects in a more precise manner than Epicc. We evaluate IC3 on 350 applications and find that its precision is significantly higher than Epicc.

- *We develop a formal process for statically computing links between application components using ICC specifications.* We develop an efficient algorithm for finding all potential links between a set of components. We identify challenges in visualizing these inter-component links and we propose some solutions for them. In particular, we attempt to distinguish between artifacts of the IC3 inference process and the actual structure of inter-component links.

1.3 Dissertation Outline

In Chapter 2, we present some concepts that are used throughout. We begin by presenting the mechanisms of Android inter-component communication. We then show an overview of the Interprocedural Distributive Environment (IDE) framework, which is used to solve interprocedural data flow problems. These preliminary concepts are followed by a description of the related work in Chapter 2. We survey existing work related

to program retargeting, interprocedural constant propagation and Android application security.

Chapter 4 addresses the problem of retargeting Android applications to Java bytecode. It describes the challenges involved in retargeting and the first solutions provided by the ded tool. We then identify limitations of this first approach and introduce a formalization of the retargeting process used by the Dare tool. We also describe principled techniques to retarget invalid input bytecode to generate valid Java bytecode.

In Chapter 5, we develop an IDE model for ICC that allows us to determine specifications of ICC objects. This results in Epicc, our first ICC analysis tool. The IDE problem is solved using existing algorithms, resulting in a precise and efficient analysis.

In Chapter 6, we generalize the ideas first introduced in the Epicc system. We introduce a new type of constant propagation problem. We propose the COAL language to specify such problems in a descriptive, textual manner. These problems are solved using the COAL solver, which is based on a generic reduction to an IDE problem. Using COAL, we build the IC3 tool to find the values of ICC objects. Unlike Epicc, IC3 handles all ICC functions and it is therefore more precise.

In Chapter 7, we use ICC specifications to generate and visualize inter-component links. We formalize the matching of components sending ICC messages with potential target components. We develop efficient algorithms to perform this linking process. We find that the visualization process is made difficult by some artifacts of the static analysis that determines ICC specifications and we provide solutions to isolate the artifacts from the links that describe the structure of inter-component communication.

Finally, Chapter 8 concludes this dissertation and offers directions for future work.

Chapter 2

Preliminary Concepts

2.1 Android Inter-Component Communication

Figure 2.1 shows a conceptual representation of an Android application. Android applications are built in components, which perform specific tasks. In practice, implementing a component is done by subclassing one of four classes from the Android framework. *Activities* are the most common component. They represent a user screen. *Services* perform long-running background processing. *Broadcast Receivers* receive system-wide notifications, such as the one that is sent by the operating system when a text message is received. Another notable example is the notification that the system has just finished starting, which is received by applications that request to start when the phone boots. Finally, *Content Providers* provide a way of sharing structured data between applications.

Almost all components are declared in the manifest file that is part of all application packages. The only exception is that Broadcast Receivers can be created and registered dynamically in the application code at runtime. Application components can communicate with one another using two mechanisms, *Uniform Resource Identifiers (URIs)* and *Intents*. URIs are used to address Content Providers. Intents, on the other hand, are messages that are sent between the three other component types.

An Intent can be *explicit*, which means that its target is explicitly named. In other words, the Intent specifies the application and the class name for its target. Intents can also be *implicit*, in which case they only specify the functionality that they desire for their target. In this case, the desired functionality is described using three items:

- An *action* string specifies the action to be performed with the Intent. For example,

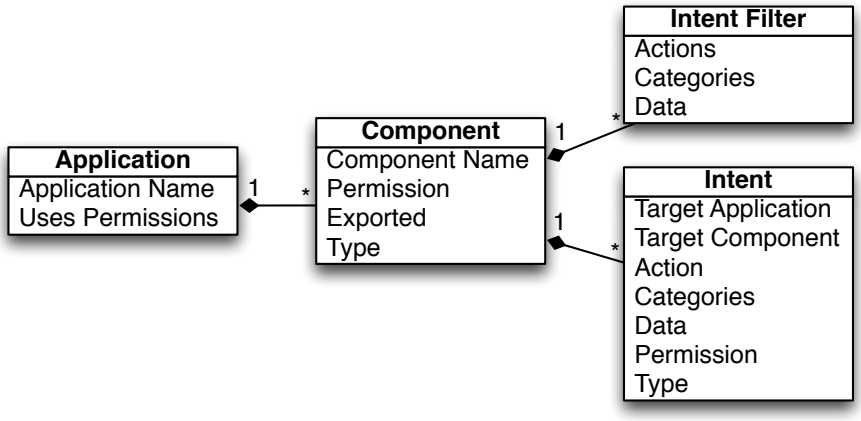


Figure 2.1. Representation of an Android application with components, Intents and Intent Filters.

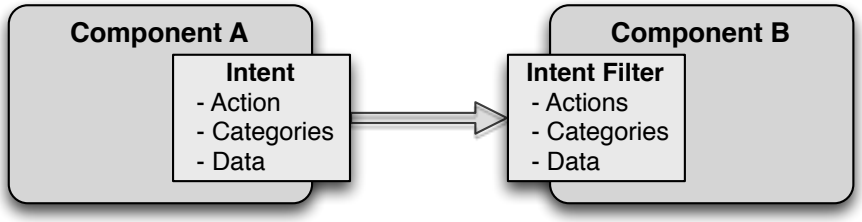


Figure 2.2. Implicit Intent ICC.

a very common action is the `ACTION_VIEW` action, which is utilized when some data needs to be displayed (e.g., web page).

- A set of *category* strings describes additional information about what should be done with the Intent. For example, `CATEGORY_BROWSABLE` indicates that the target can be safely invoked from a browser.

- A set of *data* fields specifies data to be acted upon. This can for instance be a web address or a phone number.

Data is described in several ways. A URI can encode multiple types of data, such as file paths or web addresses. A MIME type can also be specified to describe the data to be acted upon. For ease of exposition, in the remainder of this chapter we do not describe all data fields separately. The ideas we introduce are applicable to all individual data fields.

In order for a component to be able to receive implicit Intents, *Intent Filters* have to be specified for it in the application's manifest file. Illustrated in Figure 2.2, Intent

```
1 public void map(float latitude, float longitude) {
2   Intent intent = new Intent();
3   intent.setAction("android.intent.action.VIEW");
4   Uri geoUri = Uri.parse("geo:" + latitude + "," +
       longitude);
5   intent.setData(geoUri);
6   startActivity(intent); }
```

(a) Intent targeted at components that can render a map.

```
1 <activity android:name="MapRenderingActivity">
2   <intent-filter>
3     <action android:name="android.intent.action.VIEW"/>
4     <data android:scheme="geo"/>
5     <category
         android:name="android.intent.category.DEFAULT"/>
6   </intent-filter>
7 </activity>
```

(b) Example Intent Filter declaration to receive the Intent in (a).

Figure 2.3. Example Intent and Intent Filter used for rendering a map.

Filters describe the action, category or data fields of the Intents that should be delivered by the operating system to a given application component. Components have an *exported* attribute, which when set to true makes the components accessible to other applications through ICC. Components that are not exported are only accessible to other components in the same application. Component access can also be protected by a permission. When a component declares a permission, applications need to request the permission at install time in order to be able to send Intents to the component. This is done using the *uses-permissions* attribute in the manifest file. Intents that target Broadcast Receivers can also be protected by a permission, in which case the application containing the target component needs to request the permission.

Matching Intents with their target is done by the operating system during an *Intent resolution* process. For implicit Intents, it involves matching the action, category and data fields with compatible Intent Filters. In this paper, we statically perform this matching process in order to visualize ICC. Note that in Figure 2.1 some fields can be undefined. For example in an explicit Intent the action, categories and data fields are usually null.

Figure 2.3 shows a representative example of Android ICC. Figure 2.3(a) shows a

method that sends an Intent in order to render a map centered at given coordinates. An Intent *intent* is created. Its action is set to `android.intent.action.VIEW`, which is a generic action used to display many kinds of data. The data of the Intent is defined to be a URI with the `geo` scheme followed by coordinates. When the *startActivity*() method is called, the operating system resolves potential target components, prompting the user to choose a recipient if several components match.

Figure 2.3(b) is a component declaration as it can be found in an application manifest. The `activity` element (Line 1) declares that the application contains an Activity component with name `MapRenderingActivity`. It includes an Intent Filter with several attributes. The `action` line specifies that the action field of Intents received by the component should have value `android.intent.action.VIEW`. The `data` declaration at Line 4 specifies that any received Intent should carry data in the form of a URI with a `geo` scheme. Finally, the `category` line declares that received Intents should also carry the `android.intent.category.DEFAULT` category. This category is added by the OS to implicit Intents targeting activities, such as the one on Line 6 of Figure 2.3(a). Therefore, the component declared in Figure 2.3(b) could receive the Intent created in Figure 2.3(a).

Intents can carry extra data in the form of key-value mappings. This data is contained in a Bundle object associated with the Intent. Intents can also carry data in the form of URIs with context-specific references to external resources or data.

2.2 The IDE Framework

The main part of our Inter-Component Communication analysis as presented in Chapters 5 and 6 is based on the IDE framework [12]. In this section, we summarize the main ideas and notations of the IDE framework. A complete description is available in [12]. The IDE framework solves a class of interprocedural data flow analysis problems. In these problems, an environment contains information at each program point. For each program idiom, environment transformers are defined and modify the environment according to semantics. The solution to this class of problems can be found efficiently.

2.2.1 Supergraphs

A program is represented using a *supergraph* G^*. G^* is composed of the control flow graphs of the procedures in the program. Each procedure call site is represented by two nodes, one call node representing control right before the callee is entered and one

```
1  public void onClick(View v) {
2    Intent i = new Intent();
3    i.putExtra("Balance", this.mBalance);
4    if (this.mCondition) {
5      i.setClassName("a.b", "a.b.MyClass");
6    } else {
7      i.setAction("a.b.ACTION");
8      i.addCategory("a.b.CATEGORY");
9      i = modifyIntent(i);
10   }
11   startActivity(i);
12 }
13 public Intent modifyIntent(Intent in) {
14   Intent intent = new Intent(in);
15   intent.setAction("a.b.NEW_ACTION");
16   intent.addCategory("a.b.NEW_CATEGORY");
17   return intent;
18 }
```

Figure 2.4. Example interprocedural Intent creation.

return-site node to which control flows right after exiting the callee. Figure 2.5 shows the supergraph of the program in Figure 2.4.

The nodes of a supergraph are program statements. There are four kinds of edges between these nodes. Given a call to procedure (p) with call node (c) and return-site (r), three kinds of edges are used to model the effects of the procedure call on the environment:

- A call edge between (c) and the first statement of (p).
- A return edge between the last statement of (p) and (r).
- A call-to-return edge between (c) and (r).

All other edges in the supergraph are normal intraprocedural flow edges. Informally, the call edge transfers symbols and associated values from the calling method to the callee when a symbol of interest is a procedure argument. The return edge transfers information from the return value of the callee to the environment in the calling procedure. Finally, the call-to-return edge propagates data flow information that is not affected by the callee, "in parallel" to the procedure call (e.g., local variables).

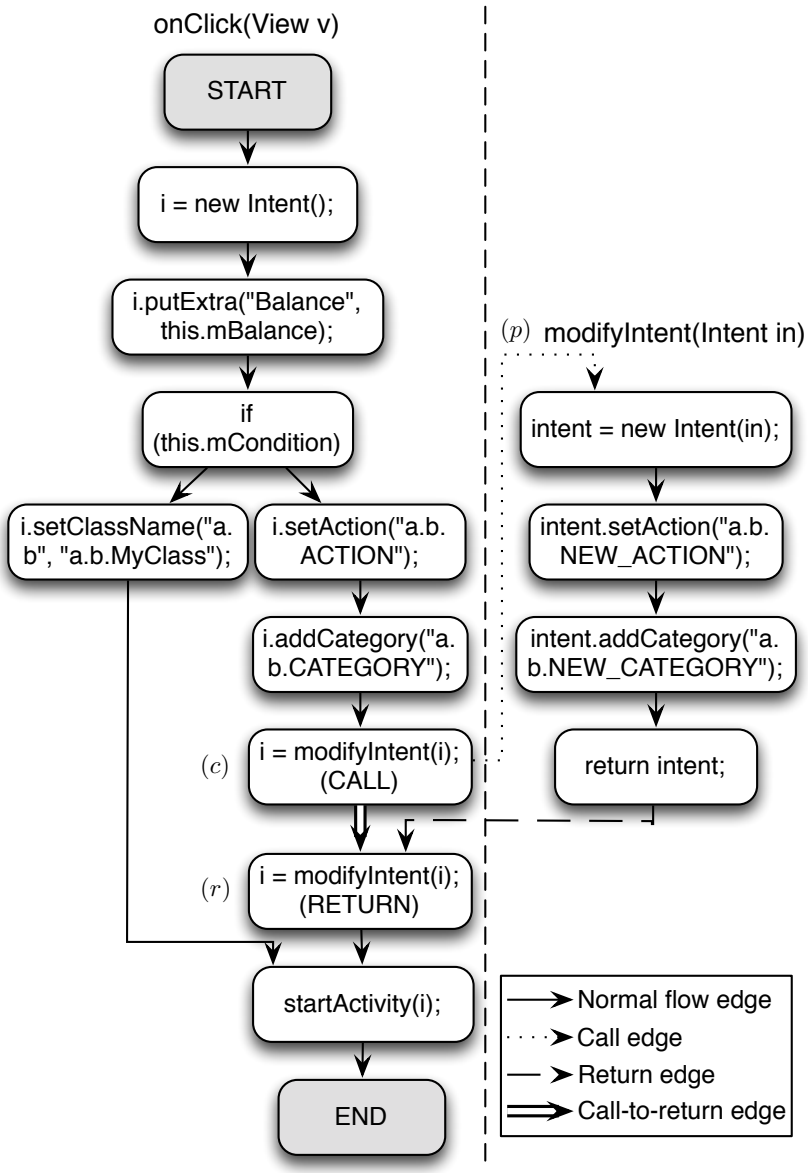


Figure 2.5. Supergraph G^* for the program from Figure 2.4.

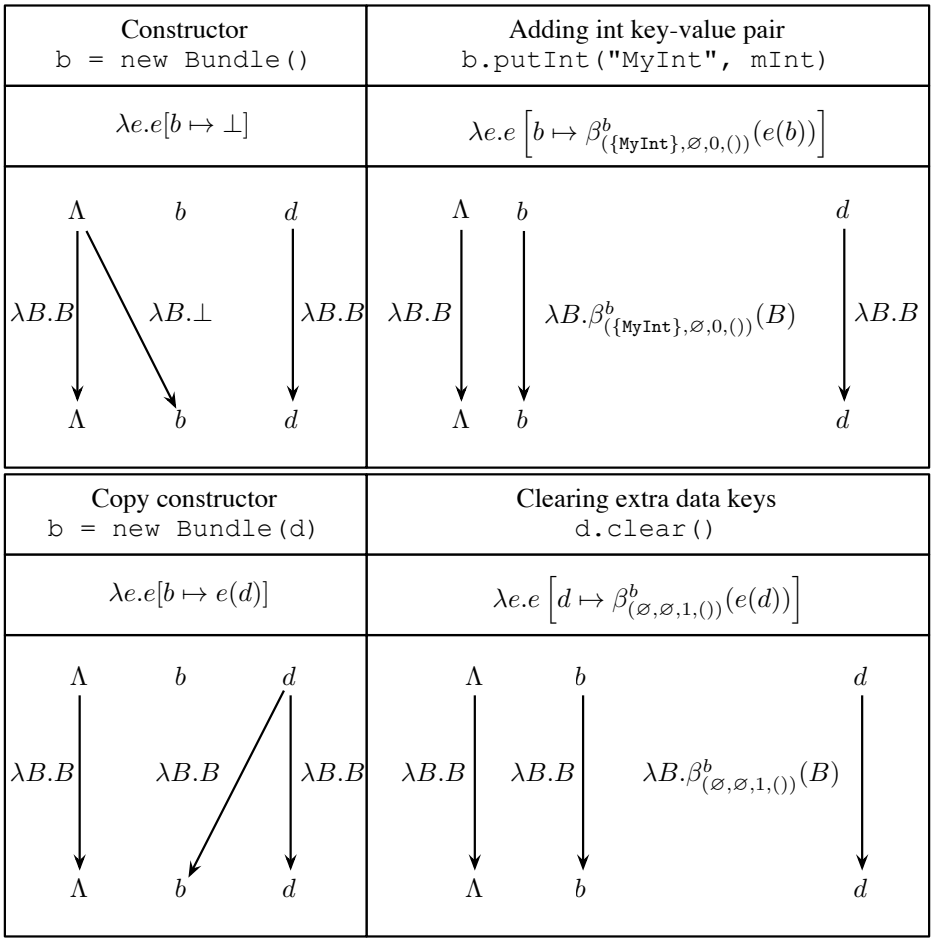


Figure 2.6. Pointwise environment transformers for common Bundle operations.

2.2.2 Environment transformers

Let D be a finite set of symbols (e.g., program variables). D contains at least a symbol Λ that represents the absence of a data flow fact. Let $L = (V, \sqcup)$ be a join semilattice with bottom element $\bot$, where V is a set of values[1]. An *environment* e is a function from D to L. The set of environments from D to L is denoted by $Env(D, L)$.

Operator $\sqcup$ is defined over $Env(D, L)$ as a natural extension of $\sqcup$ in semilattice L: for $e_1, e_2 \in Env(D, L)$, $e_1 \sqcup e_2$ is such that, for all $d \in D$, $(e_1 \sqcup e_2)(d) = e_1(d) \sqcup e_2(d)$.

An *environment transformer* is a function from $Env(D, L)$ to $Env(D, L)$. The algorithms from [12] require that the environment transformers be *distributive*. An environment transformer t is said to be distributive if for all $e_1, e_2, \ldots \in Env(D, L)$, and $d \in D$, $(t(\sqcup_i e_i))(d) = (\sqcup_i t(e_i))(d)$. It is denoted by $t : Env(D, L) \longrightarrow_d Env(D, L)$. Environ-

[1] A join semilattice is a partially ordered set in which any two elements have a least upper bound.

ment transformers have a pointwise representation. We show an example on Figure 2.6. Given environment $e \in Env(D, L)$, transformer $\lambda e.e$ is the *identity*, which preserves the value of e. Given symbol $b \in D$ and value $B \in L$, $\lambda e.e[b \mapsto B]$ transforms e to an environment where all values are the same as in e, except that symbol b is associated with value B. The functions from L to L (represented next to each arrow in Figure 2.6) are called *micro-functions*.

The environment transformer for the copy constructor call `b = new Bundle(d)` is $\lambda e.e[b \mapsto e(d)]$. It means that the value associated with b after the instruction is the same as d's value before the instruction. In the pointwise representation, this is symbolized by an arrow between d and b with an identity function next to it.

We are trying to determine the value associated with each symbol at program points of interest, which is done by solving an Interprocedural Distributive Environment (IDE) problem. An instance IDE problem is defined as a tuple (G^*, D, L, M), where:

- $G^* = (N^*, E^*)$ is the supergraph of the application being studied.
- D is the set of symbols of interest.
- L is a join semilattice $(V, \sqcup)$ with least element $\perp$.
- M assigns distributive environment transformers to the edges of supergraph G^*, i.e., $M : E^* \longrightarrow (Env(D, L) \longrightarrow_d Env(D, L))$.

Under certain conditions on the representation of micro-functions, an IDE problem can be solved in time $O(ED^3)$ [12]. For example, micro-functions should be applied in constant time. In the model we present in Section 5.3, we relax some of these constraints but find that the problem can still be solved efficiently in the average case. When the problem is solved, we know the value associated with each symbol at important program points.

Chapter 3

Related Work

This dissertation introduces methods to study inter-component communication in smart phone applications after retargeting them to Java bytecode. In this chapter, we start by presenting some approaches that have been used for purposes related to retargeting. We then show existing approaches for constant propagation since we later reduce inter-component communication to a constant propagation problem. Finally, since the most compelling application of our work is in security, we survey the security challenges addressed in the literature as well as methods to address them.

3.1 Techniques for Application Retargeting

Retargeting refers to the transformation of a program from one binary format to another. It is closely related to decompilation, which aims to recover source code from binary code. Java decompilers have been around for almost as long as the language itself. Krakatoa [13] was introduced in 1997 and Mocha [14] shortly thereafter. These earlier tools could not process a number of common bytecode structures. Dava [15, 16, 17] significantly improved the state of the art in Java decompilation in 2001. There are two types of Java decompilers. The first type targets code generated by specific Java compilers. Such tools generate control flow structures such as loops by recognizing known bytecode structures. These tools may not perform well with code that is generated by other compilers that generate unexpected bytecode control flow patterns. The other type of decompilers can handle bytecode generated by any type of compiler. Since control flow structures such as loops and if statements can be hard to interpret from the bytecode, these decompilers sometimes produce code that is not easily readable. Dava belongs to

the second category of decompilers. It addresses the issue of code readability through manipulations of the Abstract Syntax Tree (AST) that represents the program. Dava is built upon the Soot framework [18]. Other tools such as Jad [19] and JD [20] are available but their algorithms are not published. Decompilation of other languages such as C/C++ [21, 22, 23, 24, 25] differs vastly from the challenges that we describe in Chapter 4, in that the residual information in the executable code is substantially lower than Android or Java bytecode.

Program retargeting and decompilation often requires some type inference. Type inference in Java has been widely studied [26, 27, 28]. Conceptually, these systems generate and solve constraints on types. A solution for the constraints is found using a variety of algorithms: graph heuristics [27], solving assignment constraints before use constraints [26] or introducing new types [28]. These approaches achieve different guarantees, e.g., optimal typing [26] or polynomial time solution [27]. The constraint-solving algorithm we use in Section 4.5.2 for type inference has been used by other authors [29, 30].

Retargeting Dalvik bytecode as presented in Chapter 4 is closely related to the Java decompilation problem. In both cases variable types need to be inferred from bytecode and the instructions interpreted and translated into the targeted language (in our case Java bytecode). A major difference is that decompiling Java bytecode requires recovering control flow statements from bytecode. In the case of Dava, this is done by making transformations to the AST of individual methods. An alternate approach recognizes bytecode patterns generated by known compilers. However, for retargeting Dalvik to Java bytecode, we can avoid these complexities by mirroring the original Dalvik control flow statements directly.

The idea of having a rich intermediate language (with types or semantics) has been used by several authors. For example, Lim and Reps [31] developed a language called TSL for describing the semantics of an instruction set, along with a run-time system to support the static analysis of executables written in that instruction set. A language similar to TSL called λ-RTL was developed by Ramsey and Davidson [32]. Typed Assembly Language (TAL) extends traditional untyped assembly languages with typing annotations, memory management primitives, and a sound set of typing rules. These typing rules guarantee the memory safety, control flow safety, and type safety of TAL programs [33].

3.2 Interprocedural Constant Propagation

Constant propagation traditionally attempts to determine when variables always have the same value at a particular program point. Initial work on constant propagation [34, 35] performs intra-procedural analyses, which means that they do not cross the boundaries of functions. In this case, analyses have to assume that function arguments and return values can take on any value, which is often imprecise. Later work has addressed interprocedural constant propagation [36, 37, 38, 39, 40], initially using techniques such as procedure inlining [36].

In the IDE framework [12], a dynamic programming version of the functional approach [41] to interprocedural analysis is used. The functional approach summarizes the influence of procedures on program symbols using a summary function. The summary function can simply be applied to determine the influence of a procedure call on the variables of a program. The first example that is presented using the IDE framework is a constant propagation problem. A more detailed presentation of the IDE framework is included in Section 2.2.

For each constant, all these works seek to find a single value that is common to all interprocedural paths. In Chapters 5 and 6, we use multi-valued constant propagation, where we determine all possible values of a constant at each program path of interest. Multi-valued constant propagation [42] has received less attention.

3.3 Mobile Application Security

Mobile application security has been addressed in several ways [43]. In this section, we describe the research that has been performed on permissions, inter-component communication and information flow analysis.

3.3.1 Permissions

Permissions are an essential part of the security model of mobile platforms [44]. They protect access to sensitive resources such as GPS and accelerometer data. In Android, they are requested and granted when an application is installed and enforced at runtime by the operating system.

Permission analysis tries to infer properties about applications based on the permissions requested at install time. Various approaches have been used to perform permission analysis. Requirements engineering [45], machine learning [46, 47, 48], static

analysis [49, 50] and testing [51] are among the techniques that have been used. In early permission analysis work, the Kirin system [45] uses permissions to flag applications with potential dangerous functionality, as indicated by the requested permissions. It extends the Android application installation process to prevent installation of flagged applications. [52] also use permission analysis as the basis of a system to detect Android malware. Additional heuristics are used to filter out applications and reduce false positives.

Other work has attempted to detect applications that request too many permissions for the functions that they are using, thereby breaking the principle of least privilege [53]. The challenging part of this work consists in mapping protected API methods to the permissions that protect them. Due to the evolving nature of the platform [54, 55], the Android developer documentation does not accurately describe this mapping. Thus, approaches that are based on parsing developer documentation [56] cannot get accurate mappings. Finding permission maps is done using testing [51] or static analysis [49, 50, 57]. Once the mapping is done, a simple reachability analysis can detect which API calls are made by a given application. The permission map is then used to compute the list of permissions that are actually needed by the application, which is compared to the permissions requested by the application.

There have been several attempts to remedy the limitations of the permission model in smart phone platforms. A first limitation is that individual permissions are coarse-grained. For example, the Internet permission gives access to all Internet domains. It has been suggested that this model could be improved if applications were able to request finer-grained permissions [46]. For example, an application could be given access to specific Internet domains only. To address this issue, [58] proposes an approach where permission usage is inferred in order to allow repackaging of applications to use finer-grained permission. This approach also gives some insight to address another limitation of the permission model. When users decide to grant permissions to an application, they often cannot know what the permissions are used for. The capabilities granted by permissions can be used for benign as well as malicious purposes and it is not possible for users to distinguish these uses *a priori*. The WHYPER system [59] uses natural language processing to infer permission usage patterns from application descriptions. While the system is helpful in informing users about how sensitive resources are used, it cannot prevent malicious permission usage, since such usage is presumably not explicitly described in application descriptions.

Saint [60] modifies the Android framework to enable more expressive policies for

permissions and application interaction. It allows application developers to specify how their custom permissions can be granted. It also regulates application interaction so that ICC interfaces can only be used in ways that are compliant with a policy.

Apex [61] extends the permission-granting process to allow users to only select the permissions that they are willing to grant at install-time. Policies can also be specified to restrict the usage of protected resources, for example to limit the number of text messages sent by an application. This finer-grained policy is then enforced at runtime. Similarly, CRePE [62] enables applications to be granted in a fine-grained way, depending on context such as physical location. Aurasium [63] repackages Android application into its own sandbox that intercepts the system calls made by the application. Higher-level semantics are reconstructed from the system calls in order to enforce security policies. For example, Aurasium can infer when an application is attempting to access a device identifier and it can prevent it. A variety of policies beyond permission control can be enforced by using the approach introduced by Aurasium.

Usability is an important aspect of permission-based access control, since the decision to grant permissions is made by the users of mobile platforms. It has been found that users often click through security prompts without thoroughly understanding them [64], including in the context of smart phone permissions [65, 66, 67]. Several approaches have been proposed to improve the placement of permission prompts. A graph-based analysis [68] enables placing permission prompts such that sensitive resource accesses are protected but users are faced with a minimal number of prompts. In order to cope with the fact that Android permissions are not effective since users tend to forget that they granted them during application installation, it has also been suggested that permissions should be granted in heterogeneous ways [69]. For example, some resources should be protected by install-time permissions while others should be protected by runtime prompts. Other methods for controlling access to sensitive resources include capturing user intent [63] and using crowdsourcing to bridge the gap between user expectations and application behavior [70, 71].

3.3.2 Inter-Component Communication

Android applications can reuse functionality by using Inter-Component Communication (ICC). For example, it is possible to open a web page from an application by sending a message (or *Intent*) that is forwarded as appropriate to a web browser application. iOS provides similar facilities where applications can register their handling of specific URL types. Other applications can then use the registered functionality. This method

of performing IPC is powerful and relatively easy to use for developers. Instead of expressing the target of IPC through component names, it can be expressed by desired functionality. Unfortunately, this mechanism also increases the attack surface of applications. ICC messages can be intercepted by unexpected or even malicious applications. This can lead to the theft of sensitive data. Component functionality can also be hijacked by malicious messages. This is a particularly serious issue when the vulnerable component has elevated privileges, as it can result in confused deputy [72] issues. Other vulnerabilities can be exploited in target applications [73]. Finally, an attack vector for Android is application collusion [6, 7], where two or more applications communicate to perform malicious functions. Each individual application may only have an innocuous set of permissions, but malicious behavior is achieved through information and privilege sharing.

In early ICC work, SCanDroid [74] provides a static data flow analysis. To resolve the string arguments of ICC objects, a pointer analysis is used to generate a flow graph for string operations. Methods that modify the strings are partially evaluated to obtain approximations of the prefix of the actual values. An interprocedural taint analysis based on the IFDS framework [75] is used. Given the flows from different applications and inter-component flows, a constraint-based approach was used to check for data leaks. Results were reported on a few examples but no testing was done on real-world applications.

The ComDroid tool [4] is used to find ICC vulnerabilities in Android applications. It looks for exposed components using the manifest file of each application. It additionally infers a limited number of properties of Intents using a flow-sensitive intraprocedural analysis with a limited interprocedural analysis. ComDroid has a high false positive rate. This is due to the fact that, while a component may be exposed, there is no vulnerability if the component does not have elevated privileges or if there is no path between the entry point of the component and the sensitive API calls. Other work has attempted to find ICC vulnerabilities with better precision [5]. [5] builds call graphs of exposed application components to find paths between entry points and sensitive API calls. It is able to find vulnerabilities in five system applications. To mitigate these attacks, it proposes IPC Inspection, which restricts permissions in the case of inter-application ICC. When such ICC is used, it drop privileges to the permissions that are common to both applications. The path-finding procedure of [5] is limited in that the call graph it uses does not take application lifecycle or object-oriented primitives (e.g., inheritance) into account. It may miss problematic paths and thus false negatives are possible.

The CHEX tool [76] addresses some of the limitations of the early work on static component hijacking detection. It models applications as a set of *splits*. A split is a piece of code that can execute independently of other splits. This representation models the fact that application components are relatively independent and can be called in an arbitrary order. Splits additionally model event-driven programming in Android, where event handlers can also be called in arbitrary orders in response to user interactions. For each split, a *split data-flow summary* is generated using a data dependence graph (DDG). The hijacking detection is done on a permutation of the splits and consists in a basic reachability analysis. Out of 5486 applications, CHEX finds 254 as potentially vulnerable. Manual analysis estimates the accuracy of the 254 flags to be 81%.

The Woodpecker [77] and DroidChecker [78] tools also apply static analysis to find component hijacking vulnerabilities. They also use reachability analysis to find paths between entry points and sensitive method calls. Woodpecker is applied to applications pre-installed on a variety on Android phones. Out of 13 permissions examined by Woodpecker, 11 are found to be leaked by at least one the eight smart phones under test. DroidChecker is tested on 711 applications that request dangerous permissions and has a precision of 26%. The SEFA vulnerability detection tool [79] also aims to detect the impact of vendor customizations on the security of Android phones. It leverages existing permission mapping work [50] to detect over privilege. It also detects component hijacking using a traditional reachability analysis. Unlike Woodpecker, it adds support for ICC and it is able to detect inter-component flows that lead to component hijacking. However, no details are given as to how Intent values are resolved.

ContentScope [80] looks for cases where Content Providers are not protected by permissions and may either leak data or be vulnerable to unexpected database modifications. Out of 62519 applications, 3018 are found to be unprotected by permissions. A reachability analysis is used to isolate 1279 applications vulnerable to content leaks and 871 vulnerable to unexpected modifications.

Several approaches mitigate ICC vulnerabilities using dynamic monitoring. The TrustDroid tool [81] assigns three trust levels to each application running on a phone: system applications, trusted third-party and untrusted third-party applications. It prevents ICC between applications from different trust levels. It also prevents the sharing of Content Provider data between applications with different trust levels. It further provides file system isolation, since Android external storage is shared between all applications that have the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permissions. It uses both middleware modifications and customizations to the TOMOYO Linux ker-

nel security module to enforce the mandatory access control (MAC) policies. The ICC enforcement mechanism is derived from the XManDroid tool [82].

The Quire [83] provenance system annotates ICC messages with the full ICC call chain to allow applications to make informed security decisions before making privileged operations. It is implemented using Binder and Java library extensions.

SEAndroid [84] provides a MAC framework for Android. It provides both kernel extensions derived from SELinux [85] and middleware modifications. Building on SEAndroid, FlaskDroid [86] provides MAC for all resources on Android. This includes permission-related policies as in Saint [60] and mediation of inter-component communication.

3.3.3 Information Flow Analysis and Monitoring

Android users cannot know what happens to their data once an application is granted permission to use it. It has been shown that collecting personal information is a common behavior in both malware [87, 88] and popular applications [89]. Information flow analysis has been proposed as a solution to this limitation. In early Android information flow analysis work, TaintDroid [89] performs dynamic taint tracking on Android. It tracks both intra-component and inter-component flows. It can also taint files, which can potentially be used to share data between applications. It has found widespread leakage of personal data to third parties. An extension to TaintDroid handles implicit flows [90] by monitoring and recording control flow information. TaintDroid is also used in the AppFence system [91], which actively prevents sensitive data exfiltration from mobile devices. It replaces users' private data with fake data to preserve privacy. It also prevents exfiltration of private data to the network when data has been restricted to be used on the phone only. Similarly, TISSA [92] allows users to specify whether an application should be granted access to specific resources. For each resource, users can choose to provide real, anonymized, fake or no data. MockDroid [93] similarly enables fake data to be provided to applications to protect sensitive features such as location data and Internet access.

Dynamic analyses such as TaintDroid are limited by the way they interact with the User Interface (UI). SmartDroid [94] tackles this issue by combining static and dynamic analyses. It is able to simulate the UI to expose hidden behavior for seven malware families. The static analysis performs some ICC analysis to resolve a subset of the fields of Intents.

Static taint analysis has been used in several tools to find leaks of sensitive data.

AndroidLeaks [95] uses a System Dependence Graph (SDG), which incorporates both control and data flows between program statements. The SDG is used to compute program slices [96] starting at sensitive sources. Leaks are found by looking for sinks on each slice. Both sources and sinks are defined to be methods that are protected by permissions. Evaluation of AndroidLeaks is performed using 25976 applications from 13 Android markets. 7414 applications are found to potentially leak data, most of which occurs in code that belongs to ad libraries. The verification of 60 leaks in ad library code shows 21 false positives, that is, a precision of 65%. LeakMiner [97] adopts a similar approach using slicing, also using sources and sinks that are method calls protected by permissions.

Ad libraries have in some cases threatened the privacy of mobile users [89]. It has been found that these libraries request many permissions and that they are requesting more and more privileges [98]. In order to study the dangers caused by ad libraries, the AdRisk system is used to perform a study of 100 ad libraries found in a sample of 100,000 applications [99]. From disassembled Dalvik bytecode, the authors build a call graph and try to find paths from dangerous API calls to network sinks. However, the call graph they generate has discontinuities and their analysis involves some manual effort.

PiOS [100] performs static information flow analysis on iOS applications. Applications that run on iOS are compiled from Objective-C, an object-oriented language. Analyzing binaries compiled from Objective-C requires solving several challenges. The control flow graph of such binaries is hard to recover, since Objective-C uses indirect virtual dispatch of instance methods. All virtual calls are made by sending a message through the `objc_msgSend` method. Recovering the possible dynamic targets of such calls is done by PiOS using backward slicing to determine the arguments to `objc_msgSend` function calls. Then paths between sources and sinks are found using reachability analysis on the control flow graph. Finally, a data flow analysis uses these paths to confirms whether data can actually flow between sources and sinks. Experiments on 1407 applications reveal that the leaking of device identifiers is pervasive. A minority of applications leak other sensitive data such as location or contact data.

ScanDal [101] converts Dalvik bytecode to Dalvik Core, a formally defined intermediate language for Android application analysis. Potentially harmful flows are detected using abstract interpretation. Its analysis is path-insensitive and has context-sensitivity limited to a depth of one. It is able to find some actual privacy leaks, but is limited by a high number of false positives and flows that are impossible to confirm. In particular, its handling of Android libraries is problematic – modeling the interaction of applications

with Android libraries is, in itself, a difficult challenge.

In contrast to the approaches that are based on program slicing, FlowDroid [102] uses a reduction to the IFDS framework [103] by Reps *et al.*. Unlike previous approaches, it is context, flow, field and object-sensitive. It models component lifecycles and takes event handler registration into account when building call graphs. In order to facilitate comparison with other taint analysis approaches, [102] introduces the DroidBench benchmark suite. FlowDroid achieves 93% recall and 86% precision on DroidBench.

AppSealer [104] combines static analysis with dynamic enforcement to protect users' privacy. Using static analysis, a slice is generated between sensitive sources and sinks. The slices are used to instrument application code to keep track of the taint status of the program variables. The instrumentation is performed using bytecode rewriting with Soot. At runtime, before tainted data flows to a sink, the user is prompted for confirmation. This approach is also useful for preventing the component hijacking issue. For this case, Intents coming from outside an application are defined to be sensitive sources for the taint propagation. In order to detect whether an Intent is external to an application, all Intents are instrumented with a field that keeps track of their provenance. Applying this patching process on 16 applications that are vulnerable to component hijacking successfully prevents exploits, while keeping runtime overhead to 2% on average.

When performing taint analysis, it is difficult to select sources and sinks when the framework is as large as the Android software stack. There are many sensitive resources, many of which are accessible using several different API methods. In an attempt to provide a systematic way of identifying sources and sinks, SuSi [105] uses a machine learning approach for automated classification. It uses support vector-machines [106] with a set of 144 features. The features considered range from method metadata (e.g., name, required permission), to method code (e.g., local data flow to return statement). The sources and sinks are classified into 12 and 15 semantic categories (e.g., account, bluetooth). SuSi achieves 93% recall and precision. It finds many sources and sinks not considered by TaintDroid, SCanDroid and the decompilation-based method presented in [107]. It also finds cases where TaintDroid tracks the results of methods that are not sources.

In order to cope with the fact that advertisement libraries use users' private data, several approaches have been proposed. AdSplit [108] extends the Android platform to enable applications to run separately from their ad libraries. This allows applications to request only the permissions they need to implement their features. Using AdSplit,

the application and the library can share the screen so that the library can be displayed while the application is running. Using Quire [83], mechanisms that prevent click fraud are implemented. On the other hand, AdDroid [109] introduces an advertisement API associated with specific permissions. The goal is to distinguish permissions that are used for advertisement purposes from the ones that are used to implement the main application features.

Before the advent of Android, there were already techniques for information flow analysis of Java code. [110] presents an analysis of Java programs based on an encoding of flows using boolean functions. This analysis considers all fields to be static and therefore does not distinguish between flows through fields of different instances of the same type. [111] proposes a sound analysis of Java programs based on a points-to analysis. Their relatively scalable context-sensitive analysis allows them to find previously unknown vulnerabilities in existing Java source code. [112] describes a type-based system for secure information flow in Java bytecode. Java bytecode is transformed to introduce new types and the subsequent verification of secure flow only relies on the standard JVM verifier. It differs from approaches such as [30] in that it does not require the programmer to specify type annotations before compilation. [113] and [114] propose a framework that can be used for static information flow analysis of Java programs. The analysis is context-sensitive and works on program fragments. It is based on a points-to analysis, which is used both to generate a call graph and to distinguish between fields of different instances of the same class. This allows the analysis to accurately track flows through object fields. Their framework was later extended to handle implicit flows [115], which represent data transmitted by control flow.

Chapter 4

Retargeting Android Applications to Java Bytecode

At the time of registration, Android developers submit an "application package" containing the program bytecode, resources and an XML manifest to the market. The submitted applications are initially developed in Java, but compiled by the developer into Dalvik bytecode [116]. Android runs each application on the phone in its own instance of the Dalvik Virtual Machine (DVM). The DVM has some major differences with traditional JVM. For example, the DVM is a register-based architecture and has ambiguous register typing (see Section 4.1). These different bytecode and program structures make it impossible to leverage existing Java tools such as Soot [18] or WALA [117] for program analysis of Android applications. Thus, in the absence of usable analysis tools, markets can do little to vet applications.

In this chapter, we develop and evaluate algorithms for retargetting Dalvik to Java bytecode. We present the two retargeting tools we developed. The first one is called `ded`, for "DEx Decompiler". It was used to perform a first study of security properties of Android applications [107]. The output of that retargeting tool was input to Dava [15] – a Java decompiler integrated with the Soot toolkit [18] – and the resulting source code analyzed using Fortify SCA [118]. We found that the *ad hoc* methods used for retargeting in `ded` were often unreliable or failed outright. These failures limited the visibility of the code (and thus the coverage of the analysis), and prevented conclusive results. More specifically, while we were often able to retarget and eventually decompile portions of the application code, about half the applications had classes which were unrecoverable, which made program analysis of complete applications impossible. Further, `ded` was targeting

decompilers and its success rate was measured in terms of decompilation success.

The second tool is called Dare, which stands for DAlvik REtargeting[1]. For Dare, we do not target a specific tool but instead seek to produce verifiable Java bytecode, which ensures that it is accepted by analysis tools. The limitations of `ded` largely motivated and guided the making of our second retargeting tool, which is why we are presenting both in this chapter.

By providing the Java bytecode of Android applications via Dare, we provide a path for users, developers, application market providers (such as Amazon) to perform analysis on Android applications. The following sections detail the structure of Dare. Principally, we focus on solutions that address the key challenges of retargeting Dalvik bytecode. This chapter makes the following contributions:

- We present the `ded` decompiler and discuss its limitations.
- We introduce the Tyde intermediate representation for structured semantic mapping between the VMs. All 257 Dalvik instructions are translated using only 9 translation rules.
- Because sound bytecode typing is necessary for verifiability, we use a strong constraint-based type inference algorithm.
- We introduce code transformations to fix unverifiable input bytecode. In addition to making the code verifiable, these transformations accurately mirror VM runtime behavior.
- We evaluate our algorithms on a sample of 1,100 applications. We successfully retarget 99.99% of the 262,110 classes. Further, while previous tools were able to completely recover less than 60% of the applications in the corpus, we recover over 99%. Retargeting is efficient, taking less than 20 minutes for the entire sample. Finally, our experiments reveal that over 20% of applications in the sample have unverifiable Dalvik bytecode in at least one class.

The remainder of this paper explores the algorithms and structure of `ded` and Dare. Section 4.1 introduces the challenges of retargeting Dalvik applications. Next, Section 4.2 presents the initial `ded` decompiler and discusses its limitations. Then, Section 4.3 outlines the Dare retargeting process. Next, we describe how DVM bytecode is translated into the Tyde intermediate representation (Sections 4.4 and 4.5) and then converted to

[1]Source code and documentation for Dare are available at `http://siis.cse.psu.edu/dare/`.

Java bytecode (Section 4.6). Then we show the causes of unverifiability in Dalvik bytecode and how to reliably retarget unverifiable bytecode (Section 4.7). We finally present the empirical study of Dare in Section 4.8.

4.1 Retargeting Challenges

Instruction Set – Dalvik instructions are vastly different from Java instructions. DVM bytecode has 257 different instructions and 3 pseudo-instructions. Dalvik instructions are two to ten bytes long, and pseudo-instructions have a variable length. The DVM has substantially more instruction formats (over 20) than the JVM.

Pseudo-instructions are used to store extra information related to other instructions (and thus are never executed). Specifically, the Dalvik switch instructions (`packed-switch` and `sparse-switch`) store an offset to a pseudo-instruction. The data describing the switch statement (case values and targets) is stored in a pseudo-instruction placed at the end of the bytecode block. The `fill-array-data` instruction fills an array of primitive elements with values stored in a pseudo-instruction.

The DVM is register-based, whereas the JVM is stack-based. Thus, the DVM uses registers to manage local variables rather than pushing them onto a stack. For example, in Dalvik, `add-int` v_1, v_2, v_3 adds the contents of registers v_2 and v_3 and stores the result to v_1. In contrast, Java bytecode would first push the integer variables onto the stack with `iload` 2 and `iload` 3, perform the addition with `iadd` and store the result using `istore` 1.

Exceptions – There is a significant difference in the type inference algorithm used by the verifiers, related to how they handle exceptions. During the path-sensitive type verification process, the Java verifier considers that any instruction in a `try` block may throw an exception. In reality, not all instructions in each try block are able to throw exceptions. Therefore, the Java verifier considers some unfeasible execution paths. On the other hand, the Dalvik verifier does not consider these unfeasible paths. Occasionally, an unfeasible path leads from a register assignment to a register use with an incompatible type (e.g., an int register assignment reaches a use with float type). It is not an issue in the DVM, since the spurious execution path is not considered by the verifier. However, since the Java verifier follows the unfeasible path during type inference, it leads to unverifiable Java bytecode if nothing is done to remove it.

Bytecode Type System – DVM typing is very different than that of JVM bytecode. The primary differences include:

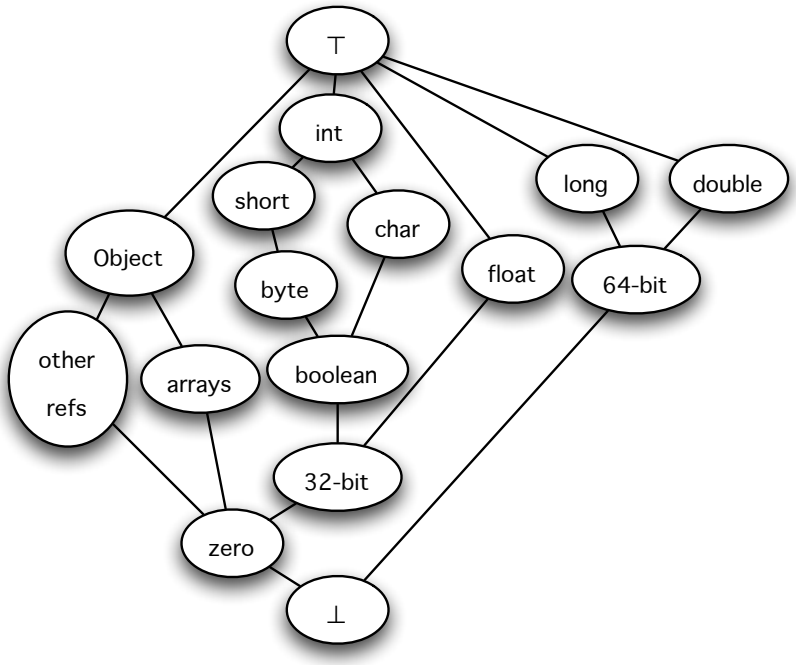


Figure 4.1. Dalvik type lattice.

- *Primitive Assignments* – Dalvik primitive constant assignments specify only the width of the constant (32 or 64 bits). Thus, no distinction is made between int and float or between long and double. In contrast, primitive constants in Java are fully typed.

- *Array Load/Store Instructions* – The DVM has common array-specific load and store instructions for int and float arrays (`aget` and `aput`) and for long and double arrays (`aget-wide` and `aput-wide`). Here again, this introduces type ambiguity.

- *Object References* – Java bytecode uses the `null` reference type to track and detect undefined object references. Conversely, Dalvik uses an integer constant with value 0 to represent both the number zero and the `null` reference. Adding to this ambiguity, a comparison between two integers uses the same instructions as a comparison between object references.

Figure 4.1 shows the lattice of types in the Dalvik architecture. It depicts subtyping relations between types. We have collapsed array and other reference types. The zero, 32-bit and 64-bit types are not valid Java types.

Unverifiable Dalvik Bytecode – Occasionally, part of the Dalvik bytecode of appli-

cations found in the markets is unverifiable. As the DVM and JVM verification processes are similar, retargeting unverifiable Dalvik bytecode usually leads to unverifiable Java bytecode. Our goal is to generate verifiable bytecode on any input, therefore properly dealing with unverifiable input code is an important challenge.

4.2 The `ded` Decompiler

Building a decompiler from DEX to Java for the study proved to be surprisingly challenging. On the one hand, Java decompilation has been studied since the 1990s — tools such as Mocha [14] date back over a decade, with many other techniques being developed [19, 20, 15, 16, 13]. Unfortunately, prior to our work, there existed no functional tool for the Dalvik bytecode.[2] Because of the vast differences between JVM and DVM, simple modification of existing decompilers was not possible.

This choice to decompile the Java source rather than operate on the DEX opcodes directly was grounded in two reasons. First, we wanted to leverage existing tools for code analysis. Second, we required access to source code to identify false-positives resulting from automated code analysis, e.g., perform manual confirmation.

`ded` extraction occurs in three stages: *a*) retargeting, *b*) optimization, and *c*) decompilation. This section presents the challenges and process of `ded`, and concludes with a brief discussion of its validation.

4.2.1 Application Retargeting

The initial stage of decompilation retargets the application `.dex` file to Java classes. Figure 4.2 overviews this process: (1) recovering typing information, (2) translating the constant pool, and (3) retargeting the bytecode.

Type Inference – The first step in retargeting is to identify class and method constants and variables. However, the Dalvik bytecode does not always provide enough information to determine the type of a variable or constant from its register declaration. There are two generalized cases where variable types are ambiguous: 1) constant and variable declaration only specifies the variable width (e.g., 32 or 64 bits), but not whether it is a float, integer, or null reference; and 2) comparison operators do not distinguish between integer and object reference comparison (i.e., null reference checks).

[2] The undx and dex2jar tools attempt to decompile `.dex` files, but were non-functional when `ded` was first developed.

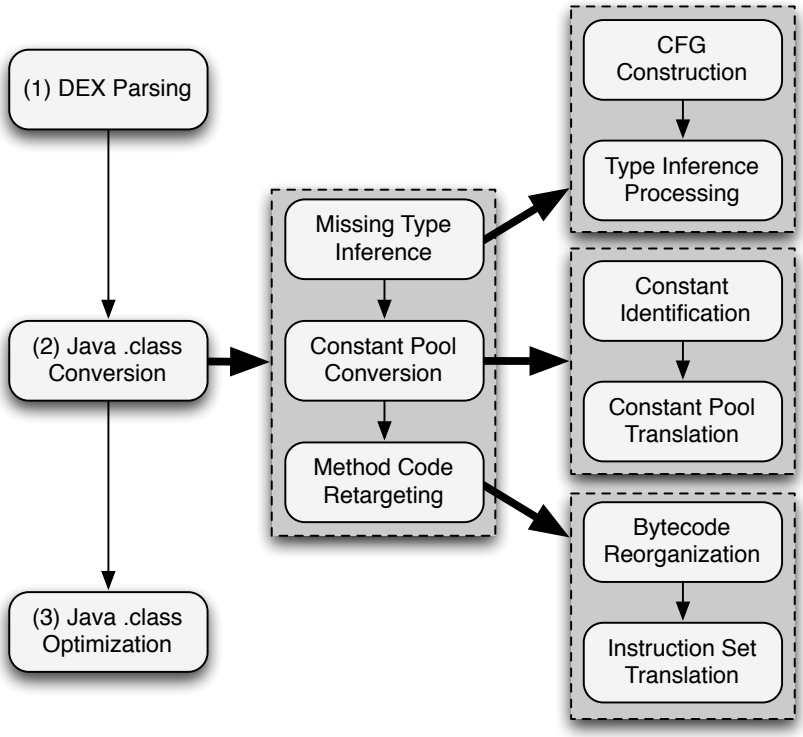


Figure 4.2. Dalvik bytecode retargeting.

Type inference has been widely studied [119]. The seminal Hindley-Milner [120] algorithm provides the basis for type inference algorithms used by many languages such as Haskell and ML. These approaches determine unknown types by observing how variables are used in operations with known type operands. Similar techniques are used by languages with strong type inference, e.g., OCAML, as well as weaker inference, e.g., Perl.

`ded` adopts the accepted approach: it infers register types by observing how they are used in subsequent operations with known type operands. Dalvik registers loosely correspond to Java variables. Because Dalvik bytecode reuses registers whose variables are no longer in scope, we must evaluate the register type within its context of the method control flow, i.e., inference must be *path-sensitive*. Note further that `ded` type inference is also *method-local*. Because the types of passed parameters and return values are identified by method signatures, there is no need to search outside the method.

There are three ways `ded` infers a register's type. First, any comparison of a variable or constant with a known type exposes the type. Comparison of dissimilar types requires type coercion in Java, which is propagated to the Dalvik bytecode. Hence legal Dalvik

comparisons always involve registers of the same type. Second, instructions such as `add-int` only operate on specific types, manifestly exposing typing information. Third, instructions that pass registers to methods or use a return value expose the type via the method signature.

The `ded` type inference algorithm proceeds as follows. After reconstructing the control flow graph, `ded` identifies any ambiguous register declaration. For each such register, `ded` walks the instructions in the control flow graph starting from its declaration. Each branch of the control flow encountered is pushed onto an inference stack, e.g., `ded` performs a depth-first search of the control flow graph looking for type-exposing instructions. If a type-exposing instruction is encountered, the variable is labeled and the process is complete for that variable.[3] There are three events that cause a branch search to terminate: a) when the register is reassigned to another variable (e.g., a new declaration is encountered), b) when a return function is encountered, and c) when an exception is thrown. After a branch is abandoned, the next branch is popped off the stack and the search continues. Lastly, type information is forward propagated, modulo register reassignment, through the control flow graph from each register declaration to all subsequent ambiguous uses. This algorithm resolves all ambiguous primitive types, except for one isolated case when all paths leading to a type ambiguous instruction originate with ambiguous constant instructions (e.g., all paths leading to an integer comparison originate with registers assigned a constant zero). In this case, the type does not impact decompilation, and a default type (e.g., integer) can be assigned.

Constant Pool Conversion – The `.dex` and `.class` file constant pools differ in that: *a*) Dalvik maintains a single constant pool for the application and Java maintains one for each class, and *b*) Dalvik bytecode places primitive type constants directly in the bytecode, whereas Java bytecode uses the constant pool for most references. We convert constant pool information in two steps.

The first step is to identify which constants are needed for a `.class` file. Constants include references to classes, methods, and instance variables. `ded` traverses the bytecode for each method in a class, noting such references. `ded` also identifies all constant primitives.

Once `ded` identifies the constants required by a class, it adds them to the target `.class` file. For primitive type constants, new entries are created. For class, method,

[3]Note that it is sufficient to find *any* type-exposing instruction for a register assignment. Any code that could result in different types for the same register would be illegal. If this were to occur, the primitive type would be dependent on the path taken at run time, a clear violation of Java's type system.

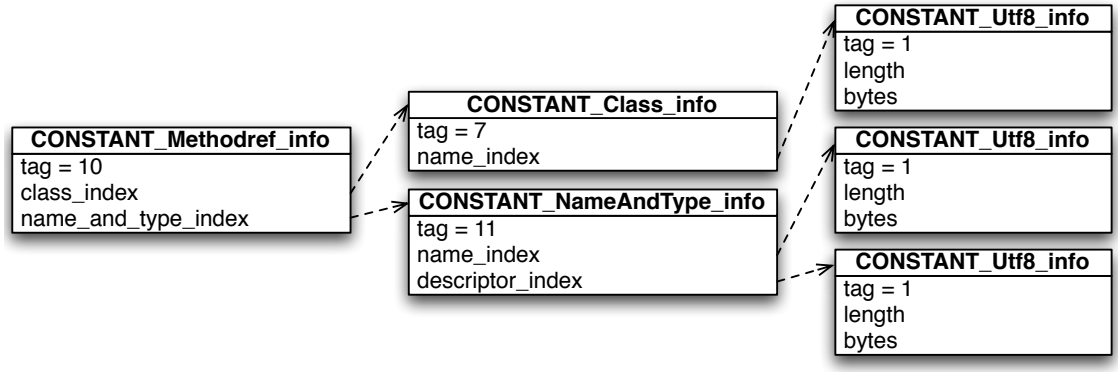


Figure 4.3. Java constant pool entry defining "class name," "method name," and "descriptor" for a method reference.

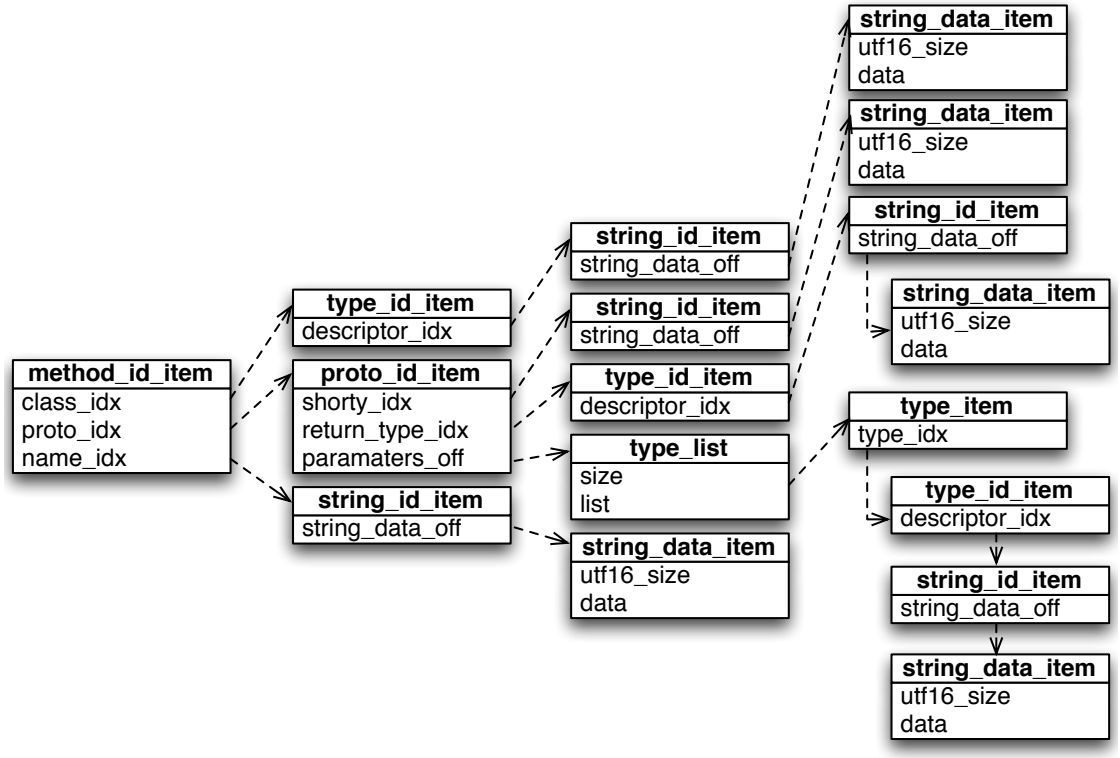


Figure 4.4. Dalvik constant pool entry defining "class name," "method name," and "descriptor" for a method reference.

and instance variable references, the created Java constant pool entries are based on the Dalvik constant pool entries. The constant pool formats differ in complexity. Specifically, Dalvik constant pool entries use significantly more references to reduce memory overhead.

Figures 4.3 and 4.4 depict the method entry constant in both Java and Dalvik formats. Other constant pool entry types have similar structures. Each box is a data structure. Index entries (denoted as "idx" for the Dalvik format) are pointers to a data structure. The Java method constant pool entry, Both figures provide three strings: 1) the class name, 2) the method name, and 3) a descriptor string representing the argument and return types, but vary in complexity.

Method Code Retargeting – The final stage of the retargeting process is the translation of the method code. First, we preprocess the bytecode to reorganize structures that cannot be directly retargeted. Second, we linearly traverse the DVM bytecode and translate to the JVM.

The preprocessing phase addresses multidimensional arrays. Both Dalvik and Java use blocks of bytecode instructions to create multidimensional arrays; however, the instructions have different semantics and layout. `ded` reorders and annotates the bytecode with array size and type information for translation.

The bytecode translation linearly processes each Dalvik instruction. First, `ded` maps each referenced register to a Java local variable table index. Second, `ded` performs an instruction translation for each encountered Dalvik instruction. As Dalvik bytecode is more compact and takes more arguments, one Dalvik instruction frequently expands to multiple Java instructions. Third, `ded` patches the relative offsets used for branches based on preprocessing annotations. Finally, `ded` defines exception tables that describe `try/catch/finally` blocks. The resulting translated code is combined with the constant pool to create a legal Java `.class` file.

The following is an example translation for `add-int`:

Dalvik	Java
`add-int` d_0, s_0, s_1	`iload` s'_0
	`iload` s'_1
	`iadd`
	`istore` d'_0

where `ded` creates a Java local variable for each register, i.e., $d_0 \rightarrow d'_0$, $s_0 \rightarrow s'_0$, etc. The translation creates four Java instructions: two to push the variables onto the stack, one to add, and one to pop the result.

4.2.2 Optimization and Decompilation

At this stage, the retargeted `.class` files can be decompiled using existing tools, e.g., Soot [18]. However, `ded`'s bytecode translation process yields unoptimized Java code. For example, Java tools often optimize out unnecessary assignments to the local variable table, e.g., unneeded return values. Without optimization, decompiled code is complex and frustrates analysis. Furthermore, artifacts of the retargeting process can lead to decompilation errors in some decompilers. The need for bytecode optimization is easily demonstrated by considering decompiled loops. Most decompilers convert `for` loops into infinite loops with `break` instructions. While the resulting source code is functionally equivalent to the original, it is significantly more difficult to understand and analyze, especially for nested loops. Thus, we use Soot as a post-retargeting optimizer. While Soot is centrally an optimization tool with the ability to recover source code in most cases, it does not process certain legal program idioms (bytecode structures) generated by `ded`. In particular, we encountered two central problems involving, 1) interactions between synchronized blocks and exception handling, and 2) complex control flows caused by break statements. While the Java bytecode generated by `ded` is legal, the source code failure rate reported in the following section is almost entirely due to Soot's inability to extract source code from these two cases. We will consider other decompilers in future work, e.g., Jad [20] and JD [19].

4.2.3 Source Code Recovery Validation

We have performed extensive validation testing of `ded` [121]. The included tests recovered the source code for small, medium and large open source applications and found no errors in recovery. In most cases the recovered code was almost identical to the original source, except for comments and method local-variable names, which are not included in the bytecode.

We also used `ded` to recover the source code for the top 50 free applications (as listed by the Android Market) from each of the 22 application categories—1,100 in total. The application images were obtained from the market using a custom retrieval tool on September 1, 2010. Table 4.2.3 lists decompilation statistics. The decompilation of all 1,100 applications took 497.7 hours (about 20.7 days) of compute time. Soot dominated the processing time: 99.97% of the total time was devoted to Soot optimization and decompilation. The decompilation process was able to recover over 247 thousand classes spread over 21.7 million lines of code. This represents about 94% of the total classes in

Category	Total Classes	Retargeted Classes	Decompiled Classes	LOC
Comics	5627	99.54%	94.72%	415625
Communication	23000	99.12%	92.32%	1832514
Demo	8012	99.90%	94.75%	830471
Entertainment	10300	99.64%	95.39%	709915
Finance	18375	99.34%	94.29%	1556392
Games (Arcade)	8508	99.27%	93.16%	766045
Games (Puzzle)	9809	99.38%	94.58%	727642
Games (Casino)	10754	99.39%	93.38%	985423
Games (Casual)	8047	99.33%	93.69%	681429
Health	11438	99.55%	94.69%	847511
Lifestyle	9548	99.69%	95.30%	778446
Multimedia	15539	99.20%	93.46%	1323805
News/Weather	14297	99.41%	94.52%	1123674
Productivity	14751	99.25%	94.87%	1443600
Reference	10596	99.69%	94.87%	887794
Shopping	15771	99.64%	96.25%	1371351
Social	23188	99.57%	95.23%	2048177
Libraries	2748	99.45%	94.18%	182655
Sports	8509	99.49%	94.44%	651881
Themes	4806	99.04%	93.30%	310203
Tools	9696	99.28%	95.29%	839866
Travel	18791	99.30%	94.47%	1419783
Total	**262110**	**99.41%**	**94.41%**	**21734202**

Table 4.1. Studied Applications (from Android Market).

the applications. All decompilation errors are manifest during/after decompilation, and thus are ignored for the study reported in the latter sections. There are two categories of failures:

Retargeting Failures – 0.59% of classes were not retargeted. These errors fall into three classes: *a*) unresolved references which prevent optimization by Soot, *b*) type violations caused by Android's `dex` compiler and *c*) extremely rare cases in which `ded` produces illegal bytecode. In Sections 4.3 through 4.8, we introduce a significantly more reliable retargeting approach.

Decompilation Failures – 5% of the classes were successfully retargeted, but Soot failed to recover the source code. Here we are limited by the state of the art in decompilation. In order to understand the impact of decompiling `ded` retargeted classes verses ordinary Java `.class` files, we performed a parallel study to evaluate Soot on Java applications generated with traditional Java compilers. Of 31,553 classes from a variety of packages, Soot was able to decompile 94.59%, indicating we cannot do better while using Soot for decompilation.

4.2.4 Discussion

The limitations of `ded` largely guided the work presented in the next chapters of this thesis. The main insights we took away from `ded` were the following:

- In order to improve code coverage, a retargeting tool should target any analysis tool, instead of only aiming to be used as pre-processing before decompilation. The decompilation step itself is a source of incomplete code coverage, therefore we should try not to be dependent on it. That is why the Dare retargeting process presented in the remainder of this chapter targets any analysis tool and not just decompilers. That is also why the analyses presented in Chapters 5 and 6 are performed on Java bytecode instead of Java source code.

- Formal retargeting methods should be used. While the *ad hoc* methods described in this section work in a large majority of cases, it is important to handle more complex cases. Indeed, the goal is to be able to analyze all the cases in an application after the retargeting process. The use of formal methods for retargeting enables us to elegantly handle these pathological cases. We show the use of formal methods in the Dare retargeting process in Sections 4.4, 4.5 and 4.6.

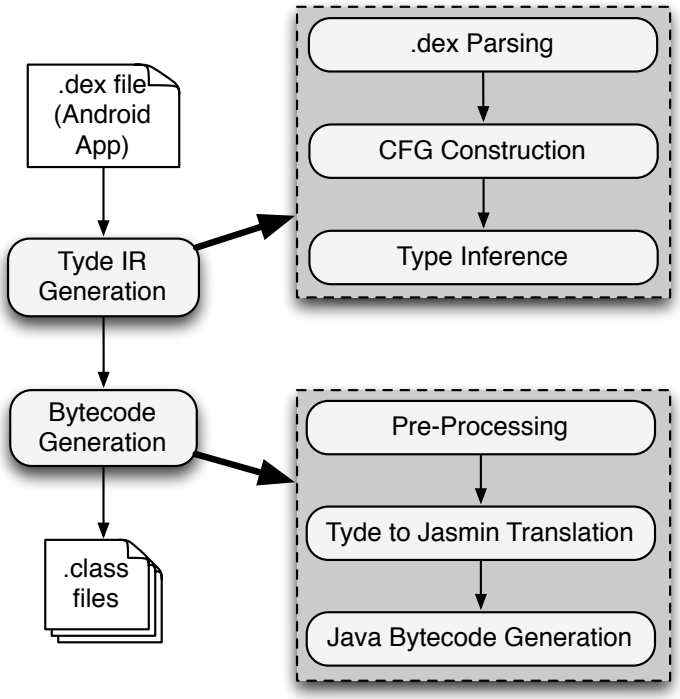


Figure 4.5. Verifiable Dalvik bytecode retargeting overview.

- Invalid input Dalvik bytecode should be handled. Invalid bytecode is common in real-world applications. Note that an application may have invalid bytecode but still function properly, as long as the invalid bytecode is never executed. However, these issues cause some analysis difficulties. It can cause, among other problems, the unresolved references observed in Soot and mentioned above. We address the problem of malformed Dalvik bytecode using Dare in Section 4.7.

4.3 A Formal Retargeting Process for Verifiable Dalvik Bytecode

Figure 4.5 describes the Dare retargeting process for verifiable Dalvik bytecode. We address the issue of unverifiable Dalvik bytecode in Section 4.7. The application bytecode is initially translated into the Tyde intermediate representation (IR) in three steps: *a*) the `.dex` file is parsed and code structures, methods and the global constant pool are interpreted and annotated, *b*) a control flow graph is generated and *c*) register types used in ambiguous instructions are inferred. The Java bytecode is thereafter generated from this IR in three phases: *d*) a pre-processing step generates labels and maps registers

to local variables, e) the IR is translated to Jasmin [122] code, and f) the Jasmin tool generates the final `.class` files.

To illustrate, Figure 4.6(a) shows the source code for a hypothetical method $m2$ and Figure 4.6(b) shows Java bytecode generated by the Java compiler. The `iload_1` instruction loads local variable 1 (variable `a` in the source) onto the stack. The next instruction compares its value to 0. If it is not 0, then `dconst_1` loads double value 1.0 onto the stack and `dreturn` returns it. Otherwise, `ifeq` branches to offset 6. `ldc2_w` loads a constant with value 2.5 from the constant pool; the constant is then returned with `dreturn`.

Figure 4.6(c) shows the Dalvik bytecode for $m2$. `if-eqz` compares the value of register v_3 to 0. If it is not 0, then a 64-bit constant is assigned to register v_0 with `const-wide/high16` and returned with `return-wide`. If v_3 is 0, then the instruction at offset 5 is executed and assigns a different 64-bit constant to v_0. Next, `goto` transfers control to offset 4. The 64-bit constants are detected as long by default by the disassembler we used (dexdump), which is why they appear as long (instead of double) in Figure 4.6(c).

Figures 4.6(d), 4.6(e) and 4.6(f) show the stages of retargeting. The Tyde representation of $m2$ is generated by mapping the Dalvik structures and generated control flow graph into the IR and performing type inference on the ambiguous register references. Once in Tyde, all registers are fully typed in accordance with the Java type system. Figure 4.6(f) shows the retargeted Java bytecode after remapping and Jasmin assembly. This bytecode is functionally equivalent to the one in Figure 4.6(b), albeit longer. That is mostly due to the presence of spurious store/load instructions. However, we are not concerned with optimality but only with semantic equivalence. Tools such as Soot [18] can optimize the resulting bytecode if necessary.

4.4 The Tyde Intermediate Representation

The DVM recognizes 257 different instructions. A naïve approach to converting Dalvik bytecode to Java bytecode would be to have 257 translation rules, which is very cumbersome. Moreover, analyzing the equivalence of the semantics of the translation rules would be very time consuming. The naïve approach would also make the implementation error-prone and hard to maintain.

In this section we describe a typed Intermediate Representation (IR) called *Tyde* (for *Ty*ped *de*x) whose main purpose is to enable easy translation of Dalvik bytecode to Java

```
1 public double m2(int a) {
2   if (a != 0) {
3     return 1.0;
4   } else {
5     return 2.5;
6   }
7 }
```

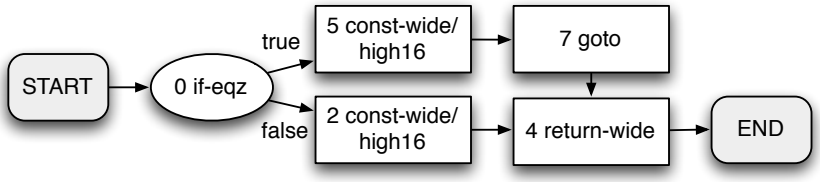
(a) Source code.

```
public double m2(int);
0: iload_1
1: ifeq 6
4: dconst_1
5: dreturn
6: ldc2_w #double 2.5d
9: dreturn
```

(b) Original Java bytecode.

```
public double m2(int);
0: if-eqz v3, 5 // +5
2: const-wide/high16 v0, #long 4607182...
4: return-wide v0
5: const-wide/high16 v0, #long 4612811...
7: goto 4 // -3
```

(c) Dalvik bytecode.



(d) Control flow graph.

```
public double m2(int);
0: if-eqz (3, int, δs), 5
2: const-wide/high16 (0, double, δd) #double 1.0
4: return-wide (0, double, δs)
5: const-wide/high16 (0, double, δd) #double 2.5
7: goto 4
```

(e) Tyde representation – δ_s (resp. δ_d) indicates a source (resp. destination) register.

```
public double m2(int);     8:  dload_2
0:  iload_1                9:  dreturn
1:  ifeq 10                10: ldc2_w #double 2.5d
4:  ldc2_w #double 1.0d    13: dstore_2
7:  dstore_2               14: goto 8
```

(f) Retargeted Java bytecode.

Figure 4.6. Stages of retargeting for method $m2$.

bytecode. As described in Section 4.6, translating the Tyde IR to Java bytecode is done *with only 9 translation rules* for all 257 Dalvik opcodes. The corresponding semantic mapping is much easier to analyze than 257 translation rules. Moreover, this approach also leads to a cleaner and maintainable implementation.

The insight behind Tyde is that, by typing all instruction arguments, load/store operations can be translated independently of opcodes. For instance, let us consider instructions `add-int` v_0, v_1, v_2 (integer addition) and `add-float` v_3, v_4, v_5 (float addition). By typing all registers and specifying if they are source or destination, we can use a single translation rule for both instructions: first translate all source register loads, then translate the opcode and finally translate the destination register store. If we did not determine the type information about the registers, retargeting those instructions would require two different translation rules.

Another advantage of the Tyde representation is that Dalvik pseudo-instructions (such as `packed-switch-payload`) are not used, which leads to a more compact representation. Let us consider method `m3`, whose source code is presented in Figure 4.7(a). Figure 4.7(b) shows the corresponding Dalvik bytecode. It uses a `packed-switch-payload` pseudo-instruction which contains data about the switch statement at offset 0 (note that the default case is implicit). Moreover, in the Dalvik bytecode the registers used as arguments are not typed. Figure 4.7(c) shows the Tyde IR for `m3`. In the Tyde representation, the `packed-switch` instruction has all the necessary data (with an explicit default case) and no pseudo-instruction is used.

4.4.1 Specification

Figure 4.8 presents the type lattice used by the Tyde IR. All types on this lattice are valid Java types. In Tyde, we introduce the notion of typed registers. It adds two elements to Dalvik registers: a type τ and information about whether the register is a source or destination register (represented by terminals δ_s and δ_d).

The notion of typed value is used for all source or destination operands. Typed source values can either be typed source registers or integer literals. Typed destination values are defined as typed destination registers, ρ_p or ρ_{2p}. ρ_p (resp. ρ_{2p}) represents the case where a single-word (resp. double-word) return value is ignored after a method invocation. This is summarized in Figure 4.9, in which τ is any type from the lattice in Figure 4.8.

Tyde only defines proper instructions, i.e., no pseudo-instructions are used. Also, instead of constant pool indices and most inline numeric literals, Tyde directly uses Java

```
1  public double m3(int a) {
2    switch (a) {
3      case 0:
4        return 1.0;
5      case 1:
6        return 2.5;
7      default:
8        return 4.0;
9    }
10 }
```

(a) Source code.

```
public double m3(int);
0:  packed-switch v3, 12 // +12
3:  const-wide/high16 v0, #long 4616189...
5:  return-wide v0
6:  const-wide/high16 v0, #long 4607182...
8:  goto 5 // -3
9:  const-wide/high16 v0, #long 4612811...
11: goto 5 // -6
12: packed-switch-payload
      entries: 2
      first key: 0
      targets: 6, 9
```

(b) Dalvik bytecode.

```
public double m3(int);
0:  packed-switch (3, int, δs)
      first key: 0 - targets: 6, 9 - default: 3
3:  const-wide/high16 (0, double, δd), #double 4.0
5:  return-wide (0, double, δs)
6:  const-wide/high16 (0, double, δd), #double 1.0
8:  goto 5 // -3
9:  const-wide/high16 (0, double, δd), #double 2.5
11: goto 5 // -6
```

(c) Tyde representation.

Figure 4.7. Method `m3`.

constants. Note that while several constant types are used, we use a generic constant C for ease of exposition. Finally, Tyde does not use offsets to refer to other instructions to represent branches. Instead, Tyde instructions use pointers to other instructions, represented as $ptr_{\mathcal{T}}$.

Table 4.2 shows the syntax of Tyde instructions. There are 9 formats, each of which is later translated to Java using a single rule (see Section 4.6). $\{A\}_a^b$ indicates that symbol A is repeated between a and b times. We note $\{A\}_a = \{A\}_a^a$ and $[A] = \{A\}_0^1$. Finally, $\{A\}$ indicates that A is repeated zero or more times.

$\mathcal{T}_{uo}$ represents instructions which have zero or one typed destination value v_d, zero or more source values v_s, and zero or one Java constant C. Finally, their opcode has an unambiguous semantic equivalent in the JVM. Examples include a vast majority of unary and binary operators and method invocations. $\mathcal{T}_{ao}$ is almost the same format: the only difference is that the corresponding Java opcode is ambiguous. For example, `return-wide` is included in that format because it can be used to return a long (`lreturn`

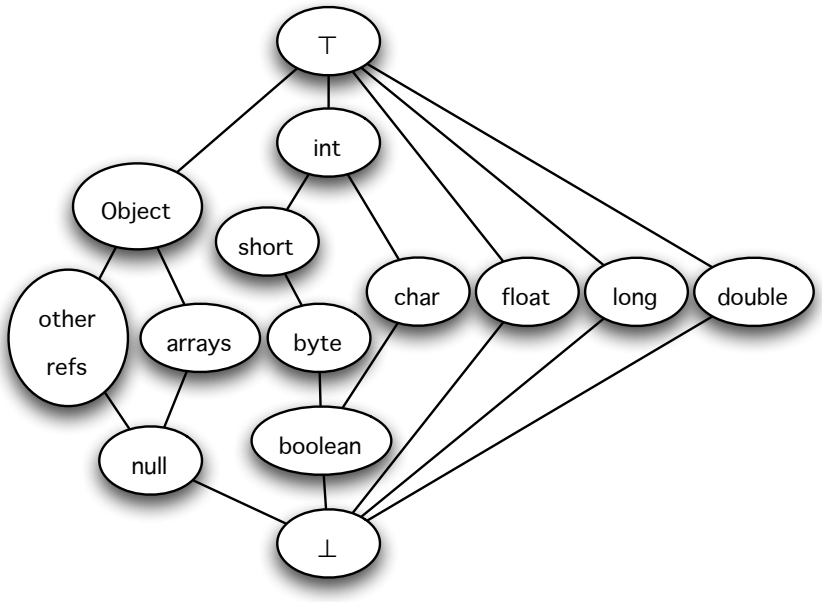


Figure 4.8. Tyde type lattice.

Name	**Syntax**	**Dalvik Instructions**
$\mathcal{T}_{uo}$	$\mathcal{O}_{uo}$, $[v_d]$, $\{v_s\}$, $[C]$	222
$\mathcal{T}_{ao}$	$\mathcal{O}_{ao}$, $[v_d]$, $\{v_s\}$, $[C]$	12
$\mathcal{T}_{ub}$	$\mathcal{O}_{ub}$, $\{\rho_s\}_0^2$, $ptr_{\mathcal{T}}$	11
$\mathcal{T}_{ab}$	$\mathcal{O}_{ab}$, $\{\rho_s\}_1^2$, $ptr_{\mathcal{T}}$	4
$\mathcal{T}_{fna}$	$\mathcal{O}_{fna}$, ρ_d, $\{\rho_s\}$, τ	3
$\mathcal{T}_{no}$	$\mathcal{O}_{no}$, ρ_d, ρ_s	2
$\mathcal{T}_{fad}$	`fill-array-data`, ρ_s, $\{C\}$	1
$\mathcal{T}_{ps}$	`packed-switch`, ρ_s, l, $ptr_{\mathcal{T}}$, $\{ptr_{\mathcal{T}}\}$	1
$\mathcal{T}_{ss}$	`sparse-switch`, ρ_s, $\{l\}_m$, $ptr_{\mathcal{T}}$, $\{ptr_{\mathcal{T}}\}_m$	1

Table 4.2. Simplified syntax of Tyde Instructions.

Register Indices	$r ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{...}$
Typed Source Registers	$\rho_s ::= (r, \tau, \delta_s)$
Typed Dest. Registers	$\rho_d ::= (r, \tau, \delta_d)$
Typed Registers	$\rho ::= \rho_s \mid \rho_d$
Integer Literals	$l_i ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{...} \mid \mathbf{-1} \mid \mathbf{-2} \mid \mathbf{...}$
Typed Source Values	$v_s ::= \rho_s \mid l_i$
Typed Destination Values	$v_d ::= \rho_d \mid \rho_p \mid \rho_{2p}$

Figure 4.9. Tyde typed registers and values.

in Java) or a double (`dreturn` in Java). Opcode set $\mathcal{O}_{uo}$ is partially shown in the first column of Table 4.4; the complete list for $\mathcal{O}_{uo}$ is available in Appendix 8.5. Set $\mathcal{O}_{ao}$ is shown in Table 4.5 (first column).

$\mathcal{T}_{ub}$ represents branching instructions whose opcode have an unambiguous semantically equivalent Java opcode. In addition to an opcode, they are composed of zero, one or two typed source registers ρ_s and a pointer to a target Tyde instruction $ptr_{\mathcal{T}}$. $\mathcal{T}_{ab}$ is similar: the two differences are the number of source registers and the fact that the corresponding Java opcode is ambiguous. Sets $\mathcal{O}_{ub}$ and $\mathcal{O}_{ab}$ are shown in Tables 4.6 and 4.7 (first column).

$\mathcal{T}_{fna}$ instructions are the `filled-new-array` instructions. They are used to create a new array and fill it with the contents of registers. In addition to their opcode, they are composed of a destination register ρ_d, an arbitrary number of sources registers ρ_s and a type τ. Set $\mathcal{O}_{fna}$ is { `filled-new-array`, `filled-new-array/range`, `filled-new-array/jumbo` }. $\mathcal{T}_{fad}$ instructions are `fill-array-data` instructions (presented in Section 4.1). In addition to their opcode and a typed source register ρ_s, they have an arbitrary number of Java constants.

$\mathcal{T}_{no}$ represents the `not-int` and `not-long` unary operators defined in the DVM, which do not have a trivial semantically equivalent opcode in the JVM. Set $\mathcal{O}_{no}$ is { `not-int`, `not-long` }.

$\mathcal{T}_{ps}$ are the `packed-switch` instructions. They are composed of a `packed-switch` opcode, a typed source register ρ_s, an integer literal l_i (switch lowest case value) and a strictly positive number of pointers to Tyde instructions $ptr_{\mathcal{T}}$ (switch targets, including the default case handler). $\mathcal{T}_{ss}$ are the `sparse-switch` instructions. They are composed of a `sparse-switch` opcode, a typed source register ρ_s, m integer literals l_i (switch case values) and $m+1$ pointers to Tyde instructions $ptr_{\mathcal{T}}$ (switch targets, including the default case handler), for some integer m.

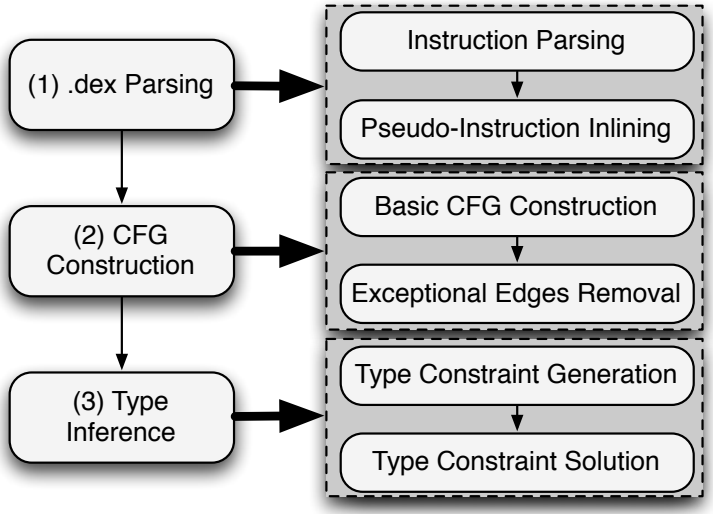


Figure 4.10. Tyde intermediate representation construction overview.

Note that with this representation, not all instructions correspond to valid instructions. For example, with an `add-int` instruction in $\mathcal{T}_{uo}$, there would only be two source values, even though the format accepts an arbitrary number of source values. The third column of Table 4.2 indicates how many types of Dalvik instructions are mapped to each Tyde instruction format.

4.5 Transforming Dalvik Bytecode to Tyde

As depicted in Figure 4.10, Tyde IR is generated in three steps. The first step consists of parsing the `.dex` file, which involves parsing data related to classes and methods (e.g., access flags and names), as well as fields and method instructions. In the next step, a Control Flow Graph (CFG) is constructed. In the final step, a type inferencing algorithm infers types of registers that are ambiguous.

Parsing – Since parsing class and method data and fields is straightforward, we focus our description on instruction parsing. While parsing instructions, type information for registers is determined. For example, the types of several unary and binary operators can be known from their opcode, e.g., an `add-long` instruction takes two long integers as sources and a long integer as destination. Also, during this parsing step, for every instruction which uses a constant pool reference, a new Java constant is generated on the fly. The only exception is when the instruction is an ambiguous numeric constant assignment. In that case, type information is needed before the constant can be created.

The parsing step also removes pseudo-instructions. As mentioned before, Dalvik bytecode uses pseudo-instructions to store complementary data about a proper instruction. In the parsing step, we simply store the contents of each pseudo-instruction in the proper instruction which refers to it and eliminate the pseudo-instruction.

4.5.1 Building a Control Flow Graph

In the second step, we build a CFG from the Dalvik bytecode. Techniques used for constructing CFGs are quite standard, so we focus on details that are specific to our system. As required by the Tyde IR, relative offsets in branching instructions are converted to pointers. Also, exception tables are used to generate CFG edges. When an instruction I is protected by a `try` block, we add a CFG edge from its predecessors to the appropriate exception handlers. Standard CFG construction often also includes edges from I to exception handlers if I has side effects. This is done in order to account for the case in which the side effects are committed before the exception is thrown. For our purposes, we do not include these edges. They are not included by the Dalvik or Java verifiers: even though I might commit side effects before throwing an exception, the type state will not be changed. This entire process is quite straightforward, since all relative offsets are statically known.

4.5.1.1 Removing Unfeasible Exceptional Control Flow Graph Edges

As we previously described, DVM and JVM verifiers differ in the way they handle CFG edges related to exceptions: while the Java verifier considers that all instructions inside a `try` block may throw an exception, the Dalvik verifier only considers edges for instructions which might actually throw an exception. For our CFG construction, we adopt the Dalvik approach. In order to generate verifiable Java bytecode, we also modify the exception tables (which describe `try/catch/finally` blocks) to only include instructions which might throw an exception in `try` blocks. We now explain why these modifications do not modify the semantics of the program.

The Java platform has two types of exceptions [123, §2.16]. *Synchronous* exceptions occur as a result of the execution of a particular instruction. For example, if an integer division instruction has a divisor with value 0, it will throw an `ArithmeticException` when it is executed. On the other hand, *asynchronous* exceptions can happen at any point in the execution of a method. Asynchronous exceptions only have two possible causes. First, an asynchronous exception can be thrown when the deprecated `Thread.stop()`

method is called (the thread on which it is called will immediately throw an exception). The only other case where an asynchronous exception can be thrown is when an internal error in the virtual machine implementation occurs.

Removing CFG edges for instructions which cannot throw synchronous exceptions does not violate synchronous exception handling semantics; as we know which instructions can throw synchronous exceptions, we just have to make sure to leave them in their `try` blocks. Asynchronous exceptions may seem less trivial, since they might in theory be thrown by any instruction. However, exceptions cannot be thrown asynchronously in the DVM (in particular, `Thread.stop()` is not supported). That is the reason why the Dalvik verifier only considers synchronous exceptions in its CFG construction. Therefore, removing CFG edges corresponding to asynchronous exception handling does not modify the semantics of the program.

4.5.2 Type Inference

The problem we are trying to solve is the following: given a method with typing following the lattice on Figure 4.1, find a typing which is valid with respect to the lattice on Figure 4.8.

We defer formal proofs to future work, but the type inference algorithm we present below was able to find a valid Java typing for all 1.6 million retargeted methods from our sample. There are several reasons why our algorithm works well in practice. The type system enforced by the Dalvik verifier is quite similar to the type system enforced by the Java verifier. For example, consider a register assignment which might be for a float or an int. If it is used as a float, it cannot be subsequently used as an int on the same branch. However, the Dalvik verifier does not back-propagate unambiguous use type information to ambiguous assignments. Therefore, if an ambiguous assignment (e.g., 32-bit constant) reaches two uses with incompatible types through two different branches (e.g., int on one branch and float on another branch), the code could still be Dalvik-verifiable even though it does not have a valid Java typing. However, we did not find any occurrence of this, most likely because Dalvik code is compiled from Java code and the analysis required to merge registers with different types but identical bit patterns would be costly and provide very minimal gain.

Type inference for Dalvik bytecode uses the following approach: First we generate constraints on types based on definitions and uses. These constraints are then solved to infer unknown types. Note that our goal is not to determine types for all variables, unlike previous work [26]. In particular, with the exception of array types we do not

need to know precise types for references.

In this section, $\tau_c(v_i, I_j)$ and $\tau_v(v_i, I_j)$ denotes the type of register v_i at instruction I_j. We use τ_c to denote a type constant and τ_v to denote a type variable.

4.5.2.1 Constraint Generation

Constraints are generated by traversing the CFG starting at ambiguous register definitions (respectively uses), looking for uses (respectively assignments) of the same register. For example, let us consider method `m2`, defined in Figure 4.6. Its CFG is shown in Figure 4.6(d). Method `m2` generates the following constraints:

$$\tau_v(v_0, I_2) \leq \text{double}$$

$$\tau_v(v_0, I_5) \leq \text{double}$$

$$\text{int} \leq \tau_v(v_3, I_0)$$

It has two ambiguous definitions (the two `const-wide/high16` instructions) and one ambiguous use (the `if-eqz` instruction). The first two constraints are generated because ambiguous definitions reach instruction `return-wide`, whose type is known from the method signature. The last constraint (on instruction I_0) is given by the method signature, according to which register v_3 is assigned an integer argument before method execution starts.

The constraints related to variable definitions and uses induce inequality constraints. Instructions dealing with arrays introduce another kind of constraint on their operands. For example, consider instruction `aget` v_0, v_1, v_2, which loads element at index v_2 from the array referenced by v_1 into v_0. If it is determined that the array referenced by v_1 is an array of integers, then v_0 is an integer. Eventually, we obtain four types of constraints: type (1) $\tau_c \leq \tau_v$, type (2) $\tau_v \leq \tau_c$, type (3) $\tau_{v_1} \leq \tau_{v_2}$ and type (4) $\tau_{v_1} = [\tau_{v_2}$ (where $[\tau_{v_2}$ means "array of τ_{v_2}").

4.5.2.2 Constraint Solution

Solving these constraints is performed in three phases. Throughout these three phases, whenever a type variable involved in a type (4) constraint is determined, the other side of the type (4) equality is also determined (or checked if it is already known).

Phase 1 – First, we use Algorithm D by Rehof and Mogensen [124] to find the least solution to type (1) and (3) constraints. The Rehof-Mogensen (RM) algorithm finds the least solution τ. The RM-algorithm also checks that the solution τ satisfies type (2)

constraints. For the constraint system shown earlier, the RM-algorithm finds the solution $\tau_v(v_3, I_0) = \text{int}$. However, we still need to find values for $\tau_v(v_0, I_2)$ and $\tau_v(v_0, I_5)$, which is handled in phase 2.

Phase 2 – After finishing phase 1, there are variables whose type is $\perp$ (unknown). We need to infer the types of these variables. In our example, after phase 1 $\tau_v(v_0, I_2)$ and $\tau_v(v_0, I_5)$ are $\perp$. Phase 2 finds the types of these variables. Assume that type variable τ_v has value $\perp$ after phase 1, but has the following constraints of type (2): $\tau_v \leq \tau_{c_1}, \tau_v \leq \tau_{c_2}, \cdots, \tau_v \leq \tau_{c_k}$. Then we set $\tau_v = \tau_{c_1} \wedge \tau_{c_2} \wedge \cdots \wedge \tau_{c_k}$ (that is, the greatest common subtype of $\tau_{c_1}, \tau_{c_2}, \cdots, \tau_{c_k}$). This ensures that all type (2) inequalities involving τ_v are satisfied. For our example, this yields the following types: $\tau_v(v_0, I_2) = \text{double}$ and $\tau_v(v_0, I_5) = \text{double}$. In general, this process may determine the type of the left-hand side of some type (3) inequalities. In order to solve these constraints properly, we run the algorithm by Rehof and Mogensen again.

Phase 3 – After phase 1 and 2 there are some types that are still undetermined (e.g., ambiguous assignment reaching only an ambiguous use, which is not itself reached by any unambiguous assignment). In that case, we set these types to safe default types. In Section 4.1, we explained that all instructions with ambiguous typing have a limited set of types that they can possibly take. For example, a `const-wide` instruction can only take types long or double. After this, all variable types are safe Java types.

4.6 Generating Java Bytecode

A Dalvik code in Tyde IR is translated into Java bytecode in three steps. In the first step, registers are mapped to Java local variables and labels are generated to support control-flow instructions. In the second step, instructions in Tyde IR are converted to Jasmin instructions (Jasmin is a Java bytecode assembler). The third step is to use Jasmin to generate Java `.class` bytecode. We will provide a brief description of the first two steps.

4.6.1 First Step (Pre-Processing)

Register Mapping – Tyde is a register-based representation, therefore we need to map every register to a Java local variable. A *register map* m is a function that maps a Tyde IR register to a Java local variable. In the JVM, the first local variable indices are reserved for the `this` reference (for non-static methods) and the method arguments [123,

Typed Value	**Java Instruction**
l_i	sipush l_i
(r, τ_i, δ_s)	iload $m(r)$
$(r, \text{float}, \delta_s)$	fload $m(r)$
$(r, \text{long}, \delta_s)$	lload $m(r)$
$(r, \text{double}, \delta_s)$	dload $m(r)$
(r, τ_r, δ_s)	aload $m(r)$
ρ_p	pop
ρ_{2p}	pop2
(r, τ_i, δ_d)	istore $m(r)$
$(r, \text{float}, \delta_d)$	fstore $m(r)$
$(r, \text{long}, \delta_d)$	lstore $m(r)$
$(r, \text{double}, \delta_d)$	dstore $m(r)$
(r, τ_r, δ_d)	astore $m(r)$

Table 4.3. Opcode map f_{ds} for typed values.

§3.6.1]. In the DVM, these variables use the last register indices, so our register map has to respect this constraint.

Label Generation – In Java bytecode, branching instructions include relative offsets to their targets. Also, exceptions tables describe the boundaries of `try/catch/finally` blocks using absolute offsets. However, Jasmin uses labels for these, which is why we need to generate labels before we can generate Jasmin code. We define b such that, for any pointer $ptr_{\mathcal{T}}$ to a Tyde instruction, $b(ptr_{\mathcal{T}})$ is the label of the instruction corresponding to $ptr_{\mathcal{T}}$.

4.6.2 Second Step (Translating Instructions)

In order to map a Tyde method to a Java method, we introduce helper functions. The general idea is that each Tyde instruction format is mapped to Java instructions using a single pattern. In this section, we use the symbol $\mathcal{J}_\epsilon$ to represent the "null" Java instruction (which in case of Jasmin is the empty string).

Typed Values – In Table 4.3, we define a function f_{ds} which maps typed values to Java instructions. For this definition, we use the register map m. τ_i is any single-word integer type (boolean, char, byte, short or int) and τ_r is any reference (array or non-array)

Tyde Opcode	Java Opcode
nop	nop
move	$\mathcal{J}_\epsilon$
const-wide/16	ldc2_w
monitor-exit	monitorexit

Table 4.4. Opcode map f_{uo} for set $\mathcal{O}_{uo}$ (partial definition).

type. Essentially $f_{ds}(v)$ corresponds to the Java instruction to load or store the Tyde typed value v. These Java instructions can be divided into two categories: *a)* pushing values onto the operand stack before an operation is applied to them (`load` instructions), and *b)* popping values resulting from an operation from the stack, either to store them in a local variable (`store` instructions) or to balance the stack if a method's return value is discarded (`pop` instructions). Since these values are typed and their register use (source or destination) is known, they can be translated independently from other parts of the Tyde instruction, i.e., knowing the Dalvik opcode is not necessary. One of the advantages of having types on sources and destinations in Tyde is that Java instructions to load/store/pop typed values can be generated independently of the instruction opcode.

Constants – We use $f_{\mathcal{C}}$ to represent the translation of a constant pool reference. If we were generating binary bytecode, it would simply be the index of the constant in the constant pool. Since we generate instructions for Jasmin, it is a textual description of the constant (e.g., value for an integer constant). If there is no constant, $f_{\mathcal{C}}$ simply returns an empty string.

Opcodes – In general, unambiguous opcodes in Tyde IR have a corresponding opcode in Jasmin. For ambiguous opcodes, the types of the operands in Tyde IR are used to determine the corresponding opcode in Jasmin. In Table 4.4, we define a function f_{uo} which maps unambiguous Tyde operators to semantically equivalent Java opcodes. A vast majority of Dalvik opcodes (222 out of 257) are in this class $\mathcal{O}_{uo}$. Their semantic mapping to a Java opcode is trivial and the equivalent Java opcode can be known by only knowing the Dalvik opcode. Table 4.4 only shows a subset of the mappings. The complete definition is available in Appendix 8.5. In Table 4.5, we define a function f_{ao} which maps ambiguous opcodes and Tyde types to Java opcodes. τ_p is any primitive type, i.e. τ_i, float, long or double. In these cases, one Dalvik opcode maps to several Java opcodes and thus an argument type is needed to disambiguate the mapping.

In Table 4.6, we define a function f_{ub} which maps unambiguous branching opcodes

Tyde		Java
Opcode	**Type**	**Opcode**
return	τ_i	ireturn
	float	freturn
return-wide	long	lreturn
	double	dreturn
const/4, const/16, const, const/high16	τ_i, float	ldc
	τ_r	aconst_null
aget	int	iaload
	float	faload
aget-wide	long	laload
	double	daload
aput	int	iastore
	float	fastore
aput-wide	long	lastore
	double	dastore
new-array, new-array/jumbo	τ_p	newarray
	τ_r	anewarray

Table 4.5. Opcode map f_{ao} for set $\mathcal{O}_{ao}$.

Tyde Opcode	**Java Opcode**
goto, goto/16, goto/32	goto
if-lt	if-icmplt
if-ge	if-icmpge
if-gt	if-icmpgt
if-le	if-icmple
if-ltz	iflt
if-gez	ifge
if-gtz	ifgt
if-lez	ifle

Table 4.6. Opcode map f_{ub} for set $\mathcal{O}_{ub}$.

Tyde		Java
Opcode	**Type**	**Opcode**
if-eq	τ_i	if_icmpeq
	τ_r	if_acmpeq
if-ne	τ_i	if_icmpne
	τ_r	if_acmpne
if-eqz	τ_i	ifeq
	τ_r	ifnull
if-nez	τ_i	if-icmpnez
	τ_r	ifnonnull

Table 4.7. Opcode map f_{ab} for set $\mathcal{O}_{ab}$.

to semantically equivalent Java opcodes. As with f_{uo}, this mapping is trivial and the equivalent Java opcode is completely determined by the Dalvik opcode. Table 4.7 defines a function f_{ab} which maps ambiguous branching opcodes and Tyde types to Java opcodes. As with $\mathcal{O}_{ao}$, one Dalvik opcode maps to several Java opcodes and an argument type is needed to disambiguate the mapping.

Instructions – Using the register map m and the various functions defined earlier, we can describe the translation for each of the Tyde instruction classes shown earlier. This translation is shown in Table 4.8. In addition to the functions defined earlier, our translation also uses function $f_{xastore}$ for mapping store instructions defined in Table 4.9. We now describe translation rules for a few interesting Tyde instructions.

`not` Instructions $\mathcal{T}_{no}$ – The Dalvik and Tyde instruction sets include `not` instructions for integers. While there is no trivial semantic equivalent in the Java instruction set, we can use a combination of Java instructions and take advantage of the equivalence of bitwise binary operators. Given an integer i, we define $1_{|i|}$ the integer whose bit pattern is all ones with the same width as i (32 or 64 bits). If the NOT (resp. XOR) bitwise operator is represented as $\neg$ (resp. $\oplus$), then we have $\neg i = 1_{|i|} \oplus i$. Therefore, a valid Java instruction pattern consists in pushing constant $1_{|i|}$ onto the stack (with `ldc` or `ldc2_w` depending on integer width). After pushing i onto the stack, the `ixor` (or `lxor`) opcode should be applied. Finally, the result should be popped from the stack. This pattern defines map j_{no} in Table 4.8. For this function, we use function f_{no} defined over $\mathcal{O}_{no}$ such that $f_{no}(\texttt{not-int}) = \texttt{ixor}$ and $f_{no}(\texttt{not-long}) = \texttt{lxor}$. We also define $|o_{no}| = 32$ if

Tyde Instruction Set	**Tyde Instruction**	**Equivalent Java Instructions**
$\mathcal{T}_{uo}$	$t_{uo} = (o_{uo}, v_d, v_s^0, v_s^1, \cdots, v_s^k, C)$	$j_{uo}(t_{uo}) = f_{ds}(v_s^0)\|\|f_{ds}(v_s^1)\|\|\cdots\|\|f_{ds}(v_s^k)\|\|\,(f_{uo}(o_{uo}), f_{\mathcal{C}}(C))\,\|\|f_{ds}(v_d)$
$\mathcal{T}_{ao}$	$t_{ao} = (o_{ao}, v_d, v_s^0, v_s^1, \cdots, v_s^k, C)$	$j_{ao}(t_{ao}) = f_{ds}(v_s^0)\|\|f_{ds}(v_s^1)\|\|\cdots\|\|f_{ds}(v_s^k)\|\|\,(f_{ao}(o_{ao}, \tau), f_{\mathcal{C}}(C))\,\|\|f_{ds}(v_d)$
$\mathcal{T}_{ub}$	$t_{ub} = (o_{ub}, \rho_s^0, \rho_s^1, ptr_{\mathcal{T}})$	$j_{ub}(t_{ub}) = f_{ds}(\rho_s^0)\|\|f_{ds}(\rho_s^1)\|\|\,(f_{ub}(o_{ub}), b(ptr_{\mathcal{T}}))$
$\mathcal{T}_{ab}$	$t_{ab} = (o_{ab}, \rho_s^0, \rho_s^1, ptr_{\mathcal{T}})$	$j_{ab}(t_{ab}) = f_{ds}(\rho_s^0)\|\|f_{ds}(\rho_s^1)\|\|\,(f_{ab}(o_{ab}, \tau), b(ptr_{\mathcal{T}}))$
$\mathcal{T}_{no}$	$t_{no} = (o_{no}, \rho_d, \rho_s)$	$j_{no}(t_{no}) = (\text{ldc/ldc2\_w}, f_{\mathcal{C}}(1_{\|o_{no}\|}))\|\|f_{ds}(\rho_s)\|\|f_{no}(o_{no})\|\|f_{ds}(\rho_d)$
$\mathcal{T}_{fna}$	$t_{fna} = (o_{fna}, \rho_d, \rho_s^0, \rho_s^1, \cdots, , \rho_s^k, \tau)$	$j_{fna}(t_{fna}) = (f_{fna}(\tau), \tau)\|\|\text{dup}\|\|f_{ds}(0)\|\|f_{ds}(\rho_s^0)\|\|f_{xastore}(\tau)\|\|\text{dup}\|\|f_{ds}(1)$ $\|\|f_{ds}(\rho_s^1)\|\|f_{xastore}(\tau)\|\|\cdots\|\|\text{dup}\|\|f_{ds}(k)\|\|f_{ds}(\rho_s^k)\|\|f_{xastore}(\tau)\|\|f_{ds}(\rho_d)$
$\mathcal{T}_{fad}$	$t_{fad} = (\text{fill-array-data}, \rho_s, C_0, C_1,$ $\cdots, C_k)$	$j_{fad}(t_{fad}) = f_{ds}(\rho_s)\|\|f_{ds}(0)\|\|(\text{ldc/ldc2\_w}, f_{\mathcal{C}}(C_0))\|\|f_{xastore}(\tau(\rho_s))\|\|f_{ds}(\rho_s)$ $\|\|f_{ds}(1)\|\|(\text{ldc/ldc2\_w}, f_{\mathcal{C}}(C_1))\|\|f_{xastore}(\tau(\rho_s))\|\|\cdots\|\|f_{ds}(\rho_s)\|\|f_{ds}(k)$ $\|\|(\text{ldc/ldc2\_w}, f_{\mathcal{C}}(C_k))\|\|f_{xastore}(\tau(\rho_s))$
$\mathcal{T}_{ps}$	$t_{ps} = (\text{packed-switch}, \rho_s, l,$ $ptr_{\mathcal{T}}^{default}, ptr_{\mathcal{T}}^0, ptr_{\mathcal{T}}^1, \cdots, ptr_{\mathcal{T}}^k)$	$j_{ps}(t_{ps}) = f_{ds}(\rho_s)\|\|(\text{tableswitch}, l, b(ptr_{\mathcal{T}}^{default}), b(ptr_{\mathcal{T}}^0), b(ptr_{\mathcal{T}}^1), \cdots,$ $b(ptr_{\mathcal{T}}^k))$
$\mathcal{T}_{ss}$	$t_{ss} = (\text{sparse-switch}, \rho_s, l_1, l_2, \cdots,$ $l_k, ptr_{\mathcal{T}}^{default}, ptr_{\mathcal{T}}^1, ptr_{\mathcal{T}}^2, \cdots, ptr_{\mathcal{T}}^k)$	$j_{ss}(t_{ss}) = f_{ds}(\rho_s)\|\|(\text{lookupswitch}, b(ptr_{\mathcal{T}}^{default}), l_1, b(ptr_{\mathcal{T}}^1), l_2, b(ptr_{\mathcal{T}}^2), \cdots,$ $l_k, b(ptr_{\mathcal{T}}^k))$

Table 4.8. Tyde maps.

Tyde Type	Java Opcode
[boolean, [byte	bastore
[char	castore
[short	sastore
[int	iastore
[float	fastore
[τ_r	aastore

Table 4.9. Map $f_{xastore}$.

o_{no} = `not-int` and $|o_{no}| = 64$ if o_{no} = `not-long`.

`filled-new-array` **Instructions** $\mathcal{T}_{fna}$ – Dalvik and Tyde bytecode have instructions which create a new array and fill it with the contents of registers given as arguments. While Java does not have a direct equivalent, a semantically equivalent sequence of Java instructions is as follows. First, a `newarray` (primitive type) or `anewarray` (reference type) instruction will create a new array with the appropriate type and return a reference to the array on the stack. To fill the array with the proper values, we use a sequence of the following pattern: *i)* a `dup` instruction duplicates the array reference on the stack, then *ii)* the proper array index $l \in \mathcal{L}_i$ is pushed onto the stack, next *iii)* the array element value is pushed onto the stack, then *d)* an appropriate `xastore` instruction pops the duplicated reference, the index and the element and performs the array storage. Finally, an `astore` instruction stores the array reference to a local variable. This patterns defines map j_{fna} in Table 4.8. For this function, we use function f_{fna} such that $f_{fna}(\tau_p)$ = `newarray` and $f_{fna}(\tau_r)$ = `anewarray`.

`packed-switch` **Instructions** $\mathcal{T}_{ps}$ – The semantic equivalent of a Tyde `packed-switch` instruction is a Jasmin `tableswitch` instruction. The arguments of a `packed-switch` are a typed source register (integer used to switch), a literal corresponding to the lowest case value, a pointer to the default case handler and a set of pointers to case handlers. The corresponding Jasmin instruction is similar, except that it uses labels instead of pointers.

Example – Let us consider method `m2` introduced in Figure 4.6(a). Figure 4.6(e) shows its Tyde representation. For ease of exposition, we assume that the label of each instruction is simply its original offset and the register mapping m is such that $m(0) = 2$, $m(1) = 3$, $m(2) = 0$ and $m(3) = 1$. The first Tyde instruction has format $\mathcal{T}_{ab}$

(see Table 4.8). Using the notation from Table 4.8, o_{ab} is `if-eqz`, $\rho_s^0 = (3, \text{int}, \delta_s)$, ρ_s^1 is empty and $ptr_{\mathcal{T}}$ is a pointer to instruction at offset 5. $f_{ds}(3, \text{int}, \delta_s) =$ `iload` $m(3) =$ `iload` 1 is the first Java instruction. Then we have $f_{ds}(\rho_s^1) = \mathcal{J}_\epsilon$, $f_{ab}(o_{ab}, \text{int}) =$ `ifeq` and $b(ptr_{\mathcal{T}}) = 5$ (note that the 5 value is just a label and not the offset in the final Java bytecode). The second Java instruction is `ifeq`, label 5. The remaining Tyde instructions are translated in a similar manner to obtain the bytecode shown on Figure 4.6(f).

4.7 Unverifiable Dalvik Bytecode

In the previous sections, we have described how to generate verifiable Java bytecode from verifiable Dalvik bytecode. As we explore below, some bytecode from real-world applications is not Dalvik-verifiable. In this section, we show the errors we encountered with real bytecode. We also show how we modified our retargeting process to handle bytecode that is not Dalvik-verifiable in order to generate verifiable Java bytecode.

4.7.1 Observed Errors

Improper references – The main source of unverifiable bytecode is the presence of bad method, field, interface or class references. Two different cases were encountered:

- References to classes which are not available within the application or in the core Android classes. A special case is when the superclass of a class is missing; then the class is trivially unverifiable and is not even linked by the DVM.
- References to methods or fields which are non-existent or not accessible (e.g., private member).

There are two reasons for these missing references. The first reason is that applications commonly use private Android APIs. The Android platform includes public APIs which are documented and always backward compatible. In other words, an application using only public APIs will still work after an Operating System (OS) update. Android also includes private APIs which are meant for internal use by OS components. Unlike public APIs, private ones are not officially documented and backward compatibility is not guaranteed. Using them is strongly discouraged, as a simple OS update may break an application making calls to private APIs. In our experiments, we checked verifiability using recent Android core classes whereas the application sample was about a year older. Doing so allowed us to point out potential problems applications could have after an OS update.

The second reason is that, as we pointed out in previous work [107], developers often include entire libraries to be able to use some classes from these libraries. Parts of the included libraries sometimes make calls to other libraries, which are not themselves included with the Android application. In practice, it is not an issue, as long as these parts of the included library are not used anywhere in the application. However, the unused part of the library code making these calls will not be verifiable.

Typing and other issues – The second source of unverifiability for Dalvik bytecode is the presence of invalid typing. Other issues are a marginal cause of verification problems. These can have a very wide variety of causes (e.g., malformed class or member identifier, illegal access flag, etc.). More analysis is needed to understand how these problems can make their way to released Dalvik bytecode.

4.7.2 Handling Unverifiable Dalvik Bytecode

In this section, we describe our approach to handle unverifiable Dalvik bytecode in an application using Dalvik pre-verification. First, we verify the application using the Dalvik verifier which is part of the Android OS. We modified it in order to make it generate a detailed report describing all verifiability issues with the application. For each problem in the application, the report describes the class name, the method name and signature, the problematic code offset and the type of verification error.

Then, the report is input into Dare with the application package. The Dare parsing step (see Section 4.5) is modified as follows:

1. If the entire class could not be linked by the verifier (e.g., because its superclass is missing), then skip the entire class. None of the code of the class will be retargeted. At runtime, the class would also be missing, so not retargeting it does not change the semantics of the program.

2. If a method is entirely unverifiable because of a serious issue (e.g., typing issue), then the parsing step replaces the method with code that throws a `VerifyError`. All other verifiable methods in the class will be retargeted without modification.

3. If a method is unverifiable because of a less serious issue (e.g., missing class reference), then only replace the faulty code location with code that throws an appropriate error (for example, `NoClassDefFoundError`). That is exactly the behavior of the Dalvik VM at runtime: the faulty code locations are also rewritten during the verification process. An interesting case is when we encounter a reference to a

class that was ignored in Step 1. The reference is then replaced with code throwing a `NoClassDefFoundError`, which is the runtime behavior.

These code transformations serve two important purposes:

- They mirror the runtime behavior of the program and therefore do not cause any loss of semantics.
- They make the entire program Dalvik-verifiable.

In order to account for some subtle differences between the Java and Dalvik verifiers, we also had to consider two cases where a method was Dalvik-verifiable but it was not Java-verifiable after retargeting. The first difference involves `aget-object` instructions, which are used to access a component in an array of references. If it is used with an array reference which is known to always be null at verification time, then the verifier sets the array component to be null as well. But as we described above, in Dalvik, null and int/float with value 0 are the same type. Thus, if the register is subsequently used as an int or a float, the code will be Dalvik-verifiable, but no valid Java typing will exist for it. In order to fix this, we replace the `aget-object` instruction with code throwing a `NullPointerException`. This mirrors the runtime behavior of the program and also allows us to find a Java typing (since the null register will no longer be used as an int or a float).

Second, when a field with reference type is accessed in the Dalvik architecture, the method accessing it can still be verifiable even if the field type cannot be resolved (the type of the register storing the field is set to `java.lang.Object`). The method is verifiable only if the register is subsequently used in trivial ways. On the other hand, the Java verifier will reject a method if a field reference type cannot be resolved. In order to make the retargeted code verifiable, we had Dare automatically generate class stubs for the unresolvable field types. Since those fields are only used in trivial ways, these stub classes do not need to contain any field or method.

4.8 Evaluation

4.8.1 Dalvik Bytecode Verification

In this section, we describe the Dalvik verification issues we found in a sample of 1,100 applications. We use the 50 most popular applications in the 22 application categories

as of September 1, 2010[4]. The 1,100 applications contained 262,110 classes. We used the Dalvik verifier and the core Android classes included in Android version 4.0.3.0.2.0.1.0.

A surprising result of our Dalvik verification experiments is that 247 applications contained unverifiable code. 181 applications contained at least one class which could not be linked by the Dalvik VM (e.g., because of a missing superclass), totalling 905 trivially unverifiable classes. 214 applications had at least one unverifiable code location in non-trivially unverifiable classes. Table 4.10 presents the results of the Dalvik verification process for the classes which were not completely Dalvik-unverifiable. For each issue, we show the number of faulty code locations and the number of applications with at least one bad location. Occasionally, several locations in a single method can be unverifiable.

The first column shows the number of bad references, i.e. references to an inaccessible (e.g., private) or nonexistent class member. It is unlikely for an application to make a bad reference to its own code, so these are most likely references to private Android APIs which were modified after the application was developed. The second column shows the number of references to a missing class. These are caused by unlinkable application classes and by (supposedly unused) code in included libraries making references to other libraries which are not included with the application. Finally, we show the number of typing and other issues, which account for 6.46% of all unverifiable code.

4.8.2 Retargeting

The empirical evaluation described in this section attempts to answer two central questions: 1) are the computational costs of retargeting feasible in practice? and 2) can Dare successfully retarget market applications? The answers to these questions will determine the degree to which this is a useful tool for extracting code for further analysis. Highlights of the study include:

- After some additional code optimizations in some isolated cases, the output of Dare is verifiable for all methods for 99.64% of the applications in the corpus, and over 99.999% of methods overall. This is a substantial increase over existing tools.

- Dare can, on average, retarget each class in 4.20 msec, and was able to retarget the entire corpus of 1,100 applications containing over 260,000 classes in less than 20 min.

- The complete processing of all applications including Dalvik pre-verification (modified Dalvik verifier), retargeting (Dare) and assembly (Jasmin) took less than 70

[4]Experiments run on a smaller sample from March 2012 show near-identical results.

	Bad References	Missing References	Typing Issue	Other
Applications	93	168	73	13
Code Locations	1,335	6,413	488	54

Table 4.10. Verification results for partially verifiable classes.

compute-minutes.

We compare Dare against `dex2jar` [125], the most popular tool for retargeting Dalvik bytecode to Java. We do not report bytecode verification results for our previous tool: `ded` [121, 126] was built for the purpose of decompilation and did not include all information that is required for a verifiable class file, for example the maximum stack size for each method (which was set to a default value of 0). As a consequence, while the output of `ded` can typically be processed by decompilers and accurately captures the semantics of the original program, it is generally trivially unverifiable.

We evaluate Dare on two key metrics: performance and retargeting success rate. We retargeted the entire corpus of applications described in the previous section. We used Jasmin version 2.4.0 for Java bytecode assembly.

Performance – The total processing time was 4,198 seconds, with Dalvik pre-verification consuming 229 seconds (5.45%), Dare 1,101 seconds (26.23%) and Jasmin 2,868 seconds (68.32%). Dare processing was dominated by the file output operations. They are performed at the same time as the translation of Tyde to Jasmin. Together they take 85% of the total processing time. The type inference algorithm accounts for 5% of the total processing time. Other parts of the retargeting process take less than 5% each. Retargeting is efficient and can be a fast first step before application analysis.

Retargeting – The success metrics reported below measure the ability of Dare to generate valid bytecode. A method is said to be successfully retargeted when Dare generates bytecode that is verifiable (and thereafter is ready for inspection and analysis by existing tools). A class is said to be successfully retargeted if all the methods it contains are successfully retargeted. Finally, an application is said to be successfully retargeted if all classes within the application are retargeted. For the Java bytecode verification experiments, we used the Oracle Labs Maxine VM verifier [127].

Table 4.11 shows the results of the Dare retargeting. The first column shows the total number of classes and non-abstract methods in our sample. The second column shows how many classes were safely removed using the Dalvik verification reports as described in Section 4.7 (the application count is the number of applications in which at least one class was removed). The next column presents the number of classes and methods which were modified following the Dalvik verification reports. The next column shows the number of retargeted classes which were completely unverifiable (the application count is the number of applications in which at least one class was completely unverifiable). For each of these classes, the issue was caused by a single method which had a code size

	Total	Removed Code	Modified Code	Unverifiable Classes	Unverifiable Code	Verifiable Code (%)
Applications	1,100	181	214	7	3	99.09%
Classes	262,110	905	3,354	14	4	99.99%
Methods	1,620,813	9,658	6,858	100	4	99.99%

Table 4.11. Dare retargeting success rates.

	Total	Completely Unverifiable Classes	Unverifiable Code	Verifiable Code (%)
Applications	1,100	422	206	59.64%
Classes	262,110	1,405	776	99.17%
Methods	1,620,813	25,972	1272	98.32%

Table 4.12. `dex2jar` retargeting success rates.

over the maximum allowed size of 65536 bytes. We were able to fix 13 of these 14 issues by running the Soot optimizations on these classes. Only one of these 14 failures could not be fixed: the bytecode optimizations did not sufficiently reduce the code size. After optimization, only one of the 100 methods was not verifiable.

The next column shows the number of methods which had an issue that did not cause Maxine to reject the entire class but only a single method. One of these failures was caused by a reference to one of the classes which was not verifiable because of code size, as described above. It was fixed after the code optimizations reduced the code size and made the referenced class verifiable. The other 3 failures were related to a pathological difference between the Dalvik and Java verifiers. In the case of Java, when a method in a class `C` tries to access a protected method from a superclass `D` which is in a different package, it can only do so if the instance on which the method invocation occurs is an instance of a subclass of `C`. The Dalvik verifier, however, does not enforce this rule and only checks that `C` is a subclass of `D`. As a consequence, the Dalvik verification step accepted the 3 methods, which were subsequently rejected by the Java verifier after retargeting. Since a Java compiler would not generate code with this issue, the issue is most likely due to a private API method which was public when the applications were created and was later changed to be protected (all 3 failures occurred in a wrapper for the same Android API class and involve a call to the same protected method).

The final column shows the overall success rates as a percentage of the retargeted code. While we do not implement a solution for the 4 failures which did not have a trivial fix, we do not consider them to be significant. They only represent less than 0.00025% of the methods in our sample. Moreover, we were able to check that the issues with the code in these 4 methods do not necessarily prevent them from being processed by analysis tools: all 4 were successfully optimized by Soot. Typing problems, on the other hand, would prevent any serious analysis. No type issue was found by the Java verifier, which strongly validates our type inference algorithm.

Table 4.12 shows the retargeting results using the latest version of `dex2jar` (0.0.9.8), currently the most widely used retargeting tool. There are two main reasons why `dex2jar` performs less well at the retargeting experiments than Dare. First, it does not handle unverifiable Dalvik bytecode: the result of retargeting unverifiable Dalvik bytecode is unverifiable Java bytecode. The second reason is that, similarly to `ded`, `dex2jar` aims to be used for decompilation and typically decompilers can decompile unverifiable code if the cause for unverifiability is not too serious. Verifiability is a stronger criterion for success and ensures that the application is processed by analysis tools (and not only decompilers). In the case of `dex2jar`, a number of classes are completely unverifiable for trivial reasons (e.g., illegal member access flags). In addition, several methods are unverifiable for various reasons ranging from bad references to illegal typing. As a result, even though the class and method retargeting success rates are high (respectively over 99% and over 98%), less than 60% of applications are completely verifiable. It is a serious obstacle to whole-program analysis.

Chapter 5

Analysis of Inter-Component Communication in Android with *Epicc*

Past analyses of Android applications [107, 3, 51, 5, 77, 52] have largely focused on analyzing application components in isolation. Recent works have attempted to expose and analyze the interfaces provided by components to interact [4, 107], but have done so in ad hoc and imprecise ways.

Conversely, this chapter attempts to formally recast Inter-Component Communication (ICC) analysis to infer the locations and substance of all inter- and intra-application communication available for a target environment. This approach provides a high-fidelity means to study how components interact, which is a necessary step for a comprehensive security analysis. For example, our analysis can also be used to perform information flow analysis between application components and to identify new types of attacks, such as application collusion [128, 129], where two applications work together to compromise the privacy of the user. In general, most vulnerability analysis techniques for Android need to analyze ICC, and thus can benefit from our analysis.

Android application components interact through ICC objects – mainly *Intents*. Components can also communicate across applications, allowing developers to reuse functionality. The proposed approach identifies a *specification* for every ICC source and sink. This includes the location of the ICC entry point or exit point, the ICC Intent action, data type and category, as well as the ICC Intent key/value types and the target component name. Note that where ICC values are not fixed we infer all possible ICC

values, thereby building a complete specification of the possible ways ICC can be used. The specifications are recorded in a database in flows detected by matching compatible specifications. The structure of the specifications ensures that ICC matching is efficient.

We make the following contributions in this chapter:

- We show how to reduce the analysis of Intent ICC to an Interprocedural Distributive Environment (IDE) problem. Such a problem can be solved efficiently using existing algorithms [12].
- We develop Epicc, a working analysis tool built on top of an existing IDE framework [130] within the Soot [131] suite, which we have made available at `http://siis.cse.psu.edu/epicc/`.
- We perform a study of ICC vulnerabilities and compare it to ComDroid [4], the current state-of-the-art. Our ICC vulnerability detection shows significantly increased precision, with ComDroid flagging 32% more code locations. While we use our tool to perform a study of some ICC vulnerabilities, our analysis can be used to address a wider variety of ICC-related vulnerabilities.
- We perform a study of ICC in 1,200 representative applications from the free section of the Google Play Store. We found that the majority of specifications were relatively narrow, most ICC objects having a single possible type. Also, key/value pairs are widely used to communicate data over ICC. Lastly, our analysis scales well, with an average analysis time of 113 seconds per application.

This chapter is organized as follows. In the next section, we formulate the ICC problem and motivate it with examples of analyses. Then, in Section 5.2 we present our methodology. Next, we present our formal model for ICC in Section 5.3. Finally, we evaluate our approach in Section 5.4.

5.1 Problem Formulation

As highlighted above, the goal of the analysis presented in this chapter is to infer specifications for each ICC source and sink in the targeted applications. These specifications detail the type, form, and data associated with the communication. We consider communication with Content Providers to be out of scope. Our analysis has the following goals:

```
1 private OnClickListener mMyListener = new OnClickListener()
    {
2   public void onClick(View v) {
3     Intent intent = new Intent();
4     intent.setAction("a.b.ACTION");
5     intent.addCategory("a.b.CATEGORY");
6     startActivity(intent);
7   }
8 };
```

Figure 5.1. Example of implicit Intent communication.

Soundness – The analysis should generate *all* specifications for ICC that may appear at runtime. Informally, we want to guarantee that no ICC will go undetected. Our analysis was designed to be sound under the assumption that the applications use no reflection or native calls, and that the components' life cycle is modeled completely.

Precision – The previous goal implies that some generated ICC specifications may not happen at runtime ("false positives"). Precision means that we want to limit the number of cases where two components are detected as connected, even though they are not in practice. Our analysis currently does not handle URIs[1]. Since the data contained in Intents in the form of URIs is used to match Intents to target components, not using URIs as a matching criterion potentially implies more false positives. Other possible sources of imprecision include the points-to and string analyses. We empirically demonstrate analysis precision in Section 5.4.1.

5.1.1 Applications

Although Android applications are developed in Java, existing Java analyses cannot handle the Android-specific ICC mechanisms. The analysis presented in this chapter deals with ICC and can be used as the basis for numerous important analyses, for example:

Finding ICC vulnerabilities – Android ICC APIs are complex to use, which causes developers to commonly leave their applications vulnerable [4, 107]. Examples of ICC vulnerabilities include sending an Intent that may be intercepted by a malicious component, or exposing components to be launched by a malicious Intent. The first application of our work is in finding these vulnerabilities. We present a study of ICC vulnerabilities

[1]Extending the analysis to include URIs is a straightforward exercise using the same approaches defined in the following sections. We have a working prototype and defer reporting on it to future work.

in Section 5.4.4.

Finding attacks on ICC vulnerabilities – Our analysis can go beyond ICC vulnerability detection and can be used for a holistic attack detection process. For each application we compute entry points and exit points and systematically match them with entry and exit points of previously processed applications. Therefore, our analysis can detect applications that may exploit a given vulnerability.

Inter-component information flow analysis – We compute which data sent at an exit point can potentially be used at a receiving entry point. An information flow analysis using our ICC analysis could find flows between a source in a component and a sink in a different component (possibly in a different application).

In the case where the source and sink components belong to different applications, we can detect cases of *application collusion* [128, 129]. The unique communication primitives in Android allow for a new attack model for malicious or privacy-violating application developers. Two or more applications can work together to leak private information and go undetected. For example, application A can request access to GPS location information, while application B requests access to the network. Permissions requested by each application do not seem suspicious, therefore a user might download both applications. However, in practice it is possible for A and B to work together to leak GPS location data to the network. It is almost impossible for users to anticipate this kind of breach of privacy. However, statically detecting this attack is a simple application of our ICC analysis, whereas the current state-of-the-art requires dynamic analysis and modification of the Android platform [129].

5.1.2 Examples

Figure 5.1 shows a representative example of ICC programming. It defines a field that is a click listener. When activated by a click on an element, it creates Intent *intent* and sets its action and category. Finally, the *startActivity*() call takes *intent* as an argument. It causes the OS to find an activity that accepts Intents with the given action and category. When such an activity is found, it is started by the OS. If several activities meeting the action and category requirements are found, the user is asked which activity should be started.

This first example is trivial. Let us now consider the more complex example from Figure 5.2, which was first presented in Figure 2.4 and will be used throughout this chapter. Let us assume that this piece of code is in a banking application. First, Intent

```
1  public void onClick(View v) {
2    Intent i = new Intent();
3    i.putExtra("Balance", this.mBalance);
4    if (this.mCondition) {
5      i.setClassName("a.b", "a.b.MyClass");
6    } else {
7      i.setAction("a.b.ACTION");
8      i.addCategory("a.b.CATEGORY");
9      i = modifyIntent(i);
10   }
11   startActivity(i);
12 }
13
14 public Intent modifyIntent(Intent in) {
15   Intent intent = new Intent(in);
16   intent.setAction("a.b.NEW_ACTION");
17   intent.addCategory("a.b.NEW_CATEGORY");
18   return intent;
19 }
```

Figure 5.2. Intent communication: running example.

intent containing private data is created. Then, if condition *this.mCondition* is true, *intent* is made explicit by targeting a specific class. Otherwise, it is made implicit. Next, an activity is started using *startActivity*(). Note that we have made the implicit Intent branch contrived to demonstrate how function calls are handled. In this example, the safe branch is the one in which *intent* targets a specific component. The other one may leak data, since it might be intercepted by a malicious Activity. We want to be able to detect that possible information leak. In other words, we want to infer the two possible Intent values at *startActivity*(). In particular, knowing the implicit value would allow us to find which applications can intercept it and to detect possible eavesdropping.

5.2 Connecting Application Components: Overview

Our analysis aims at connecting components, both within single applications and between different applications. For each input application $\mathcal{A}$, it outputs the following:

1. A list of entry points for $\mathcal{A}$ that may be called by components in $\mathcal{A}$ or in other applications.

2. A list of exit points for $\mathcal{A}$ where $\mathcal{A}$ may send an Intent to another component. That component can be in $\mathcal{A}$ or in a different application. The value of Intents at

each exit point is precisely determined, which allows us to accurately determine possible targets.

3. A list of links between $\mathcal{A}$'s own components and between $\mathcal{A}$'s components and other applications' components. These links are computed using 1. and 2. as well as all the previously analyzed applications.

Let us consider the example in Figure 5.2, which is part of our example banking application. The `startActivity(i)` instruction is an exit point for the application. Our analysis outputs the value of i at this instruction as well as all the possible targets. These targets can be components of our banking application itself or components of previously analyzed applications.

Figure 5.3 shows an overview of our component matching process. It can be divided into three main functions:

- Finding target components that can be started by other components (i.e., "entry points") and identifying criteria for a target to be activated.
- Finding characteristics of exit points, i.e., what kind of targets can be activated at these program points.
- Matching exit points with possible targets.

Given an application, we start by parsing its manifest file to extract package information, permissions used and a list of components[2] and associated intent filters (1). These components are the potential targets of ICC. We match these possible entry points with the pool of already computed exit points (2). We then add the newly computed entry points to our database of entry points (3). This database and the exit points database grow as we analyze more applications. Then we proceed with the string analysis, which identifies key API method arguments such as action strings or component names (4). Next, the main Interprocedural Distributive Environment (IDE) analysis precisely computes the values of Intent used at ICC API calls (5). It also computes the values of Intent Filters that select Intents received by dynamically registered Broadcast Receivers. These exit points are matched with entry points from the existing pool of entry points (6). The newly computed exit points are stored in the exit point database to allow for later matching (7). The values associated with dynamically registered Broadcast Receivers are used

[2]Broadcast Receivers can be registered either statically in the manifest file or dynamically using the *registerReceiver*() methods.

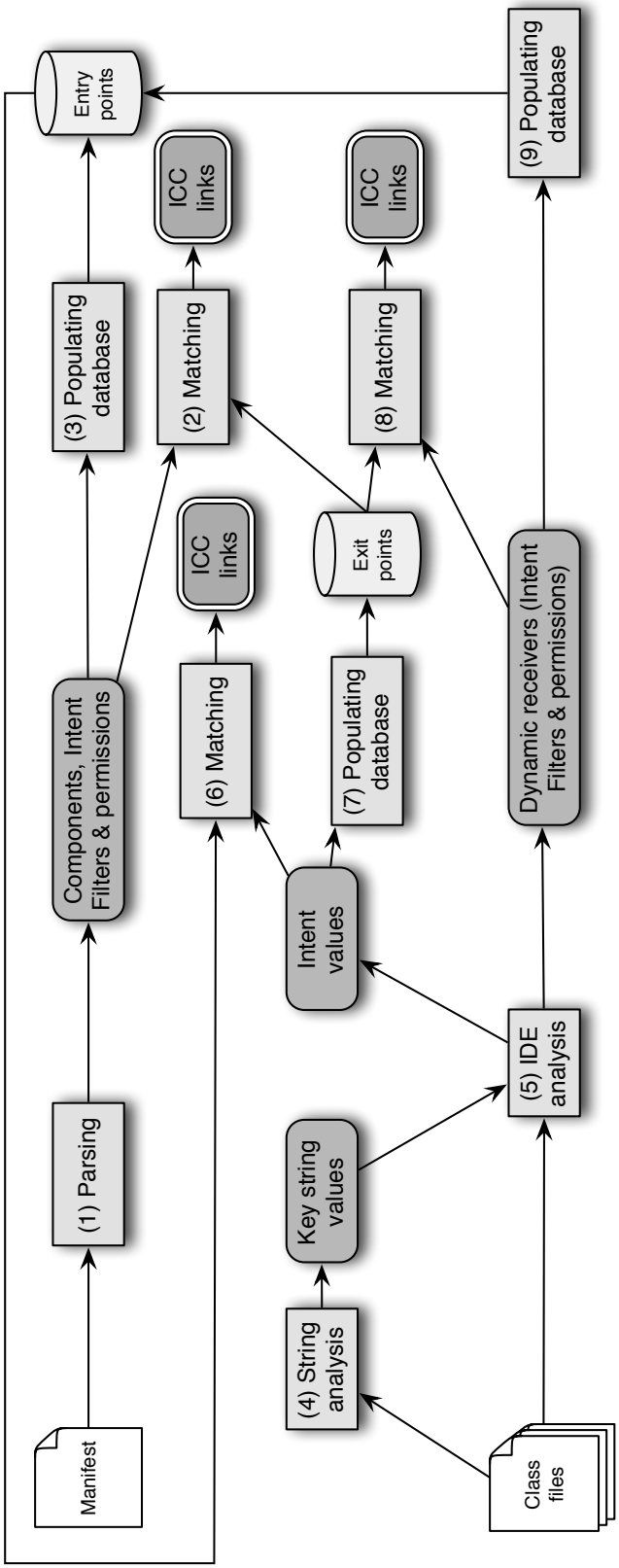


Figure 5.3. Connecting application components.

for matching with exit points in the database (8). Finally, these values are stored in the entry point database (9).

One of the inputs to our analysis is a set of class files. These classes are in Java bytecode format, since our analysis is built on top of Soot [131], an existing Java analysis framework. Android application code is distributed in a platform-specific Dalvik bytecode format that is optimized for resource-constrained devices, such as smartphones and tablets. Therefore, we use Dare [10], an existing tool that efficiently and accurately retarget Dalvik bytecode to Java bytecode. While other tools such as dex2jar[3] and ded [121] are available, Dare is currently the only formally defined one and other tools' outputs are sometimes not reliable.

The manifest parsing step is trivial and we use a simple string analysis (see Section 4.8). Also, the matching process matches exit points with entry points. It can be made efficient if properly organized in a database. Thus, we focus our description on the main IDE analysis.

It is important to distinguish between what is computed by the string analysis and by the IDE analysis. In the example from Figure 5.1, the string analysis computes the values of the arguments to the API calls *setAction*() and *addCategory*(). The IDE analysis, on the other hand, uses the results from the string analysis along with a model of the Android ICC API to determine the value of the Intent. In particular, in Figure 5.1, it determines that, at the call to *startActivity*(), Intent *intent* has action `a.b.ACTION` and category `a.b.CATEGORY`. In Figure 5.2, the IDE analysis tells us that i has two possibles values at the call to *startActivity*() and determines exactly what the two possible values are.

Reducing the Intent ICC problem to an IDE problem [12] has important advantages. Our analysis is scalable (see Section 4.8). Further, it is a precise analysis, in the sense that it generates few false positives (links between two components which may not communicate in reality). Thus, security analyses using our ICC analysis will not be plagued by ICC-related false positives. This precision is due to the fact that the IDE framework is flow-sensitive, inter-procedural and context-sensitive.

The flow-sensitivity means that we can distinguish Intent values between different program points. In the example from Figure 5.2, if Intent i was used for ICC right before the call to $modifyIntent()$, we would accurately capture that this value is different from the one at *startActivity*(). The context-sensitivity means that the analysis of the call to $modifyIntent()$ is sensitive to the method's calling context. If $modifyIntent()$ is

[3]Available at `http://code.google.com/p/dex2jar/`.

called at another location with a different argument $i2$, the analysis will precisely distinguish between the values returned by the two calls. Otherwise, in a context-insensitive analysis, the return value would summarize all possible values given all contexts in which $modifyIntent()$ is called in the program. The value of i computed by a context-insensitive analysis would be influenced by the value of $i2$, which is not the case in reality. That would be significantly less precise, resulting in more false positives.

5.3 Reducing Intent ICC to an IDE problem

To solve the Intent ICC problem, we need to model four different kinds of objects. First, ComponentName objects contain a package name and a class name. They can be used by explicit Intents. For example, in method $makeComponentName()$ of Figure 5.4, a ComponentName object can take two different values depending on which branch is executed. In the first branch, it refers to class `a.b.MyClass` from application package `c.d`. In the second one, it refers to class `a.b.MySecondClass`. We want to know the possible return values of $makeComponentName()$.

Second, Bundle objects store data as key-value mappings. Method $makeBundle()$ of Figure 5.4 creates a Bundle and modifies its value. We need to find the possible return values of $makeBundle()$.

Third, Intent objects are the main ICC communication objects. They contain all the data that is used to start other components. In method $onClick()$ of Figure 5.4, the target class of *intent* is set using the return value of $makeComponentName()$. Its extra data is set to the return value of $makeBundle()$. Finally, a new Activity is started using the newly created Intent. We need to determine the value of *intent* at the `startActivity(intent)` instruction.

Fourth, IntentFilter objects are used for dynamic Broadcast Receivers. For example, in the $registerMyReceiver()$ method on Figure 5.4, an action and a category are added to IntentFilter f. Then a Broadcast Receiver of type `MyReceiver` (which we assume to be defined) is registered using method $registerReceiver()$. It receives Intents that have action `a.b.ACTION` and category `a.b.CATEGORY` and that originate from applications with permission `a.b.PERMISSION`. We want to determine the arguments to the $registerReceiver()$ call. That is, we want to know that f contains action `a.b.ACTION` and category `a.b.CATEGORY`. We also want to know that the type of the Broadcast Receiver is `MyReceiver`.

In this section, we use the notations from Sagiv *et al.* [12] summarized in Section 2.2.

```
1  public ComponentName makeComponentName() {
2    ComponentName c;
3    if (this.mCondition) {
4      c = new ComponentName("c.d", "a.b.MyClass");
5    } else {
6      c = new ComponentName("c.d", "a.b.MySecondClass");
7    }
8    return c;
9  }
10
11 public Bundle makeBundle(Bundle b) {
12   Bundle bundle = new Bundle();
13   bundle.putString("FirstName", this.mFirstName);
14   bundle.putAll(b);
15   bundle.remove("Surname");
16   return bundle;
17 }
18
19 public void onClick(View v) {
20   Intent intent = new Intent();
21   intent.setCompontent(makeComponentName());
22   Bundle b = new Bundle();
23   b.putString("Surname", this.mSurname);
24   intent.putExtras(makeBundle(b));
25   registerMyReceiver();
26   startActivity(intent);
27 }
28
29 public void registerMyReceiver() {
30   IntentFilter f = new IntentFilter();
31   f.addAction("a.b.ACTION");
32   f.addCategory("a.b.CATEGORY");
33   registerReceiver(new MyReceiver(), f, "a.b.PERMISSION",
       null);
34 }
```

Figure 5.4. ICC objects example.

We assume that string method arguments are available. We describe the string analysis used in our implementation in Section 4.8.

5.3.1 ComponentName Model

In this section, we introduce the model we use for ComponentName objects. We introduce the notion of a branch ComponentName value. It represents the value that a ComponentName object can take on a single branch, given a single possible string argument value for each method setting the ComponentName's package and class names, and in the absence of aliasing.

Definition 1. *A* branch ComponentName value *is a tuple* $c = (p, k)$*, where* p *is a package name and* k *is a class name.*

In method *makeComponentName*() of Figure 5.4, two branch ComponentName values are constructed:

$$(\texttt{c.d}, \texttt{a.b.MyClass}) \tag{5.1}$$

and

$$(\texttt{c.d}, \texttt{a.b.MySecondClass}). \tag{5.2}$$

The next definition introduces ComponentName values, which represent the possibly multiple values that a ComponentName can have at a program point. A ComponentName can take several values in different cases:

- After traversing different branches, as in method *makeComponentName*() of Figure 5.4.
- When a string argument can have several values at a method call.
- When an object reference is a possible alias of another local reference or an object field.
- When an object reference is a possible array element.

In the last two cases, in order to account for the possibility of a false positive in the alias analysis, we keep track of two branch ComponentName values. One considers the influence of the call on the possible alias and the other one does not.

Definition 2. *A* ComponentName value C *is a set of branch ComponentName values:* $C = \{c_1, c_2, \cdots, c_m\}$. *The set of ComponentName values is denoted as* V_c. *We define* $\bot = \varnothing$ *and* $\top$ *as the ComponentName value that is the set of all possible branch ComponentName values in the program. The operators* $\cup$ *and* $\subseteq$ *are defined as traditional set union and comparison operators: for* $C_1, C_2 \in V_c$, $C_1 \subseteq C_2$ *iff* $C_1 \cup C_2 = C_2$. $L_c = (V_c, \cup)$ *is a join semilattice.*

Note that given the definitions of $\bot$ and $\top$ as specific sets, $\cup$ and $\subseteq$ naturally apply to them. For example, for all $C \in V_c$, $\top \cup C = \top$.

In method *makeComponentName*() from Figure 5.4, the value of c at the return statement is

$$\{(\texttt{c.d}, \texttt{a.b.MyClass}), (\texttt{c.d}, \texttt{a.b.MySecondClass})\}. \tag{5.3}$$

It simply combines the values of c created in the two branches, given by Equations (5.1) and (5.2).

We define transformers from V_c to V_c that represent the influence of a statement or a sequence of statements on a ComponentName value. A pointwise branch ComponentName transformer represents the influence of a single branch, whereas a pointwise ComponentName transformer represents the influence of possibly multiple branches.

Definition 3. *A* pointwise branch ComponentName transformer *is a function*

$$\delta^c_{(\pi,\chi)} : V_c \to V_c,$$

where π *is a package name and* χ *is a class name. It is such that, for each* $C \in V_c$,

$$\delta^c_{(\pi,\chi)}(C) = \{(\pi, \chi)\}.$$

Note that $\delta^c_{(\pi,\chi)}(C)$ is independent of C, because API methods for ComponentName objects systematically replace existing values for package and class names. In the example from Figure 5.4, the pointwise branch ComponentName transformer corresponding to the first branch is

$$\delta^c_{(\texttt{c.d},\texttt{a.b.MyClass})}, \tag{5.4}$$

and the one for the second branch is

$$\delta^{c}_{(\texttt{c.d},\texttt{a.b.MySecondClass})}. \tag{5.5}$$

Definition 4. *A* pointwise ComponentName transformer *is a function*

$$\delta^{c}_{\{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}} : V_c \to V_c$$

such that, for each $C \in V_c$*,*

$$\delta^{c}_{\{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}}(C) = \{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}.$$

A pointwise ComponentName transformer summarizes the effect of multiple branches (or a single branch with multiple possible string arguments, or with possible aliasing) on a ComponentName value. That is, given the value C of a ComponentName right after statement s_i and given transformer $\delta^{c}_{\{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}}$ that summarizes the influence of statements $s_{i+1},\cdots,s_k$ on C, $\delta^{c}_{\{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}}(C)$ represents all the possible values of C right after s_k. In method *makeComponentName*() of Figure 5.4, the pointwise ComponentName transformer that models the two branches is

$$\delta^{c}_{\{(\texttt{c.d},\texttt{a.b.MyClass}),(\texttt{c.d},\texttt{a.b.MySecondClass})\}}. \tag{5.6}$$

It combines the transformers given by Equations (5.4) and (5.5). In order to understand how this transformer is applied in practice, we should mention that the algorithm to solve IDE problems initially sets values to $\perp$ [12]. Therefore, in method *makeComponentName*(), the value associated with c is initially $\perp = \varnothing$. Using Definition 4, we can easily see that if we apply the transformer given by Equation (5.6), we get the value given by Equation (5.3). This confirms that the transformer models the influence of the two branches:

$$\delta^{c}_{\{(\texttt{c.d},\texttt{a.b.MyClass}),(\texttt{c.d},\texttt{a.b.MySecondClass})\}}(\perp) = \{\,(\texttt{c.d},\texttt{a.b.MyClass})\,, \\ (\texttt{c.d},\texttt{a.b.MySecondClass})\}.$$

5.3.2 Bundle Model

The model of Bundle objects is defined similarly to the model of ComponentName objects. An additional difficulty is introduced. The data in a Bundle can be modified by

adding the data in another Bundle to it, as shown in method *makeBundle*() of Figure 5.4. In this example, the data in Bundle b is added to the data in Bundle *bundle*. Bundle *bundle* is later modified by removing the key-value pair with key `Surname`. The issue is that when the data flow problem is being tackled, the value of b is not known. Therefore, the influence of the call to `remove("Surname")` is not known: if a key-value pair with key `Surname` is part of b, then the call removes it from *bundle*. Otherwise, it has no influence.

Our approach to deal with this object composition problem is to perform two successive analyses. In Analysis I, we use placeholders for Bundles such as b in instruction `bundle.putAll(b)`. We also record all subsequent method calls affecting *bundle*. After the problem is solved, b's key-value pairs at the `putAll(b)` method call are known, as well as the subsequent method calls. We then perform Analysis II, in which b's key-value pairs are added to *bundle*'s. The influence of the subsequent method call is precisely evaluated and finally the value of *bundle* at the return statement can be known.

5.3.2.1 Analysis I

In the first analysis, we consider intermediate values that contain "placeholders" for Bundle values that are not known when the problem is being solved.

Definition 5. *An* intermediate branch Bundle value *is a tuple* $b_i = (E, O)$*, where:*

- E *is a set of keys describing extra data.*
- O *is a tuple of two types of elements.* O *contains references to particular Bundle symbols at instructions where putAll*() *calls occur.* O *also contains functions from* V_b^i *to* V_b^i*, where* V_b^i *is the set of intermediate Bundle values defined below. These functions represent a sequence of method calls affecting a Bundle.*

The difference with previous definitions is the introduction of O, which models calls to *putAll*() as well as subsequent calls affecting the same Bundle. In method *makeBundle*() of Figure 5.4, at the return statement, the intermediate branch Bundle value associated with *bundle* is (E, O), where

$$E = \{\texttt{FirstName}\}, \tag{5.7}$$

$$O = \left((b, \texttt{bundle.putAll(b)}), \beta^b_{(\varnothing, \{\texttt{Surname}\}, 0, ())}\right). \tag{5.8}$$

In O, (b, `bundle.putAll(b)`) is a reference to variable b at instruction `bundle.putAll(b)`. $\beta^b_{(\varnothing, \texttt{Surname}, 0, ())}$ models the *remove*() method call. It is defined below.

We just defined intermediate branch Bundle values. As we did before, we need to consider multiple branches and related issues (e.g., several possible string values):

Definition 6. *An* intermediate Bundle value B_i *is a set of intermediate branch Bundle values:* $B_i = \{b_{i_1}, \cdots, b_{i_m}\}$. *The set of intermediate Bundle values is* V_b^i. *We define* $\perp = \varnothing$ *and* $\top$ *as the intermediate Bundle value that is the set of all possible intermediate branch Bundle values in the program. We define* $\subseteq$ *and* $\cup$ *as natural set comparison and union operators. They are such that, for* $B_{i_1}, B_{i_2} \in V_b^i$, $B_{i_1} \subseteq B_{i_2}$ *iff* $B_{i_1} \cup B_{i_2} = B_{i_2}$. $L_b^i = (V_b^i, \cup)$ *is a join semilattice.*

In method *makeBundle*() from Figure 5.4, since there is only a single branch, the intermediate Bundle value associated with *bundle* at the return statement is $\{(E, O)\}$, where E and O are given by Equations (5.7) and (5.8).

Pointwise transformers are defined from V_b^i to V_b^i. Similarly to the ComponentName model, we first introduce pointwise branch Bundle transformers before defining pointwise Bundle transformers. In the definitions below, we use the $\setminus$ notation for set difference, and $\cup$ is naturally extended to tuples such that

$$(a_1, \cdots, a_k) \cup (a_{k+1}, \cdots, a_l) = (a_1, \cdots, a_k, a_{k+1}, \cdots, a_l).$$

Definition 7. *A* pointwise branch Bundle transformer *is a function*

$$\beta^b_{(\eta^+, \eta^-, cl, \Theta)} : V_b^i \to V_b^i,$$

where:

- η^+ *is a set of string keys describing extra data. It models calls to* $putExtra()$ *methods.*
- η^- *is a set of string keys describing removed extra data. It represents the influence of calls to the* $removeExtra()$ *method.*
- cl *takes value 1 if the Bundle data has been cleared with the* $clear()$ *method and 0 otherwise.*
- Θ *is a tuple of two types of elements. It contains references to particular Bundle symbols at instructions where* $putAll()$ *calls occur. It also contains functions from* V_b^i *to* V_b^i. *These functions represent a sequence of method calls affecting a Bundle.*

It is such that

$$\beta^b_{(\eta^+,\eta^-,cl,\Theta)}(\bot) = \{(\eta^+ \backslash \eta^-, \Theta)\}$$

and, for $B_i = \{(E_1, O_1), \cdots, (E_m, O_m)\}$ *(*$B_i \neq \bot$*),*

$$\beta^b_{(\eta^+,\eta^-,cl,\Theta)}(B_i) = \{(E'_1, O'_1), \cdots, (E'_m, O'_m)\}$$

where, for each j *from* 1 *to* m*:*

$$E'_j = \begin{cases} \eta^+ \backslash \eta^- & \text{if } cl = 1 \\ (E_j \cup \eta^+) \backslash \eta^- & \text{if } cl = 0 \text{ and } O_j = \varnothing \\ E_j & \text{otherwise} \end{cases}$$

$$O'_j = \begin{cases} \Theta & \text{if } cl = 1 \text{ or } O_j = \varnothing \\ O_j \cup \left(\beta^b_{(\eta^+,\eta^-,0,\Theta)}\right) & \text{otherwise} \end{cases}$$

The definition of E'_j accounts for several possible cases:

- If the Bundle data has been cleared (i.e., $cl = 1$), then we discard any data contained in E_j. This leads to value $\eta^+ \backslash \eta^-$ for E'_j: we only keep the values η^+ that were added to the Bundle data and remove the values η^- that were removed from it.

- If the Bundle has not been cleared, then there are two possible cases: either no reference to another Bundle has been previously recorded (i.e., $O_j = \varnothing$), or such a reference has been recorded to model a call to *putAll*(). In the first case, we simply take the union of the original set E_j and the set η^+ of added values, and subtract the set η^- of removed values. This explains the $(E_j \cup \eta^+) \backslash \eta^-$ value. In the second case, a call to *putAll*() has been detected, which means that any further method call adding or removing data has to be added to set O_j instead of E_j. Therefore in this case $E'_j = E_j$.

The definition of O'_j considers several cases:

- If the Bundle data has been cleared, then the previous value of O_j is irrelevant and we set $O'_j = \Theta$. Also, if O_j is empty, then we can also just set O'_j to Θ (which may or may not be empty).

- Otherwise, the Bundle data has not been cleared ($cl = 0$) and a call to *putAll*() has been detected ($O_j \neq \varnothing$). Then it means that the current function models method calls that happened after a call to *putAll*(). Therefore we need to record $\beta^b_{(\eta^+,\eta^-,0,\Theta)}$ in O'_j, which explains the definition $O'_j = O_j \cup (\beta^b_{(\eta^+,\eta^-,0,\Theta)})$.

For example, the pointwise branch Bundle transformer that models the influence of the method *makeBundle*() from Figure 5.4 is $\beta^b_{(\eta^+,\varnothing,0,\Theta)}$, where

$$\eta^+ = \{\texttt{FirstName}\}, \tag{5.9}$$

$$\Theta = \left(\beta^b_{(\varnothing,\{\texttt{Surname}\},0,())}(b, \texttt{bundle.putAll(b)}), \beta^b_{(\varnothing,\{\texttt{Surname}\},0,())}\right). \tag{5.10}$$

Pointwise branch Bundle transformers model the influence of a single branch. In order to account for multiple branches or issues such as possible aliasing false positive, we define pointwise Bundle transformers.

Definition 8. *A* pointwise Bundle transformer *is a function*

$$\beta^b_{\{(\eta_1^+ +,\eta_1^-,cl_1,\Theta_1),\cdots,(\eta_n^+,\eta_n^-,cl_n,\Theta_n)\}} : V_b^i \to V_b^i$$

such that, for each $B_i \in V_b^i$,

$$\beta^b_{\{(\eta_1^+,\eta_1^-,cl_1,\Theta_1),\cdots,(\eta_n^+,\eta_n^-,cl_n,\Theta_n)\}}(B_i) = \beta^b_{(\eta_1^+,\eta_1^-,cl_1,\Theta_1)}(B_i) \cup \cdots \cup \beta^b_{(\eta_n^+,\eta_n^-,cl_n,\Theta_n)}(B_i).$$

For example, method *makeBundle*() from Figure 5.4 only has a single branch, thus the pointwise Bundle transformer that models it is simply $\beta^b_{\{(\eta^+,\varnothing,0,\Theta)\}}$, where η^+ and Θ are given in Equations (5.9) and (5.10). As we did for the ComponentName value example, we can confirm using Definitions 7 and 8 that $\beta^b_{\{(\eta^+,\varnothing,0,\Theta)\}}(\perp) = \{(E, O)\}$, where E and O are given by Equations (5.7) and (5.8).

5.3.2.2 Analysis II

After Analysis I has been performed, the values of the Bundles used in placeholders in intermediate Bundle values are known. Ultimately, we want to obtain branch Bundle values and finally Bundle values:

Definition 9. *A* branch Bundle value b *is a set* E *of string keys describing extra data.*

Definition 10. *A* Bundle value B *is a set of branch Bundle values:* $B = \{b_1, \cdots, b_m\}$.

Since the values of the referenced Bundles are known, we can integrate them into the Bundle values referring to them. Then the influence of the subsequent method calls that have been recorded can precisely be known.

Let us consider the example of *makeBundle*() from Figure 5.4. After Analysis I has been performed, we know that the intermediate value of *bundle* at the return statement is $\{(E, O)\}$, where

$$E = \{\texttt{FirstName}\},$$
$$O = \left((b, \texttt{bundle.putAll(b)}), \beta^{b}_{(\varnothing, \{\texttt{Surname}\}, 0, ())}\right).$$

We consider all elements of O in order. The first element of O is (b, `bundle.putAll(b)`), therefore we integrate b's value into *bundle*. From Analysis I, we know that the value of b at instruction `bundle.putAll(b)` is $\{\{\texttt{Surname}\}, \varnothing\}$. Thus, E becomes {`FirstName`, `Surname`}. The next element of O is $\beta^{b}_{(\varnothing, \{\texttt{Surname}\}, 0, ())}$. This means that we have to remove key `Surname` from E. The final value of E is therefore {`FirstName`}. Thus, the Bundle value associated with *bundle* at the return statement is $\{\{\texttt{FirstName}\}\}$.

Note that the referenced Bundle can also make references to other Bundles. In that case, we perform the resolution for the referenced Bundles first. There can be an arbitrary number of levels of indirection. Analysis II is iterated until a fix-point is reached.

5.3.3 Intent and IntentFilter Models

The Intent model is defined similarly to the Bundle model, which includes object composition. In method *onClick*() of Figure 5.4, the target of Intent *intent* is set using a ComponentName object and its extra data is set with a Bundle. Because of this object composition, finding the Intent value also involves two analyses similar to the ones performed for Bundles. First, intermediate Intent values with placeholders for referenced ComponentName and Bundle objects are found. Second, the referenced objects' values are integrated into *intent*'s value.

Similarly to the Bundle model, we define intermediate branch Intent values and intermediate Intent values. The set of intermediate Intent values is V^i_i and we define a lattice $L^i_i = (V^i_i, \cup)$ as we did for L^i_b. We also define pointwise branch Intent transformers and pointwise Intent transformers. For example, in method *onClick*() of Figure 5.4, the final intermediate value for *intent* simply has placeholders for a ComponentName and a Bundle value. Other fields, such as action and categories, are empty. The ComponentName and Bundle values are computed using the models presented in Sections 5.3.1 and 5.3.2.

Finally, we define branch Intent values and Intent values, which are output by the second analysis. The final value for *intent* after the second analysis precisely contains the two possible targets (`a.b.MyClass` and `a.b.MySecondClass` in package `c.d`) and extra data key `FirstName`. For conciseness, and given the strong similarities with the Bundle model, we do not include a full description of the Intent model here.

In order to analyze dynamic Broadcast Receivers, we model IntentFilter objects. Modeling IntentFilters is similar to modeling Intents, except that IntentFilters do not involve object composition. That is because IntentFilters do not have methods taking other IntentFilters as argument, except for a copy constructor. Thus, their analysis is simpler and involves a single step. Similarly to what we did for other ICC models, we define branch IntentFilter values, IntentFilter values, pointwise branch IntentFilter transformers and pointwise IntentFilter transformers. In particular, we define lattice $L_f = (V_f, \cup)$, where V_f is the set of IntentFilter values. In method *onClick*() from Figure 5.4, the final value of f contains action `a.b.ACTION` and category `a.b.CATEGORY`. Given the similarity of the IntentFilter model with previous models, we do not include a complete description.

5.3.4 Casting as an IDE Problem

These definitions allow us to define environment transformers for our problem. Given environment $e \in Env(D, L)$, environment transformer $\lambda e.e$ is the *identity*, which does not change the value of e. Given Intent i and Intent value I, $\lambda e.e[i \mapsto I]$ transforms e to an environment where all values are the same as in e, except that Intent i is associated with value I.

We define an environment transformer for each API method call. Each of these environment transformers uses the pointwise environment transformers defined in Sections 5.3.1, 5.3.2 and 5.3.3. It precisely describes the influence of a method call on the value associated with each of the symbols in D.

Figure 2.6 shows some environment transformers and their pointwise representation. The first one is a constructor invocation, which sets the value corresponding to b to $\perp$. The second one adds an integer to the key-value pairs in Bundle b's extra data, which is represented by environment transformer

$$\lambda e.e\left[b \mapsto \beta^{b}_{(\{\texttt{MyInt}\},\varnothing,0,())}\left(e(b)\right)\right].$$

It means that the environment stays the same, except that the value associated with b

becomes

$$\beta^{b}_{(\{\texttt{MyInt}\},\varnothing,0,())}\left(e(b)\right),$$

with $e(b)$ being the value previously associated with b in environment e. The pointwise transformer for b is

$$\beta^{b}_{(\{\texttt{MyInt}\},\varnothing,0,())},$$

which we denote by

$$\lambda B.\beta^{b}_{(\{\texttt{MyInt}\},\varnothing,0,())}(B)$$

on Figure 2.6 for consistency with the other pointwise transformers. It simply adds key `MyInt` to the set of data keys. The next transformer is for a copy constructor, where the value associated with d is assigned to the value associated with b. The last transformer clears the data keys associated with d.

Trivially, these environment transformers are distributive. Therefore, the following proposition holds.

Proposition 1. *Let G^* be the supergraph of an Android application. Let D_c, D_b D_i and D_f be the sets of ComponentName, Bundle and Intent variables, respectively, to which we add the special symbol Λ[4]. Let L_c, L^i_b, L^i_i and L_f be the lattices defined above. Let M_c, M_b, M_i and M_f be the corresponding assignments of distributive environment transformers. Then (G^*, D_c, L_c, M_c), $\left(G^*, D_b, L^i_b, M^i_b\right)$, $\left(G^*, D_b, L^i_i, M^i_i\right)$ and (G^*, D_i, L_f, M_f) are IDE problems.*

It follows from this proposition that we can use the algorithm from [12] to solve the Intent ICC problem.

The original IDE framework [12] requires that the micro-function be represented efficiently in order to achieve the time complexity of $O(ED^3)$. Our model does not meet these requirements: in particular, applying, composing, joining micro-function or testing for equality of micro-functions cannot be done in constant time. Indeed, the size of micro-functions grows with the number of branches, aliases and possible string arguments (see Equation 5.6 for an example with two branches). However, in practice we can find solutions to our IDE problem instances in reasonable time, as we show in Section 4.8.

[4]Recall from Section 2.2.2 that Λ symbolizes the absence of a data flow fact.

5.4 Evaluation

This section describes an evaluation of the approach presented in the preceding sections, and briefly characterizes the use of ICC in Android applications. We also present a study of potential ICC vulnerabilities. Our implementation is called Epicc (Efficient and Precise ICC) and is available at `http://siis.cse.psu.edu/epicc/`. It is built on Heros [130], an IDE framework within Soot [131]. We also provide the version of Soot that we modified to handle pathological cases encountered with retargeted code.

In order to compute string arguments, we use a simple analysis traversing the interprocedural control flow graph of the application. The traversal starts at the call site and looks for constant assignments to the call arguments. If a string argument cannot be determined, we conservatively assume that the argument can be any string. As we show in Section 5.4.1, in many cases we are able to find precise string arguments. More complex analyses can be used if more precision is desired [132].

For points-to analysis and call graph construction, we use Spark [133], which is part of Soot. It performs a flow-sensitive, context-insensitive analysis. We approximate the call graph in components with multiple entry points. In order to generate a call graph of an Android application, we currently use a "wrapper" as an entry point. This wrapper calls each class entry point once, which may under-approximate what happens at runtime. This impacts a specification only if an ICC field (e.g., Intent) is modified in a way that depends on the runtime execution order of class entry points. Generally, if we assume that our model of components' life cycle is complete and if the application does not use native calls or reflection, then our results are sound.

The analysis presented in this section is performed on two datasets. The first *random sample* dataset contains 350 applications, 348 of which were successfully analyzed after retargeting. They were extracted from the Google Play store[5] between September 2012 and January 2013. The applications were selected at random from over 200,000 applications in our corpus. The second *popular application* dataset contains the top 25 most popular free applications from each of the 34 application categories in the Play store. The 850 selected applications were downloaded from that application store on January 30, 2013. Of those 850 applications, 838 could be retargeted and processed and were used in the experiments below. The 14 applications which were not analyzed were pathological cases where retargeting yielded code which could not be analyzed (e.g., in some cases the Dare tool generated offsets with integer overflow errors due to excessive

[5]Available at `https://play.google.com/store/apps`.

method sizes), or where applications could not be processed by Soot (e.g., character encoding problems).

5.4.1 Complete Recovery of ICC Specifications

The first set of tests evaluates the technique's *precision* with our datasets. We define the precision metric to be the percentage of source and sink locations for which a specification is identified without ambiguity. Ambiguity occurs when an ICC API method argument cannot be determined. These arguments are mainly strings of characters, which may be generated at runtime. In some cases, runtime context determines string values, which implies that our analysis cannot statically find them.

Recall the various forms of ICC. Explicit ICC identifies the communication sink by specifying the target's package and class name. Conversely, implicit ICC identifies the sink through action, category, and/or data fields. Further, a mixed ICC occurs when a source or sink can take on explicit or implicit ICC values depending on the runtime context. Finally, the dynamic receiver ICC occurs when a sink binds to an ICC type through runtime context (e.g., Broadcast Receivers which identify the Intent Filter types when being registered). We seek to determine precise ICC specifications, where all fields of Intents or Intent Filters are known without ambiguity.

As shown in Table 5.1, with respect to the random sample corpus, we were able to provide unambiguous specifications for over 91% of the 7,835 ICC locations in the 348 applications. Explicit ICC was precisely analyzed more frequently (≈98%) than implicit ICC (≈ 88%). The remaining 7% of ICC containing mixed and dynamic receivers proved to be more difficult, where the precision rates are much lower than others. This is likely due to the fact that dynamic receivers involve finding more data than Intents: Intent Filters limiting access to dynamic receivers can define several actions, and receivers can be protected by a permission (which we attempt to recover).

In the popular applications, we obtain a precise specification in over 94% of the 58,989 ICC locations in the 838 applications. Explicit ICC was slightly more precisely analyzed than implicit ICC. Mixed ICC is again hard to recover. This is not surprising, as mixed ICC involves different Intent values on two or more branches, which is indicative of a method more complex than most others.

A facet of the analysis not shown in the table is the number of applications for which we could identify unambiguous specifications for all ICC – called 100% precision. In the random sample, 56% of the applications could be analyzed with 100% precision, 80% of the applications with 90% precision, and 91% of the applications with 80% precision.

	Random Sample				
	Precise	**%**	**Imprecise**	**%**	**Total**
Explicit	3,571	97.65%	86	2.35%	3,657
Implicit	3,225	88.45%	421	11.55%	3,646
Mixed	28	59.57%	19	40.43%	47
Dyn. Rec.	357	73.61%	128	26.39%	485
Total	7,181	91.65%	654	8.35%	7,835

	Popular				
	Precise	**%**	**Imprecise**	**%**	**Total**
Explicit	27,753	94.43%	1,637	5.57%	29,390
Implicit	23,133	93.82%	1,525	6.18%	24,658
Mixed	509	85.12%	89	14.88%	598
Dyn. Rec.	4,161	95.81%	182	4.19%	4,343
Total	55,556	94.18%	3,433	5.82%	58,989

Table 5.1. Precision metrics

In the popular applications, 23% could be analyzed with 100% precision, 82% could be analyzed with 90% precision and 94% with 80% precision. Note that a less-than-100% precision does not mean that the analysis failed. Rather, these are cases where runtime context determines string arguments, and thus *any* static analysis technique would fail.

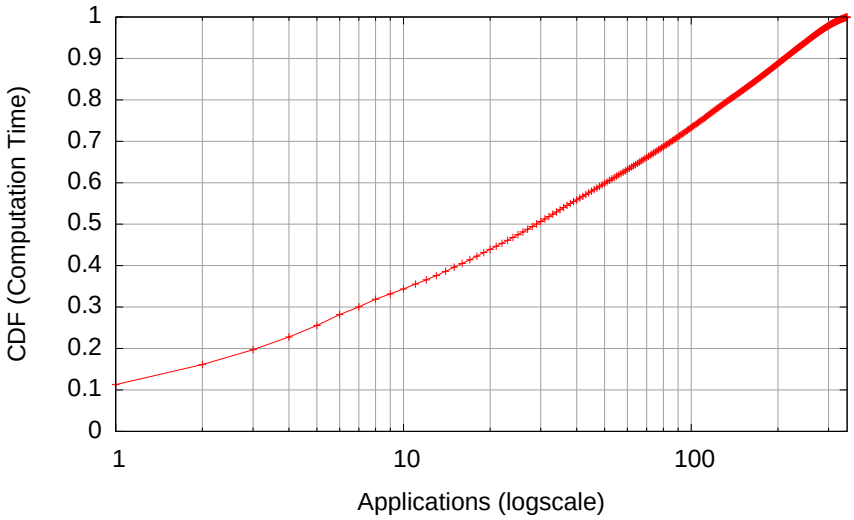
5.4.2 Computational Costs

A second set of tests sought to ascertain the computational costs of performing the IDE analysis using Epicc. For this task we collected measurements at each stage of the analysis and computed simple statistics characterizing the costs of each task on the random sample and the popular applications.

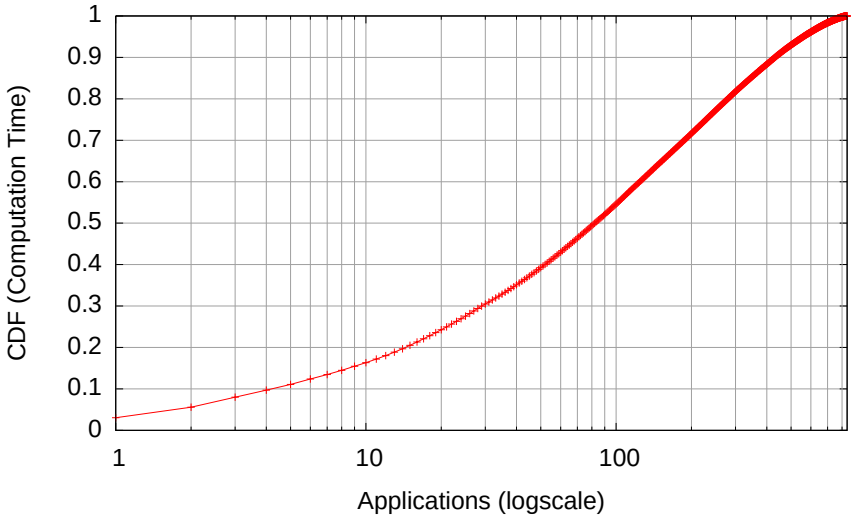
Experiment results show that ICC analysis in this model is feasible for applications in the Google Play store. We were able to perform analysis of all 348 applications in the random sample in about 3.69 hours of compute time. On average, it took just over 38 seconds to perform analysis for a single application, with a standard deviation of 99 seconds. There was high variance in the analysis run times. A CDF (cumulative distribution function) of the analysis computation time for all 348 applications is presented in Figure 5.5(a). It is clear from the figure that costs were dominated by a handful of applications; the top application consumed over 11% of the time, the top 5 consumed over 25% of the total time, and the top 29 consumed over 50% of the total time. These applications are large with a high number of entry points.

Analyzing the 838 popular applications took 33.58 hours, that is, 144 seconds per application. The standard deviation was 277 seconds. The average processing time is significantly higher than for the random sample. However, this is expected, as the average application size is almost 1,500 classes, which is significantly higher than the random sample (less than 400 classes per application). This is likely related to the popularity bias: one can expect frequently downloaded applications to have fully developed features as well as more complex/numerous features, which implies a larger code base. A CDF of the computation time for all 838 applications is presented in Figure 5.5(b). Once again, analysis time is dominated by a few applications. The top 5 consumed over 11% of the analysis time and the top 83 (less than 10% of the sample) consumed over 50% of the analysis time.

Processing was dominated by the standard Soot processing (e.g., translating classes to an intermediate representation, performing type inference and points-to analysis, building a call graph). It consumed 75% of the processing time in the random sample and 86% in the popular applications. It was itself dominated by the translation to Soot's internal



(a) Random sample.



(b) Popular applications.

Figure 5.5. CDF of computation time.

representation and by the call graph construction. The second most time-consuming task was the IDE analysis (which also includes the string analysis in our implementation). It took 15% of the processing time with the random sample and 7% with the popular one. Finally, I/O operations accounted for most of the remainder of the processing time. Loading classes took 7% of the time in the random sample and 3% in the popular one. Database operations accounted for 2% of processing for the random sample and 3% for the popular applications. Other operations (e.g., parsing manifest files) took less than 1% of processing time.

5.4.3 Entry and Exit Point Analysis

This section briefly characterizes the exit (source) and entry (sink) points of Android applications in our data sets. Note that this analysis is preliminary and will be extended in future work.

An exit point is a location that serves as a source for ICC; i.e., the sending of an Intent. In the random sample, our analysis found 7,350 exit points which can transmit 10,035 unique Intent values. About 92% of these exit points had a single Intent specification, with the remaining exit points being able to take on 2 or more values. In two pathological cases, we noted an exit point that could have 640 different Intent values (most likely the result of contrived control flow or multiple aliasing for an Intent value). The popular applications had 48,756 exit points, associated with 316,419 Intent values. Single Intent specifications were found in 90% of exit points. We found 10 pathological cases where an exit point was associated with 512 Intent values or more. The use of key value data was more prevalent than we initially expected, in about 36% of exit points in the random sample. Key-value data was present in Intents in 46% of exit points in the popular applications.

Our study of entry points focused on the sinks of ICC that were either dynamically registered broadcast receivers or component interfaces (exported or not) identified in the application manifest. In the random sample, we were able to identify 3,863 such entry points associated with 1,222 unique intent filters. The popular applications comprised 25,291 entry points with 11,375 Intent Filters. 1,174 components were exported (and thus available to other applications) in the random sample, 7,392 in the popular applications. Of those, only 6% (67) of the exported components were protected by a permission in the random sample and 5% (382) were protected in the popular applications. This is concerning, since the presence of unprotected components in privileged applications can lead to confused deputy [72] attacks [5].

Oddly, we also found 23 components that were exported without any Intent Filter in the random sample and 220 in the popular sample. Conversely, we found 32 cases where a component had an Intent Filter but was not exported in the random sample and 412 in the popular one. The latter indicates that developers sometimes use implicit Intents to address components within an application, which is a potential security concern, since these Intents may also be intercepted by other components. Lastly, application entry points were relatively narrow (with respect to intent types). Over 97% of the entry points received one Intent type in the random sample. Single Intent Filters were found in 94% of components protected by Intent Filters in the popular applications.

5.4.4 ICC Vulnerability Study

In this section, we perform a study of ICC vulnerabilities in our samples using Epicc and compare our results with ComDroid [4]. We look for the same seven vulnerabilities as in [4]. Activity and Service hijacking can occur when an Intent is sent to start an Activity or a Service without a specific target. Broadcast thefts can happen when an Intent is Broadcast without being protected by a signature or signatureOrSystem permission[6]. In all three cases, the Intent may be received by a malicious component, along with its potentially sensitive data.

Malicious Activity or Service launch and Broadcast injection are Intent spoofing vulnerabilities. They indicate that a public component is not protected with a signature or signatureOrSystem permission. It may be started by malicious components. These vulnerabilities can lead to permission leakage [5, 76, 77].

Finally, some Intent Broadcasts can only be sent by the operating system, as indicated by their action field. Broadcast Receivers can register to receive them by specifying Intent Filters with the appropriate action. However, these public components can still be addressed directly by explicit Intents. That is why the target Receivers should check the action field of the received Intent to make sure that it was sent by the system.

Table 5.2 shows the results of the study for the random and the popular samples. The first line shows the number of vulnerabilities identically detected by both analyses, the second line shows vulnerabilities detected by ComDroid only and the third line shows vulnerabilities detected by Epicc only. The last two lines show the total number of vulnerabilities found by each tool. For the three unauthorized Intent receipt vulnerabilities (first three columns), both ComDroid and Epicc indicate whether the sent

[6]The signature permission protection level only allows access to a component from an application signed by the same developer. The signatureOrSystem protection level additionally allows the operating system to start the component.

Vulnerability	Activity Hijacking		Service Hijacking		Broadcast Theft		Activity Launch		Service Launch		Broadcast Injection		System Broadcast w/o action check		Total vulnerabilities	
Sample	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P
Identical	2,591	15,214	78	1,200	503	4,825	179	1,731	23	263	273	3,503	30	126	3,677	26,862
ComDroid only	916	7,717	78	535	218	2,854	12	169	2	18	104	1,684	3	20	1,333	12,997
Epicc only	181	2,079	3	151	23	297	4	20	0	1	4	43	77	580	292	3,171
Total ComDroid	3,507	22,931	156	1,735	721	7,679	191	1,900	25	281	377	5,187	33	146	5,010	39,859
Total Epicc	2,772	17,293	81	1,351	526	5,122	183	1,751	23	264	277	3,546	107	706	3,969	30,033

Table 5.2. ICC vulnerability study results for the random sample (R) and the popular applications (P).

Intent has extra data in the form of key-value pairs, and whether the Intent has the `FLAG_GRANT_READ_URI_PERMISSION` or `FLAG_GRANT_WRITE_URI_PERMISSION` flags. These flags are used in Intents that refer to Content Provider data and may allow the recipient to read or write the data [4].

For the presence of flags and the detection of extra data, Epicc can precisely indicate when the value of an Intent depends on the execution path. On the other hand, a ComDroid specification does not make this distinction. When Epicc and ComDroid differ for a code location, we include flags in both the "ComDroid only" and "Epicc only" rows of Table 5.2.

The Activity hijacking vulnerabilities found by both ComDroid and Epicc are unsurprisingly common: they represent all cases where implicit Intents are used to start Activities. Service hijacking vulnerabilities are much less prevalent, which is correlated with the fact that Services are used less often than Activities. Broadcast theft vulnerabilities are quite common as well. As previously described in Section 5.4.3, few exported components are protected by permissions. Therefore, the high number of malicious Activity or Service launch as well as Broadcast injection vulnerabilities is not surprising. Note the discrepancy between the number of components without permissions and the total number of these vulnerabilities. A large portion of the components not protected by permissions are Activities with the `android.intent.action.MAIN` action and the `android.intent.category.LAUNCHER` category, which indicate that these components cannot be started without direct user intervention. They are therefore not counted as potential vulnerabilities.

If we consider the first three vulnerabilities (unauthorized Intent receipt), we can see that ComDroid flags a high number of locations where Epicc differs. A manual examination of a random subset of applications shows that these differences are either false positives detected by ComDroid or cases where Epicc gives a more precise vulnerability specification. We observed that a number of code locations are detected as vulnerable by ComDroid, whereas Soot does not find them to be reachable. Epicc takes advantage from the sound and precise Soot call graph construction to output fewer false positives. Additionally, the IDE model used by Epicc can accurately keep track of differences between branches (e.g., explicit/implicit Intent or URI flags), whereas ComDroid cannot. Note that when an Intent is implicit on one branch and explicit on another, ComDroid detects it as explicit, which is a false negative. On the other hand, the IDE model correctly keeps track of the possibilities.

With a few exceptions, the ComDroid and Epicc analyses detect the same possible

malicious Activity and Service launches. That is expected, since both are detected by simply parsing the manifest file. The few differences can be explained by minor implementation differences or bugs in pathological cases. The Broadcast injection vulnerability shows stronger differences, with ComDroid detecting 377 cases for the random sample and 5,187 for the popular one, whereas Epicc only finds 277 and 3,546, respectively. Some of the Broadcast injections detected by ComDroid involved dynamically registered Broadcast Receivers found in unreachable code. Once again, the call graph used by Epicc proves to be an advantage. Many other cases involve Receivers listening to protected system Broadcasts (i.e., they are protected by Intent Filters that only receive Intents sent by the system). The list of protected Broadcasts used by ComDroid is outdated, hence the false positives.

Finally, there is a significant difference in the detection of the system Broadcasts without action check, with Epicc detecting 107 vulnerabilities in the random sample and 706 in the popular one, whereas ComDroid only detects 33 and 146, respectively. The first reason for that difference is that the ComDroid list of protected Broadcasts is outdated. Another reason is an edge case, where the Soot type inference determines Receivers registered using a *registerReceiver*() method as having type `android.content.BroadcastReceiver` (i.e., the abstract superclass of all Receivers). It occurs when several types of Receivers can reach the call to *registerReceiver*(). Since no Receiver code can be inspected, even though there may be a vulnerability, our analysis conservatively flags it as a vulnerability.

Overall, Epicc detects 34,002 potential vulnerabilities. On the other hand, ComDroid detects 44,869 potential security issues, that is, 32% more than Epicc. As detailed above, the extra flags found by ComDroid that we checked were all false positives. Further, the potential causes of unsoundness in Epicc (i.e., Java Native Interface, reflection and entry point handling) are also handled unsoundly in ComDroid. Thus, we do not expect the locations flagged by ComDroid but not by Epicc to be false negatives. The precision gain over ComDroid is significant and will help further analyses. Note that it is possible that both tools have false negatives in the presence of Java Native Interface, reflection, or when the life cycle is not properly approximated. In particular, we found that 776 out of the 838 popular applications and 237 out of 348 applications in the random sample make reflective calls. Future work will seek to quantify how often these cause false negatives in practice. We will also attempt to determine if the locations flagged by Epicc are true positives.

Chapter 6

Inter-Component Communication Analysis with the COAL Constant Propagation Language

Chapter 5 has presented an approach to infer the values of the main ICC objects. Unfortunately, it only handles a subset of ICC messages. It only addresses *Intent* messages for which all fields are simple constant values. Adding support for *URIs* – the other large class of ICC messages – using the same approach as for Intent would result in a significant increase in the complexity of the formulation and implementation of the corresponding data flow functions. Thus, while the approach used for Epicc can in theory be used for other messages as well, it is not feasible in practice. Further, Epicc relies on a very limited analysis to determine the values of arguments to calls to the ICC API. Almost all arguments to ICC methods are strings of characters. Unfortunately, Epicc fails at resolving any argument that is not a constant. For example, operations such as string concatenation cause Epicc to assume that an argument can be any possible string, resulting in a significant loss of precision.

In this chapter, we generalize the analysis introduced in Chapter 5 to a large class of interprocedural constant propagation analyses. Unlike most constant propagation analyses, we attempt to find all possible values of ICC objects at important program points, making our analysis *multi-valued*. Our analysis targets *multi-field* constants, i.e., we determine the values of complex objects that may have multiple fields. We express data flow functions in terms of simple *field transformers*, which express how fields are changed by program statements. Taking advantage of the ease of specifying

field transformers, we design the COAL language for specifying multi-valued, multi-field (MVMF) constant propagation problems. In this language, we model all ICC messages with only about 750 lines of COAL specification. The model used by Epicc, on the other hand, requires about 5000 lines of code, for a much more limited coverage of ICC. Since Android ICC messages heavily rely on strings of characters, we additionally develop and implement a string analysis that is both efficient and precise. We compute ICC specifications in 350 applications from the official Play store. We are able to obtain precise ICC specifications in 86% of cases. Epicc, on the other hand, can only infer 64% precisely. The remaining 14% of values cannot be determined because of constructs not yet handled by our string analysis, cases where the values cannot be known statically and other pathological cases. Computing ICC values is efficient, taking only less than a minute per application on average. The extra precision in inferring ICC values directly translates to a significant increase in precision when matching components that send messages with potential receivers. In our experiments, such a matching yields 281361 links with Epicc specifications, whereas values computed with our new tool produce 127204 potential links. We make the following contributions in this chapter:

- We introduce COAL, a novel declarative language to specify MVMF constant propagation problems and query their solution.
- We formally define an approach to solve MVMF constant propagation problems. We implement a COAL solver based on this formalism and make its source code available at `http://siis.cse.psu.edu/coal/`.
- We develop the IC3 tool for finding the values of ICC objects in Android. It is based on a COAL specification and uses our new string analysis. We make its source code available at `http://siis.cse.psu.edu/ic3/`.
- Using the ICC values we find, we perform a preliminary study of component connectivity. We find that ICC is concentrated to a small set of components. 28.65% of components are completely isolated and only 2.69% (resp. 9.33%) of components account for 50% of all potential outgoing (resp. incoming) links.

This chapter is organized as follows. Section 6.1 shows an overview of the problem we are trying to solve. Section 6.3 presents our IDE model for MVMF constant propagation problems. Section 6.2 describes the COAL language for specifying constant propagation problems. In Section 4.8, we evaluate our implementation of ICC analysis using COAL.

6.1 Overview

The Multi-Valued Multi-Field constant propagation problem: Consider `OBJ` an object of type `class Pair{int X; int Y;}`. Assume that at some program location `OBJ` can be either `(X, Y) = (1,10)` or `(2,20)`. We would like an analysis that can determine this fact. Classical constant analysis applied for each field fails at determining a useful value because none of the fields is the same constant across all paths. Multi-Valued constant analysis could determine that `OBJ.X` $\in \{1, 2\}$ and `OBJ.Y` $\in \{10, 20\}$. These constraints accurately describe the individual fields, but they allow for imprecision in the object, because they allow the possibility that `OBJ`= `(1,20)`. We define the *Multi-Valued Multi-Field (MVMF)* constant propagation problem to be the problem of determining the set of values that an object *viewed as a tuple* (such as `(X, Y)`) can have. Note that the above Multi-Valued constant analysis applied to individual fields is a possible solution for MVMF, it may just not be precise enough for certain analyses. In the context of Android we often need more accurate solutions. We will show how to efficiently find such solutions.

We now introduce a running example that will be used throughout. Figure 6.1 shows code for a simple Intent class that contains data used for passing messages between application components. Note that this is a very simplified version of the actual Intent class used for Android ICC that we presented in Section 2.1. In particular, we omit the *data* field, as we deal with the *extras* fields in a similar manner. The values of the fields in an Intent object determine the target of the message and may also carry data between components. Figure 6.2 defines method $sendMessage()$, which we assume to be called as part of an Android application. This method creates an Intent object and adds a value to the *categories* field of the Intent at Line 4. Then, depending on the value of a boolean, one of two things can happen. In both branches, a value is added to the *categories* field. In the first branch after the `if` statement, the *action* field of *intent* is replaced with the value of the *action* field of *src* (Line 6). In the fall-through branch, the *action* of the *intent* is set to a constant value (Line 9). Next, the values in the *extras* field of the *src* Intent argument are added to the *extras* of the newly created Intent using $putExtras()$. Next, value `EXT_1` is removed from the *extras* field of *intent*. Finally, the Intent object is sent to another component using the $startActivity()$ method.

The data flow problem we are solving is to determine all the possible values of the fields of *intent* at the call to the $startActivity()$ method. In our constant propagation framework defined below, the problem can be specified using COAL, a declarative language we designed for this purpose. *The function of COAL (COnstant propAgation*

```
1  package android.content;
2  public class Intent {
3    private String action;
4    private Set<String> categories = new HashSet<String>();
5    private Set<String> extras = new HashSet<String>();
6
7    public void setAction(String value) {
8      this.action = value;
9    }
10   public void addCategory(String value) {
11     this.categories.add(value);
12   }
13   public void putExtras(Intent src) {
14     // Add all values from the extras field of src
15     // into this.extras.
16     this.extras.addAll(src.extras);
17   }
18   public void removeExtra(String name) {
19     this.extras.remove(name);
20   }
21   public String getAction() {
22     return this.action;
23   }
24 }
```

Figure 6.1. Simplified Intent class (unused methods omitted for conciseness).

Language) is to specify Multi-Valued, Multi-Field (MVMF) constant propagation problems. It specifies the types of variables for which values should be inferred and how these values are modified by program statements. It enables abstract reasoning on the semantics of API methods. The COAL language is recognized by our COAL solver, which outputs solutions for many propagation problems solely from their COAL specification.

Figure 6.3 shows how to specify the problem with our framework using COAL in order to get the desired solution. It is composed of field declarations, sinks, a source and a hotspot. The field declarations specify the field that are being tracked, as well as their type. Note that, for each field, we keep track of sets of values, even though the field declaration only specifies the type of each individual field value. The first sink indicates how the *setAction*() method influences the modeled value of an Intent object. A sink specification starts with the signature of the method modeled by the sink. Each line in a sink declaration is an argument whose value is used to modify the Intent value. Each argument declaration is composed of several attributes. The integer

```
1  public void sendMessage(Context context, boolean test,
       Intent src) {
2    Intent intent = new Intent();
3    intent.addCategory("CAT_1");
4    if (test) {
5      intent.addCategory("CAT_2");
6      intent.setAction(src.getAction());
7    } else {
8      intent.addCategory("CAT_3");
9      intent.setAction("ACT_3");
10   }
11   intent.putExtras(src);
12   intent.removeExtra("EXT_1");
13   context.startActivity(intent);
14 }
```

Figure 6.2. Message-passing code. We assume that the *extras* field of the argument Intent *src* contains *either* a single value `EXT_1`, *or* a single value `EXT_3`. We also assume that the *action* field of *src* has value `ACT_1`.

declares the position of the argument in the array of arguments to the method, with indices starting at 0. After the argument index, an operation and a field are declared. They describe both the field that is modified by the method and how it is modified. For example, in the *setAction*() sink, `0: replace action` means that the *action* field is replaced with the value of the first argument to *setAction*(). Other sinks are declared in a similar manner, except when the type of an argument is a class that is modeled with COAL. In that case, a `type` attribute is used in order to specify which field of the argument object is used. For example, in the *putExtras*() sink, the `0: add extras, type android.content.Intent:extras` argument means that the *extras* field of the Intent argument is being used. The contents of that field are added to the *extras* field of the Intent being modified.

The source element indicates that the *getAction*() method returns the value of the *action* field of the modeled value. The unique hotspot indicates that we are querying the solution at all calls to the *startActivity*() method. Similarly to the sink declaration, we specify a list of arguments. They describe the arguments whose value we would like to query. In this case, it is the first argument (as described by the `0` attribute), which is an Intent object, as declared by the `type android.content.Intent` attribute. The COAL language that we define to specify models is described in Section 6.2. The COAL solver that we use to solve MVMF constant propagation problems is described in Section 6.3.

Table 6.1 shows the expected result of our analysis. We want our analysis to recover

```
class android.content.Intent {
 String action;
 String categories;
 String extras;

 sink <android.content.Intent: void setAction(java.lang.String)> {
  0: replace action;
 }
 sink <android.content.Intent: void addCategory(java.lang.String)> {
  0: add categories;
 }
 sink <android.content.Intent: void removeExtra(java.lang.String)> {
  0: remove extras;
 }
 sink <android.content.Intent: void putExtras(android.content.Intent)> {
  0: add extras, type android.content.Intent:extras;
 }
 source <android.content.Intent: java.lang.String getAction()> {
  action;
 }
 hotspot <android.content.Context: void startActivity(android.content.Intent)> {
  0: type android.content.Intent;
 }
}
```

Figure 6.3. COAL model for the constant propagation problem. Each sink specification describes the influence of a method call on the fields of an Intent. The source specification describes how values flow out of modeled objects. The single hotspot is used to query the value of the Intent at the call to *startActivity*().

	action	categories	extras
Value 1	ACT_1	{CAT_1, CAT_2}	∅
Value 2	ACT_1	{CAT_1, CAT_2}	{EXT_3}
Value 3	ACT_3	{CAT_1, CAT_3}	∅
Value 4	ACT_3	{CAT_1, CAT_3}	{EXT_3}

Table 6.1. Possible values of the fields of *intent* at the *startActivity*() method call in Figure 6.2. The first two values correspond to the first branch after the `if` statement (Lines 5 and 6 in (b)). Value 1 represents the case where the *extras* field in *src* has a value `EXT_1`, whereas Value 2 is for the case where the field value is `EXT_3`. Values 3 and 4 are similar, except that they account for the fall-through branch (Lines 8 and 9) of the `if` statement.

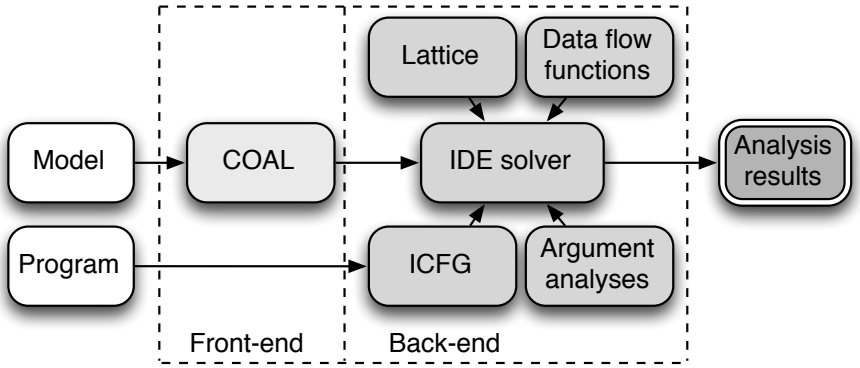


Figure 6.4. Overview of our analysis process.

the four possible values of Intent *intent*. These values correspond to all possible execution paths of the program from Figure 6.2. We wish to recover exactly these possible values, and we do not want all the possible combinations of fields. For example, it is not possible in our problem to have an Intent value with action `ACT_1` and categories {`CAT_1`, `CAT_3`}. As a result, our analysis does not simply track fields individually as separate variables, which is why we say that our analysis is *multi-field*.

Figure 6.4 shows the overview of our analysis process. It is designed to allow constant propagation analyses to be specified easily. The user simply inputs a model for the problem being solved and the program for which the data flow problem should be solved. The model can be specified through the use of the COAL language. The language allows the specification of the fields of classes being modeled, functions modifying the fields, and program locations where constants should be determined.

The back end converts the input program into an *Interprocedural Control Flow Graph (ICFG)*. An ICFG is a collection of CFGs of all the procedures in the program connected

with each other as appropriate at procedure call sites. The back end integrates a generic lattice of values that is appropriate for MVMF constant propagation problems. It also uses generic data flow functions that can be used for many propagation problems. For each problem being solved, the model input by the user is used to instantiate the proper lattice and functions. These are input with the ICFG into a solver for Interprocedural Distributive Environment (IDE) problems. We present the generic IDE model for constant propagation in Section 6.3.

Finally, since the values of arguments to functions have to be known in order to determine MVMF constants, the IDE solver also uses argument value analyses. In particular, we introduce a string analysis that is finely tuned for Android ICC [134]. It is sound, precise and efficient.

The back end outputs the analysis results. The COAL language allows specification of program points of interest (hotspots) where the MVMF constant values should be computed. This is useful when the solution should be computed for specific program points only. It is appropriate in Android, where the program points of interest (sending or receiving ICC messages) are known in advance. The results can then be output in a simple text format or accessed using a programmatic interface (API). We expect that for most problems the language will allow results to be collected at appropriate locations. However, for problems where this is not sufficient, we also allow lower-level queries to the IDE solver as part of the COAL solver API.

6.2 The Coal Language

We introduce the COAL language to specify and query a wide variety of MVMF constant propagation problems. COAL specifications are used by our constant propagation solver to automatically instantiate the lattice of values and the appropriate data flow functions, as we describe in Section 6.3. A simplified grammar for this language is presented on Figure 6.5. The {} characters symbolize repetition, while `[]` characters surround optional parts of a production.

The model for a given object is composed of field declarations, sinks, hotspots, constants and sources.

Field declarations – A field declaration specifies a field that is part of the modeled class. It describes a data type and a name for the field. In Figure 6.5, we use non-terminals ⟨*java type*⟩ and ⟨*field name*⟩ to represent valid Java types and field names.

Sinks – Sinks represent method calls where constant values flow to the modeled object.

⟨*model*⟩ ::= '`class`' ⟨*java type*⟩ '`{`' { ⟨*field*⟩ | ⟨*sink*⟩ | ⟨*hotspot*⟩ | ⟨*constant*⟩ | ⟨*source*⟩ } '`}`'

⟨*field*⟩ ::= ⟨*java type*⟩ ⟨*field name*⟩ '`;`'

⟨*sink*⟩ ::= '`sink`' ⟨*method sig*⟩ '`{`' { ⟨*sink arg*⟩ } '`}`'

⟨*hotspot*⟩ ::= '`hotspot`' ⟨*method sig*⟩ '`{`' { ⟨*hotspot arg*⟩ } '`}`'

⟨*constant*⟩ ::= '`constant`' ⟨*field sig*⟩ '`{`' { ⟨*field name*⟩ '`=`' ⟨*inline value*⟩ '`;`' } '`}`'

⟨*source*⟩ ::= '`source`' ⟨*method sig*⟩ '`{`' ⟨*field name*⟩ '`;`' '`}`'

⟨*sink arg*⟩ ::= [⟨*arg number*⟩ '`:`'] ⟨*operations*⟩ ⟨*field*⟩ ['`,`' ⟨*arg type*⟩ '`:`' ⟨*field name*⟩]

⟨*hotspot arg*⟩ ::= ⟨*arg number*⟩ '`:`' ⟨*arg type*⟩

⟨*arg number*⟩ ::= ⟨*integer*⟩ | '`(`' ⟨*integer*⟩ {'`,`' ⟨*integer*⟩ } '`)`'

⟨*arg type*⟩ ::= '`type`' ⟨*java type*⟩

⟨*field sig*⟩ ::= '`<`' ⟨*java type*⟩ '`:`' ⟨*java type*⟩ ⟨*java field*⟩ '`>`'

Figure 6.5. COAL language for the specification of MVMF constant propagation problems.

The specification of the sinks comprises a method signature (non-terminal ⟨*method sig*⟩) that identifies the method of interest. It also includes a set of arguments that describe how the arguments of the method are used to modify the fields of the modeled object. A sink argument has several attributes. An argument number identifies the method argument of interest. In some cases, several arguments contribute to the value of a single field. That is why the language supports sets of argument numbers. A field operation to be performed is also specified. This allows the solver to create appropriate data flow functions. Natively supported field operations are `add` (add argument value to the field), `remove` (remove argument value from field), `replace` (replace field with argument value) and `clear` (clear field value). A sink specification also includes a field name that identifies the field being modified. In the case where an argument is a class modeled with COAL, an argument type and field name are specified. This indicates to the solver that the value of a field of a modeled class flows to the object being modified.

Hotspots – Hotspots represent statements of interest where modeled values should be determined. At the moment COAL only expresses hotspots that are method invocations, which is sufficient for Android ICC.

Constants – Java allows specification of constants in the form of static final fields. The constants of a class are initialized in the class initializer the first time the class is referenced. A naïve way to deal with constants would consist in tracking the constant creation and initialization the same way it is done for all modeled objects. We would then propagate them throughout the entire program, which would dramatically harm performance. For example, in Android ICC, there are 128 URI constants declared by the Android framework that are used to address specific resources. For instance, the `BOOKMARKS_URI` value declared in the `android.provider.Browser` class allows developers to address a database table containing browser bookmarks and history. As a performance optimization, we allow constant modeled objects to be specified in COAL. Where these values are used, the COAL solver uses the specified value. This allows us to avoid propagation of constants in the entire program, thereby improving performance.

Sources – Sources model the case where a modeled field value flows to an argument value. This occurs for example at Line 6 in Figure 6.2 with statement `intent.setAction(src.getAction())`. Declaring *getAction*() as a source for the *action* field allows the COAL solver to use the fact that the value of the *action* field of *src* flows to the *action* field of *intent*.

Using this language, we can solve MVMF constant propagation problems such as the one from Figure 6.2 by simply using the specification from Figure 6.3. This is considerably simpler than writing a new model with a lattice of values and environment transformers. That is the reason why we are able to model all Android ICC objects (see Section 2.1). A limitation of Epicc is that some objects are not properly modeled, due to the complexity of adding new models, especially when a sink argument uses the value of another modeled object. The next two sections describe how MVMF constant propagation problems are solved, given a COAL specification.

6.3 An IDE Model for MVMF Constant Propagation

In this section, we introduce a model that is used in our COAL solver for MVMF constant propagation problems.

For any set X, we denote the power set of X by $\mathcal{P}(X)$.

6.3.1 The Pointwise Representation of Environment Transformers

We denote the set of functions from L to L by L^L. It can be shown that any environment transformer t can be written in terms of a *pointwise representation* $\mathcal{R}_t$[1], which is a function from $(D \cup \{\Lambda\}) \times (D \cup \{\Lambda\})$ to L^L. Here Λ is a special symbol to indicate a null data flow fact. The pointwise representation is useful because it allows for easy specification of transformers. The pointwise representation answers the following question: given two symbols d' and d, how does the value associated with d' contribute to the value of d? More specifically, for any environment transformer t, for all $e \in Env(D, L)$ and $d \in D$, we have

$$t(e)(d) = \mathcal{R}_t\left(\Lambda, d\right)\left(\bot\right) \sqcup \left(\bigsqcup_{d' \in D} \mathcal{R}_t\left(d', d\right)\left(e\left(d'\right)\right)\right). \tag{6.1}$$

In Section 6.3.3, we will use the pointwise representation, and more specifically we will define the functions in L^L that model the MVMF constant propagation problem. Please refer to Section 6.3.3 for examples of pointwise representations of transformers.

The remainder of this section states a result that links the distributivity of the functions in L^L to the distributivity of environment transformers.

Definition 11. *We say that a function $\mathcal{R}_t : (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \to L^L$ is* codistributive *if all elements of its range are distributive functions from L to L.*

Proposition 2. *If $\mathcal{R}_t : (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \to L^L$ is codistributive, then t defined as in Equation (6.1) is a distributive environment transformer.*

Proof. Let $e_1, e_2, \cdots \in Env(D, L)$ and $\mathcal{R}_t : (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \to L^L$ codistributive. Using Equation (6.1):

$$\begin{aligned} t\left(\bigsqcup_i e_i\right)(d) &= \mathcal{R}_t\left(\Lambda, d\right)\left(\bot\right) \sqcup \left(\bigsqcup_{d' \in D} \mathcal{R}_t\left(d', d\right)\left(\left(\bigsqcup_i e_i\right)\left(d'\right)\right)\right) \\ &= \mathcal{R}_t\left(\Lambda, d\right)\left(\bot\right) \sqcup \left(\bigsqcup_{d' \in D} \mathcal{R}_t\left(d', d\right)\left(\bigsqcup_i \left(e_i\left(d'\right)\right)\right)\right) \end{aligned}$$

by definition of $\left(\bigsqcup_i e_i\right)\left(d'\right)$. Since $\mathcal{R}_t$ is distributive, we have:

$$t\left(\bigsqcup_i e_i\right)(d) = \mathcal{R}_t\left(\Lambda, d\right)\left(\bot\right) \sqcup \left(\bigsqcup_{d' \in D} \left(\bigsqcup_i \mathcal{R}_t\left(d', d\right)\left(e_i\left(d'\right)\right)\right)\right).$$

[1]The exact expression of $\mathcal{R}_t$ is not needed for this section. Interested readers are referred to [12].

Using the commutativity of the $\sqcup$ operator, we get:

$$\begin{aligned} t\left(\bigsqcup_i e_i\right)(d) &= \mathcal{R}_t(\Lambda, d)(\bot) \sqcup \left(\bigsqcup_i \left(\bigsqcup_{d' \in D} \mathcal{R}_t(d', d)\left(e_i(d')\right)\right)\right) \\ &= \bigsqcup_i \left(\mathcal{R}_t(\Lambda, d)(\bot) \sqcup \left(\bigsqcup_{d' \in D} \mathcal{R}_t(d', d)\left(e_i(d')\right)\right)\right) \\ &= \bigsqcup_i t(e_i)(d) \end{aligned}$$

by using the idempotence and the commutativity of $\sqcup$ and Equation (6.1). □

In the next sections we will define distributive functions from L to L. This result guarantees that the environment transformers defined using these pointwise representations are distributive.

6.3.2 The L Lattice of Values

We are trying to determine the value of an object with n fields. Let $V_1, \ldots, V_n$ be finite sets. For $i \in \{1, \ldots, n\}$, let $P_i = \mathcal{P}(V_i) \cup \{\omega\}$, where ω represents an undefined value. Let $B = P_1 \times \cdots \times P_n$. We define $L = (\mathcal{P}(B), \subseteq)$ a join-semilattice with a bottom element $\bot = \varnothing$. The join operation on L is the set union $\cup$. The top element of L is the set of all elements in B.

Sets $V_1, V_2, \ldots, V_n$ are the domains of the values we are trying to determine. For example, V_1 could be the set of constant strings of characters in the program, and V_2 could be the set of integer constants in the program. A value in B represents a value as it is seen on a single path. Finally, values in L represent values of objects taking into account several paths of a program.

Let us consider the example shown on Figure 6.1. We are interested in three fields: *action*, *categories* and *extras*. Let $\mathcal{S}$ be the set of string constants in the program. In this case, we consider $P_1 = P_2 = P_3 = \mathcal{P}(\mathcal{S})$. In other words, we consider all three fields to take values in the power set of $\mathcal{S}$. We have $B = P_1 \times P_2 \times P_3$ and $L = (\mathcal{P}(B), \subseteq)$.

In method $sendMessage()$, the value associated with the *intent* variable is initially $\bot$ before Line 2. Line 2 transforms this value to $\{(\varnothing, \varnothing, \varnothing)\}$. Right after Line 3, the value is

$$\{(\varnothing, \{\texttt{CAT\_1}\}, \varnothing)\}. \tag{6.2}$$

In the first branch of the `if` statement, the value associated with *intent* is transformed

to

$$\{(\{\texttt{ACT\_1}\}, \{\texttt{CAT\_1}, \texttt{CAT\_2}\}, \varnothing)\}, \tag{6.3}$$

where the `ACT_1` value comes from the *action* field of *src*. In the fall-through branch of the `if` statement, this value becomes

$$\{(\{\texttt{ACT\_3}\}, \{\texttt{CAT\_1}, \texttt{CAT\_3}\}, \varnothing)\}. \tag{6.4}$$

When the two branches merge, at Line 10, the value becomes

$$\{(\{\texttt{ACT\_1}\}, \{\texttt{CAT\_1}, \texttt{CAT\_2}\}, \varnothing), (\{\texttt{ACT\_3}\}, \{\texttt{CAT\_1}, \texttt{CAT\_3}\}, \varnothing)\}, \tag{6.5}$$

which is the set union of the values given by Equations (6.3) and (6.4).

After Line 11, the value of *intent* becomes:

$$\begin{aligned}\{&(\{\texttt{ACT\_1}\}, \{\texttt{CAT\_1}, \texttt{CAT\_2}\}, \{\texttt{EXT\_1}\}),\\ &(\{\texttt{ACT\_1}\}, \{\texttt{CAT\_1}, \texttt{CAT\_2}\}, \{\texttt{EXT\_3}\}),\\ &(\{\texttt{ACT\_3}\}, \{\texttt{CAT\_1}, \texttt{CAT\_3}\}, \{\texttt{EXT\_1}\}),\\ &(\{\texttt{ACT\_3}\}, \{\texttt{CAT\_1}, \texttt{CAT\_3}\}, \{\texttt{EXT\_3}\})\}.\end{aligned}$$

We have used the fact that the *extras* field of *src* contains either value `EXT_1` or `EXT_3`. Finally, after Line 12, this value becomes:

$$\begin{aligned}\{&(\{\texttt{ACT\_1}\}, \{\texttt{CAT\_1}, \texttt{CAT\_2}\}, \varnothing),\\ &(\{\texttt{ACT\_1}\}, \{\texttt{CAT\_1}, \texttt{CAT\_2}\}, \{\texttt{EXT\_3}\}),\\ &(\{\texttt{ACT\_3}\}, \{\texttt{CAT\_1}, \texttt{CAT\_3}\}, \varnothing),\\ &(\{\texttt{ACT\_3}\}, \{\texttt{CAT\_1}, \texttt{CAT\_3}\}, \{\texttt{EXT\_3}\})\}.\end{aligned} \tag{6.6}$$

6.3.3 Transformers on L

The intuition behind the COAL language is that each argument in a COAL sink represents the influence of a method call on a field. Accordingly, we introduce transformers that are defined at the granularity of fields. In this section, when we construct transformers we assume that the value of *src* is available where necessary. We revisit this assumption in Section 6.3.4.

Definition 12. *For $i \in \{1, \ldots, n\}$, we define $F_i = \{\phi_i : P_i \to P_i\}$, the set of functions from P_i to P_i. Each ϕ_i is called a* **field transformer**. *We say that ϕ_i is a* **linear field**

transformer *if* $\phi_i(\omega) = \omega$ *and* ϕ_i *can be written as either:*

- *For all* $X \neq \omega$, $\phi(X) = (X \cup A_i) \backslash R_i$, *for some constant sets* A_i *and* R_i *in* P_i. *Such a function will also be denoted as* $\phi_i = \phi_{A_i}^{R_i}$.
- *For all* $X \neq \omega$, $\phi(X) = A_i$, *for some* $A_i \in P_i$. *This case is also denoted by* $\phi_i = \phi_{A_i}$.

Let $\mathcal{L}_i \subset F_i$ *be the set of linear field transformers from* P_i *to* P_i.

Let us denote the identity field transformer by *id*. We have $id = \phi_{\varnothing}^{\varnothing} \in \mathcal{L}_i$. The important idea is that each sink argument in COAL is mapped to a single field transformer. For example, let us consider the statement in Line 3 in Figure 6.2. It is modeled by the sink at Line 8 of Figure 6.3. The COAL solver interprets that sink in two steps. First, using the fact that the *categories* are strings of characters (as declared in Line 3 of Figure 6.3), it triggers a string analysis [134]. The string analysis determines that argument 0 of the *addCategory*() method has value `CAT_1`. In a second step, it uses the fact that the sink argument performs an `add` operation to generate field transformer $\phi_{\{CAT\_1\}}^{\varnothing}$.

We now show how we use field transformers as basic building blocks for data flow functions. We first gather field transformers for each field, building functions that represent the execution of a single path for all fields. We define the set $\mathcal{L}$ of functions from B to B such that for any $l \in \mathcal{L}$, there exists $(\phi_1, \ldots, \phi_n) \in \mathcal{L}_1 \times \cdots \times \mathcal{L}_n$ such that, for any $b = (\beta_1, \ldots, \beta_n) \in B$, $l(b) = (\phi_1(\beta_1), \ldots, \phi_n(\beta_n))$. We note $l = \phi_1 \times \cdots \times \phi_n$. Recall that the influence of the statement at Line 3 of Figure 6.2 on field *categories* is modeled by field transformer $\phi_{\{CAT\_1\}}^{\varnothing}$. Since the corresponding COAL sink has no additional argument, the COAL solver models the influence of the statement on all fields with $id \times \phi_{\{\texttt{CAT\_1}\}}^{\varnothing} \times id \in \mathcal{L}$.

Functions in $\mathcal{L}$ model the influence of a single execution path. We now define a set F of functions from L to L using functions in $\mathcal{L}$. Functions in F model the influence of several execution paths on all fields of an object. More specifically, any $f \in F$ is written $f = \{l_1, \ldots, l_m\}$, with $l_1, \ldots, l_m \in \mathcal{L}$, such that:

- for any $b \in B$, $f(\{b\}) = l_1(b) \cup \cdots \cup l_m(b)$,
- for any $v = \{b_1, \ldots, b_k\} \in L$, $f(b) = f(\{b_1\}) \cup \cdots \cup f(\{b_k\})$,
- f is the identity over L, denoted by id_L,

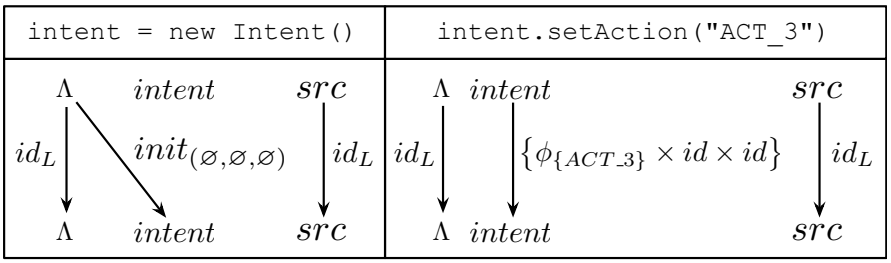


Figure 6.6. Transformers for statements from Figure 6.2.

- for all $v \in L$, $f(v) = \bot$. This function is denoted by $f = \Omega$. Informally, the Ω function is used to "kill" data flow facts.
- $f(\bot) = \{b\}$, with $b \in B$. This special function is denoted by $f = init_b$. Informally, *init* functions generate data flow facts and associate them with an initial value.

Let us now consider the `if` statement in Figure 6.2. The influence of the first branch can be summarized by function $\phi_{\{\texttt{ACT\_1}\}} \times \phi^{\varnothing}_{\{\texttt{CAT\_2}\}} \times id \in \mathcal{L}$, using the fact that the *action* field of *src* has value `ACT_1`. The second branch can be summarized by $\phi_{\{\texttt{ACT\_3}\}} \times \phi^{\varnothing}_{\{\texttt{CAT\_3}\}} \times id \in \mathcal{L}$. The influence of the two branches is summarized by

$$\left\{\phi_{\{\texttt{ACT\_1}\}} \times \phi^{\varnothing}_{\{\texttt{CAT\_2}\}} \times id, \phi_{\{\texttt{ACT\_3}\}} \times \phi^{\varnothing}_{\{\texttt{CAT\_3}\}} \times id\right\} \in F.$$

We can verify that applying this function to the value given by Equation (6.2) yields the value given by Equation (6.5).

We define environment transformers by their pointwise representation $\mathcal{R}_t$ using functions in F. Examples of environment transformers with their representation are shown in Figure 6.6. For example, for statement `intent = new Intent()`, the representation $\mathcal{R}_t$ for the corresponding transformer is defined as:

$$\mathcal{R}_t(d', d) = \begin{cases} id_L & \text{if } (d', d) = (\Lambda, \Lambda) \text{ or } (d', d) = (src, src) \\ init_{(\varnothing,\varnothing,\varnothing)} & \text{if } (d', d) = (intent, \Lambda) \\ \Omega & \text{otherwise} \end{cases}$$

This function describes the relationships between symbols before the statement (d) with symbols after the statement (d'). The first case (id_L) means that we are propagating the values of Λ (the empty data flow fact) and *src* without any changes. The second case means that we are creating a new data flow fact *intent*, as indicated by the edge between Λ and *intent*. We are associating function $init_{(\varnothing,\varnothing,\varnothing)}$ with that edge. Since the

value associated with Λ is $\perp$, this informally means that the contribution of Λ to the final value of *intent* is $init_{(\varnothing,\varnothing,\varnothing)}(\perp) = \{(\varnothing, \varnothing, \varnothing)\}$ (see Equation (6.1) in Section 6.3.1). The final case (Ω) means that there exists no relationship between any other symbol.

For statement `intent.setAction("ACT_3")`, the pointwise representation $\mathcal{R}_t$ for the corresponding transformer is defined as:

$$\mathcal{R}_t(d', d) = \begin{cases} id_L & \text{if } (d', d) = (\Lambda, \Lambda) \\ & \text{or } (d', d) = (src, src) \\ \{\phi_{\{ACT\_3\}} \times id \times id\} & \text{if } d' = d = intent \\ \Omega & \text{otherwise} \end{cases}$$

This is very similar to the previous case, except for *intent*. If we assume the value of *intent* to be v before the statement, then the value of *intent* after the statement is given by $\{\phi_{\{ACT\_3\}} \times id \times id\}(v)$.

Transformers are defined that way for all statements of interest in the program.

Proposition 3. *All elements of F are distributive functions.*

The proof of this proposition is trivial, given the definition of the functions in F. Since all elements in F are distributive, according to Proposition 2, the resulting environment transformers are distributive. It follows that the data flow problem can be solved using existing algorithms from [12].

It is worthwhile to compare the model we have just presented to an alternative model where each field of each object is a separate symbol in D. Such a model is conceptually simpler and also simpler to implement. However, our model is more precise, since it keeps track of the correlation across fields. Let us take the example of a program with two branches, where we are modeling an object with two fields. Let us assume that on one branch, the value of the object is (A, B) and on the other branch it is (A', B'). Our algorithm can determine that these are the only possible values. On the other hand, the simpler model would just determine that the two possible values are A and A' for the first field, and B and B' for the second field. Thus this model would consider values (A, B') and (A', B) to be feasible, whereas in reality they are not. Our model is therefore more precise. Returning to our example from Figure 6.2, the alternative model would

compute that the final value is:

$$\begin{aligned}
&\{(\{\texttt{ACT_1}\}, \{\texttt{CAT_1}, \texttt{CAT_2}\}, \varnothing),\\
&\ (\{\texttt{ACT_1}\}, \{\texttt{CAT_1}, \texttt{CAT_2}\}, \{\texttt{EXT_3}\}),\\
&\ (\{\texttt{ACT_3}\}, \{\texttt{CAT_1}, \texttt{CAT_3}\}, \varnothing),\\
&\ (\{\texttt{ACT_3}\}, \{\texttt{CAT_1}, \texttt{CAT_3}\}, \{\texttt{EXT_3}\}),\\
&\ (\{\texttt{ACT_3}\}, \{\texttt{CAT_1}, \texttt{CAT_2}\}, \varnothing),\\
&\ (\{\texttt{ACT_3}\}, \{\texttt{CAT_1}, \texttt{CAT_2}\}, \{\texttt{EXT_3}\}),\\
&\ (\{\texttt{ACT_1}\}, \{\texttt{CAT_1}, \texttt{CAT_3}\}, \varnothing),\\
&\ (\{\texttt{ACT_1}\}, \{\texttt{CAT_1}, \texttt{CAT_3}\}, \{\texttt{EXT_3}\})\}.
\end{aligned}$$

The last four of these values are not feasible. This is clearly less precise than the value computed by our analysis, which is given by Equation (6.6).

6.3.4 Fixed Point Iteration

Let us consider method *sendMessage*() from Figure 6.2. In Section 6.3.3, we have assumed that the value of the Intent *src* is available when we generate field transformers for *intent*. In reality, it is not initially available, because when we solve the problem for the first time, values for *intent* and *src* are computed in the same iteration. Thus, in order to fully resolve all values, we run several iterations of the COAL solver. For example, in the first iteration, the transformer that is generated for statement `intent.setAction(src.getAction())` is

$$\{\phi_{intent,1} \times \phi_{intent,2} \times \phi_{intent,3}\} = \left\{\phi_{src,1} \times \phi^{\varnothing}_{\{\texttt{CAT\_2}\}} \times id\right\},$$

where $\phi_{src,1}$ is a transformer that indicates that the value of the first (*action*) field of *intent* refers to first field of the Intent *src*. We initially start with $\phi_{intent,i}$ and $\phi_{src,i}$ mapping to ω, for $1 \leq i \leq 3$. We then iterate until a fixed point is reached for $\phi_{intent,i}$ and $\phi_{src,i}$. The fixed point iteration similarly resolves the value of *intent* in the case where *intent* references the value of the *extras* field of *src* (Line 11 in Figure 6.2).

6.4 Evaluation

In this section we evaluate our analysis tool IC3 (Inter-Component Communication analysis with COAL), which implements the techniques presented in this chapter. The eval-

uation of our approach was aimed at answering three central questions:

Q1: Can IC3 precisely infer specifications for ICC objects?

Q2: Are the computational costs of IC3 feasible in practice?

Q3: As an application of our analysis, which properties of component connectivity can be inferred?

The answer to these questions determines how effectively our analysis can be used as the basis of inter-component analyses. Highlights of our evaluation are:

- IC3 infers precise specifications for 85.53% of ICC values. The next best tool can only infer 63.96%. This is a significant increase in precision.
- On average, our analysis takes less than one minute per application. This makes it feasible in practice to use our analysis as the first step of inter-component analyses.
- When matching components that may communicate with one another, specifications from IC3 lead to 54.79% fewer component links than the current state-of-the-art. Most components are not highly connected to other components, with most potential links concentrated on a small subset of the components in our data set.

The constant propagation solver and the string analysis are implemented on top of the Soot framework [131]. We additionally use the Heros [130] IDE solver. We use Soot for several analyses that are necessary for our tool (e.g., Spark [133] pointer analysis and call-graph construction). Since Android applications have no single entry point (e.g., `main()` method), we leverage the approach presented in [102] that creates an entry point from the components of an Android application. The Spark pointer analysis is also used for aliasing. We handle aliasing as follows. When an ICC method modifies an Intent $i1$ that is a possible alias for another Intent $i2$, we keep track of two values for $i2$. One that takes the call into account and the other one does not. The one that does not account for the case where the alias analysis results in a false positive (i.e., detecting that a value may point to a certain heap location even though it does not). IC3 used Java bytecode that was retargeted from Dalvik bytecode using the Dare tool [10].

For performance reasons, we generally do not allow the constant propagation to analyze the Android framework code. The only exception is when a framework class may create or modify ICC objects, which only occurs in a small fraction of the framework code. In the few cases where ICC method arguments are not strings of characters (e.g.,

	Specification	Precise		Imprecise		Missing	
	count	Epicc	IC3	Epicc	IC3	Epicc	IC3
Intents & Filters	4742	3195 (67.38%)	4072 (85.87%)	1462 (30.83%)	589 (12.42%)	85 (1.79%)	81 (1.71%)
URIs	288	22 (7.59%)	230 (79.86%)	20 (6.94%)	11 (3.82%)	246 (85.42%)	47 (16.32%)
Total	**5030**	**3217 (63.96%)**	**4302 (85.53%)**	**1482 (29.46%)**	**600 (11.93%)**	**331 (6.58%)**	**128 (2.54%)**

Table 6.2. ICC specification precision results.

integer arguments), we use a simple analysis that looks for definitions of constant values for that argument. It simply traverses the interprocedural control flow graph starting at the method call, keeping track of all possible values. When a constant value cannot be found, a special ω value is conservatively returned.

We performed our experiments on a corpus of 350 applications previously used in [11]. These applications were randomly selected from a set of over 200,000 applications downloaded from the Google Play store between September 2012 and January 2013. 6 applications could not be processed for various reasons (e.g., insufficient memory errors), so we report numbers for 344 applications.

Specification precision – We first measured the precision of the ICC values inferred by our tool. We considered the ICC values that were inferred by our tool at program points of interest (i.e., sending a message, or programmatically registering a component with an Intent Filter). We considered an ICC specification to be imprecise if any field value of any of its possible values was completely unknown (e.g., it was a `(.*)` string value). Otherwise, we said that a specification was precise. We counted how often our tool inferred precise specifications and we compared it to Epicc [11]. We modified Epicc such that it used the same entry point construction procedure from [102]. The precision results are presented in Table 6.2. The third line shows the results for Intents and Intent Filters, whereas the fourth line shows statistics for URIs. The *specification count* column shows the total number of ICC objects that were detected. The *precise* columns present the number of precise ICC values discovered by Epicc and by IC3. The *imprecise* columns show the number of imprecise values detected by each tool. Finally, the *missing* columns show the number of locations where an ICC value was missed by both tools.

We observe that the precision of the values inferred by IC3 for Intents, Intent Filters and URIs was high, with 85.53% of values being detected accurately by our tool. Epicc, on the other hand, could only precisely detect 63.96%. Of the 877 Intent and Filter values that IC3 detected precisely but Epicc did not, 476 were due to the presence of URI data in Intent values (which is not handled by Epicc). In 4 cases, Epicc missed a value that IC3 did not. The remaining 397 cases that were precisely detected by IC3 and not by Epicc were due to the more powerful string analysis. There was also a clear difference in the case of URIs, with our tool precisely determining 230 values, compared to 22 for Epicc. That is because Epicc does not include a thorough model for URIs. In particular, a number of methods refer to other modeled objects. Since this is handled in an *ad hoc* manner in Epicc, good coverage of these methods cannot be achieved, resulting in a lot of missed values. On the other hand, using COAL specifications, IC3 can achieve

much better coverage of URI methods. In particular, references to modeled values are handled in a principled and generic manner. Finally, IC3 detected 9 fewer URI values imprecisely than Epicc, thanks to our new string analysis.

There are several reasons why IC3 missed 128 ICC values. First, some API callback methods have Intent or URI arguments that cannot be known statically. For example, method *onReceive*() is a Broadcast Receiver callback that is called when the Receiver receives an Intent. The received Intent is passed as an argument to that method by the framework upon activation of the Receiver. The value of that Intent is in general impossible to determine statically. We found 36 such cases. Another related case was when URIs were extracted from Intents that were callback arguments with the *getData*() method, before being used to address Content Providers. Another cause for missed ICC values was when Intents were extracted from containers such as sets or lists. We will investigate handling these by tracking the values of these containers. Finally, we found a few pathological cases where a call to an interface or abstract method returning an Intent was not resolved to the proper possible subtypes by the call graph construction procedure.

In the 600 cases where imprecise values were inferred, the arguments to ICC API methods could not be determined. Some cases are not yet handled by our argument analyses (e.g., integer fields and string array fields), while other cases cannot be determined statically (e.g., sequences of complex string operations). We will continue investigating the cases that can be resolved while keeping good performance.

Performance – Processing all the applications took 20538 seconds using our tool, or slightly less than 6 hours of compute time. That is less than 60 seconds per application on average. The processing time was dominated by the entry point building procedure of [102], taking 53.27% of the time overall. The second most time-consuming functions were the IDE problem solver and the string solver, taking 33.45% of the total time. Soot analyses (class loading, type inference, final call graph construction, etc.) took 7.46% of the time. Other parts of the analysis (e.g., COAL model parsing, result generation) took 5.38% of the total time.

Component matching – As an application of inferring ICC specifications, we matched the components that send the computed Intents and URIs with receiving components for the 344 applications. We implemented a matching process that was modeled after the Android Intent and URI resolution process. We performed the matching using both the specifications computed by IC3 and those calculated by Epicc.

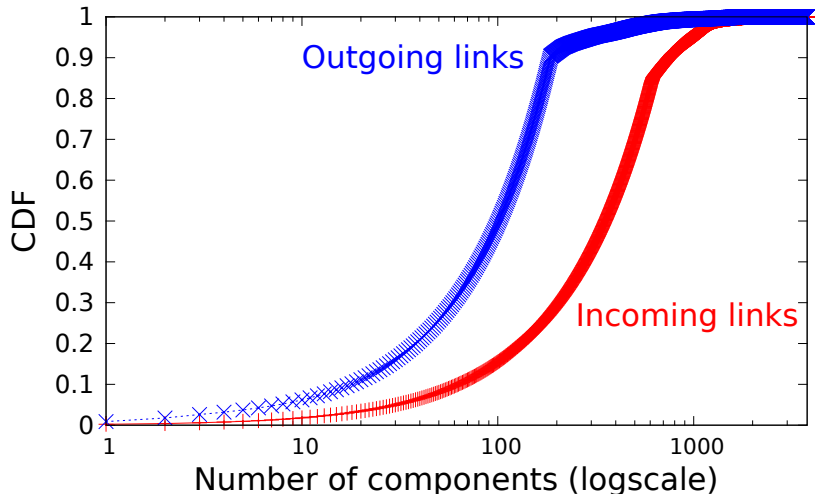


Figure 6.7. CDF of incoming and outgoing ICC links.

Matching components with one another using specifications output by IC3 produced 127204 links. In contrast, the matching that used Epicc specifications yielded 281361 links. Component matching can be used as a metric for precision. When performing inter-component analysis, fewer potential links imply fewer false positives (since the ICC value computation and matching are conservative). Using that metric, IC3 is over twice as precise as Epicc. The reason why a 21.5% gain in value precision resulted in a 50% gain in matching precision is that imprecise ICC values often cause an explosion of the number of potential links. For example, when the *action* of an Intent is not known, the matching process conservatively matches it with all Intent Filter *action* values.

We performed a preliminary study of component connectivity using the IC3 ICC values. We found that many components did not receive ICC messages. Out of 3794 components in our sample, 1087 components were completely isolated. Only 1997 (52.64%) had a potential incoming link with other components. The average in-degree of all components that were not isolated was 63.70. The average out-degree was 68.21 and most components did not send ICC messages. Only 1732 (46.65%) potentially sent data to other components. Figure 6.7 shows the CDF of the number of potential incoming and outgoing ICC links for the components in our sample. We found that ICC was concentrated in a few components, with 102 components accounting for half of the outgoing links. Additionally, only 354 components accounted for 50% of all the potential incoming links. Future studies will seek to determine structures of interest in the graph of ICC (e.g., strongly connected components). In particular, we will determine whether such structures are related to security issues (e.g., components leaking data).

Chapter 7

Visualizing Inter-Component Communication

Chapters 5 and 6 have presented techniques to infer properties of inter-component messages. This has opened the door to a wide range of inter-component analyses, including inter-component taint tracking [135], which extends intra-component information flow techniques [102]. However, no work has tried to visualize inter-component link to infer properties of ICC. We believe that the use of such techniques can help in finding potential security concerns. In the past, techniques for visualizing routing data have been used to find security issues [136]. In our context, it may be possible to look for correlations between high fan-in and privilege escalation vulnerabilities. On the other hand, a high fan-out might be indicative of a high chance of private data leaks.

In this chapter, we introduce visualization techniques for Android ICC. Links between application modules and between classes have long been used as metrics of software engineering, with fan-in and fan-out being of particular interest [137]. Fan-in and fan-out express the number of flows into or out of a module. For example, the probability of defects has been shown to be correlated to fan-out [137, 138]. In the same line of research, we are interested in finding components with potential security concerns. We introduce a formalism for statically inferring links between components using set constraints. We also present the first visualizations of ICC using a corpus of 350 applications from the official Play store. We identify challenges related both to the precision of the analysis that generates ICC values and to the structure of ICC. We propose some solutions to these challenges and offer insight on ICC structures that we observe. We expect that these techniques can be used by market providers or developers to verify that applications

are not communicating in unexpected ways. Finally, a user may want to verify how all the applications on her phone are communicating with each other in order to exclude unexpected communication patterns.

We provide the following contributions:

- We formalize the ICC component linking process using set constraints. We provide an efficient algorithm that, given a set of ICC messages and potential receivers, efficiently computes all possible ICC links.
- We visualize ICC links between the components of 350 applications, including inter-application links. We identify the challenges involved in the visualization process and propose a link confidence metric to exclude imprecise links.
- We propose a technique to merge components that have semantically equivalent interfaces to reduce clutter in the representation.

This chapter is organized as follows. In Section 7.1, we introduce a formulation of the problem of matching Intents with Intent Filters using set constraints. In Section 7.2, we describe an efficient algorithm for verifying set constraints for our problem. Finally, in Section 7.3, we offer solutions for the actual visualization of ICC.

7.1 A Set-Constraint Approach to Intent Resolution

Let us assume that we are given a set of Intents I and a set of Intent Filters F. We would like to find all tuples $(i, f) \in I \times F$ such that there is a potential ICC link between i and f. We start by formalizing the Intent and URI resolution process as set constraints. In other words, what are the conditions under which there exists an ICC link between Intent i and Filter f. For ease of exposition, we only show the Intent resolution process. The process of resolving URIs to Content Providers uses the same ideas.

Figure 7.1 shows a running example for this chapter. Figure 7.1(a) is an example of Intents. The first method creates an explicit Intent targeted at the component named `my.target.component` in application `my.target.app`. The *startActivity*() call causes the Intent to be sent to the recipient. The second method creates an implicit Intent and sets its action to `ACTION_DIAL`. It then sets a phone number as the Intent data. The *startActivity*() method call causes the Intent resolution process to find a dialer component from which the user can call the phone number. Figure 7.1(b) presents an example component declaration for a dialer Activity. Its Intent Filter declares handling the `ACTION_DIAL` and `ACTION_VIEW` actions and data with a `tel` scheme. The default

```
1  public void sendExplicitIntent() {
2    Intent intent = new Intent();
3    intent.setComponentName("my.target.app",
       "my.target.component");
4    startActivity(intent);
5  }
6
7  public void sendImplicitIntent() {
8    Intent intent = new Intent();
9    intent.setAction(Intent.ACTION_DIAL);
10   Uri phoneNumber = Uri.parse("tel:1234567890");
11   intent.setData(phoneNumber);
12   startActivity(intent);
13 }
```

(a) Explicit and implicit Intents.

```
1 <activity android:name="DialerActivity"
    android:exported="true">
2   <intent-filter>
3     <action android:name="android.intent.action.DIAL"/>
4     <action android:name="android.intent.action.VIEW"/>
5     <data android:scheme="tel"/>
6     <category
        android:name="android.intent.category.DEFAULT"/>
7   </intent-filter>
8 </activity>
```

(b) Intent Filter for a dialer component.

Figure 7.1. Running example.

category declaration at Line 5 is required, since this category string is automatically added to all Intents targeted at activities. This Intent Filter enables the component to receive the Intent declared in method *sendImplicitIntent*() of Figure 5.4(a).

Set constraints have been used to express many different program analysis problems [139]. In order to make the resolution process as generic as possible, we assume that each Intent and each Intent Filter contains all attributes necessary for the linking process. For example, in addition to the attributes that are set in the code by an application developer (e.g., action, categories), an Intent is assumed to contain:

- The name of the application containing the Intent
- The permissions that the containing application requests at install time ("uses permissions")

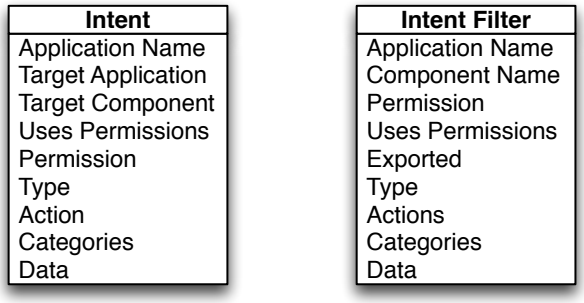


Figure 7.2. Representation of Intents and Intent Filters used for the linking process.

Figure 7.2 shows a description of Intents and Intent Filters used for matching. Note that such a representation is easy to obtain from the application representation shown in Figure 2.1 by simply adding the proper attributes from the Component and Application items. In order to handle explicit Intents in a generic way, we introduce Intent Filters for all components. Every component has at least one Filter with an Application Name and a Component Name attribute. The alternative would consist in using a resolution process between Intents and components for explicit Intents. Instead, we use a single Intent-to-Filter matching process for all Intents.

We represent each attribute of Intents and Intent Filters as a set. For example, an Intent may have no action or a single action. We represent the case with no action with an empty set $\varnothing$. In the case of a single action string a, we use set $\{a\}$. In the case of a boolean attribute such as the Exported flag, we use sets $\{\texttt{true}\}$ or $\{\texttt{false}\}$. This uniform representation allows us to represent the entire resolution process as a system of set constraints.

Traditional constraint systems translate set equality to inequalities using the fact that for sets A and B, $A = B$ is equivalent to $A \subseteq B \wedge B \subseteq A$. We additionally use the fact that $A = B$ is equivalent to $A \subseteq B \wedge |A| = |B|$, since several constraints involve sets that we know to be of the same size. For example, one constraint is that the component type targeted by an Intent $type(i)$ is the same as the type of component associated with a Filter $type(f)$. Since there is only a single type associated with each Intent and each Filter, we use constraint $type(i) \subseteq type(f)$. This allows us to use the generic constraint solving process from Section 7.2 to solve set equality constraints.

In this section, we denote the action of Intent i by $action(i)$. We use similar notations for other Intent and Intent Filter fields. Given an Intent i and an Intent Filter f_i, we say that i matches f if $match(i, f)$ is satisfied, where $match(i, f)$ is true if the following

boolean expression evaluates to true:

$$match(i, f) = type(i, f) \wedge visibility(i, f) \wedge permission(i, f) \\ \wedge (explicit(i, f) \vee implicit(i, f)). \tag{7.1}$$

In this expression, $type(i, f) = type(i) \subseteq type(f)$ as mentioned above. Also, $visibility(i, f)$ evaluates to true if Filter f is visible from Intent i. The $permission(i, f)$ predicate is true if Intent i has permission to access Filter f and f is allowed to receive i. Finally, $explicit(i, f)$ or $implicit(i, f)$ are true if i matches f as an explicit (respectively implicit) Intent.

The visibility criterion states that i can access f if i and f are in the same application or f is exported. This is expressed as:

$$visibility(i, f) = app\_name(i) \subseteq app\_name(f) \vee exported(f) \subseteq \{\texttt{true}\}$$

The permission condition stipulates that if a Filter is protected by a permission, then the application sending an Intent to it needs to request that permission at install time. Also, if the Intent is protected by a permission, then the receiving Filter must belong to an application that declares the permission. This is expressed as:

$$permission(i, f) = perm(i) \subseteq uses\_perm(f) \wedge perm(f) \subseteq uses\_perm(i)$$

An Intent explicitly matches a Filter if $explicit(i, f)$ is verified:

$$\begin{aligned} explicit(i, f) = {} & target\_comp(i) \neq \varnothing \\ & \wedge target\_comp(i) \subseteq comp\_name(f) \\ & \wedge target\_app(i) \subseteq app\_name(f) \end{aligned}$$

On the other hand, the expression is more complex in the case of implicit Intents:

$$\begin{aligned} implicit(i, f) = {} & target\_comp(i) == \varnothing \\ & \wedge action(i) \subseteq actions(f) \\ & \wedge category(i) \subseteq categories(f) \\ & \wedge data(i, f), \end{aligned}$$

where $data(i, f)$ represents the data test for Intent i and Filter f. We do not give its

expression for ease of exposition but it is also expressed using the same kinds of set constraints.

For example, let us assume that the two methods of Figure 7.1(a) are part of an application `my.app` that does not request any permissions. Then the explicit Intent i_e is such that:

$$\begin{aligned} app\_name(i_e) &= \{\texttt{my.app}\} \\ target\_app(i_e) &= \{\texttt{my.target.app}\} \\ target\_comp(i_e) &= \{\texttt{my.target.component}\} \\ uses\_perm(i_e) &= perm(i_e) = \varnothing \\ type(i_e) &= \{\texttt{activity}\} \\ action(i_e) &= categories(i_e) = data(i_e) = \varnothing \end{aligned}$$

The implicit Intent i_i is such that[1]:

$$\begin{aligned} app\_name(i_i) &= \{\texttt{my.app}\} \\ target\_app(i_i) &= target\_comp(i_i) = \varnothing \\ uses\_perm(i_i) &= perm(i_i) = \varnothing \\ type(i_i) &= \{\texttt{activity}\} \\ action(i_i) &= \{\texttt{android.intent.action.DIAL}\} \\ categories(i_i) &= \{\texttt{android.intent.category.DEFAULT}\} \\ data(i_i) &= \{\texttt{tel}\} \end{aligned}$$

For simplification, we have reduced the data field to a URI scheme. As we mentioned above, in reality data is described by several fields.

Let us assume that the Intent Filter f from Figure 7.1(b) is part of an application

[1]The `android.intent.category.DEFAULT` category is added by the operating system to all Intents targeting Activities.

`my.second.app`. It is modeled by:

$$
\begin{aligned}
app\_name(f) &= \{\texttt{my.second.app}\} \\
comp\_name(f) &= \{\texttt{DialerActivity}\} \\
uses\_perm(f) &= perm(f) = \varnothing \\
type(f) &= \{\texttt{activity}\} \\
action(f) &= \{\texttt{android.intent.action.DIAL}, \texttt{android.intent.action.VIEW}\} \\
categories(f) &= \{\texttt{android.intent.category.DEFAULT}\} \\
data(f) &= \{\texttt{tel}\}
\end{aligned}
$$

It is possible to verify that $match(i_i, f)$ holds true using the above description.

7.2 Efficient Solution of Set Constraints with Regular Expressions

We have presented a model to determine if an Intent and a Filter are connected. The remainder of this paper is concerned with finding aggregate communication patterns. In other words, given a set of Intents I and a set of Filters F, we would like to find all tuples $(i, f) \in I \times F$ such that there is a potential ICC link between i and f. Finding all links between Intents in I and Filters in F can be done by simply iterating through all elements of I and F and verifying if the $match(i, f)$ expression is satisfied, for all tuples (i, f) in $I \times F$. This process has worst-case and average time complexity that is $O(c|I||F|)$, where c is the number of clauses that need to be checked. In this section, we present an algorithm that has the same worst-case complexity, but that has much better performance on average. Having an efficient matching algorithm is important as an efficient algorithm is more likely to be utilized by developers and markets to verify that applications cannot communicate with unexpected applications.

An additional challenge that needs to be handled by the component linking process is that some of the strings for Intent or Filter attributes that are matched are expressed as regular expressions, such as `EXAMPLE_.*`. This can happen when the IC^3 tool can only determine part of a string of characters. Note that when IC^3 fails in resolving a string at all, it returns a `.*` regular expression, which can match any string. For a regular expression α, we denote the formal language described by α using $L(\alpha)$. Thus, in order for an Intent attribute $a(i)$ to match a Filter attribute $a(f)$, several cases are possible:

Algorithm 1 Procedure to efficiently find all Filters that include a given set of attributes.

Input:

- *attrs*: set of Intent attributes to match to Filters
- *simple_map*: map between Filter attributes that are simple constants and the Filters that contain them
- *regex_map*: map between Filter attributes that are regular expressions and the Filters that contain them
- *filters*: set of all Filters being considered

```
1: procedure FindFiltersWithAttributes(attrs, simple_map, regex_map, filters)
2:     result := filters
3:     for all attributes attr in attrs do
4:         found := FindFiltersWithAttribute(attr, simple_map, regex_map)
5:         result := result ∩ found
6:     end for
7:     return result
8: end procedure
```

- $a(i)$ and $a(f)$ are simple constants and $a(i) = a(f)$.
- $a(i)$ is a simple constant and $a(f)$ is a regular expression and $a(i) \in L(a(f))$.
- $a(f)$ is a simple constant and $a(i)$ is a regular expression and $a(f) \in L(a(i))$.
- $a(i)$ and $a(f)$ are regular expressions and $L(a(i)) \cap L(a(f)) \neq \varnothing$. This indicates that there is at least one string common to both regular expressions. Thus for a conservative linking process we need to consider it to be a possible match.

Note that we can consider a simple string constant s to be a regular expression and in that case $L(s) = \{s\}$. Therefore, in general, in order for an Intent attribute $a(i)$ to match a Filter attribute $a(f)$, we simply require that $L(a(i)) \cap L(a(f)) \neq \varnothing$. In order to account for the possible presence of regular expressions, we define set inclusion $A \subseteq B$ to mean the following. We say that $A \subseteq B$ if for each element $a \in A$, there is an element $b \in B$ such that $L(a) \cap L(b) \neq \varnothing$.

In order to simplify the complexity analysis, we consider the process of deciding whether $L(a) \cap L(b) \neq \varnothing$ to be bounded by a constant. In general this is not true [140], but in our problem all strings have length bounded by a constant (for database storage reasons).

Algorithm 2 Procedure to find all Filters that have a given attribute.

Input:

- *attr*: Intent attribute to match to Filters
- *simple_map*: map between Filter attributes that are simple constants and the Filters that contain them
- *regex_map*: map between Filter attributes that are regular expressions and the Filters that contain them

```
1: procedure FindFiltersWithAttribute(attr, simple_map, regex_map)
2:     result := empty set
3:     if attr is a simple constant then
4:         Add simple_map[attr] to result
5:     else
6:         for all pairs (attribute, filters) in simple_map do
7:             if L(attr) ∩ L(attribute) ≠ ∅ then
8:                 Add filters to result
9:             end if
10:        end for
11:    end if
12:    for all pairs (attribute, filters) in regex_map do
13:        if L(attr) ∩ L(attribute) ≠ ∅ then
14:            Add filters to result
15:        end if
16:    end for
17:    return result
18: end procedure
```

As we have seen in Section 7.1, in order to find all possible targets for an Intent, we need to be able to find Filters that verify three types of set constraints. Given sets A and B, we need to be able to verify:

(1) $A \subseteq B$

(2) $A = \varnothing$

(3) $A \neq \varnothing$

The general algorithm consists in going through all clauses and finding all the Filters that verify each of them. When a clause is a disjunction, we then take the union of the Filters that verify the parts of the disjunction. When a clause is a conjunction, we take the intersection of the Filters that verify the parts of the conjunction. We start by presenting how to proceed with set inclusion constraints (type (1) above). Algorithm 1 shows the

FindFiltersWithAttributes procedure for finding all Filters with a set of attributes. In other words, it solves the constraints of type (1) in the list above. For each attribute type (e.g., action and category), we maintain a map between each attribute and the set of Filters that declare the attribute. For example, we maintain a map of actions, where keys are action strings and values are the Filters that declare the action key. Since attributes may be regular expressions, FindFiltersWithAttributes takes as input both a map between string constants and Filters and another map between regular expressions and Filters. Finally, it takes as input the set of Filters currently being considered. This procedure simply takes the intersection of all Filters that have each one of the input attributes. It uses procedure FindFiltersWithAttribute from Algorithm 2, which finds all Filters that have a given attribute, taking into account regular expressions.

In procedure FindFiltersWithAttribute, if the input attribute is a string constant, then we start by adding all Filters that declare the attribute to the set of Filters to be returned (Line 4). If the input attribute is not a constant, then we need to match it with all constant Filter attributes by iterating through the map of constant Filter attributes (Lines 6-10). For example, if the input attribute is `.*`, then all Filters with a constant attribute will match. Then, we proceed to match the constant with all Filters that declare regular expressions. For example, if a Filter is found to declare `.*` by IC^3, then all input attributes will match it.

The constraints of types (2) and (3) are solved by maintaining sets of Filters that have no attribute of a given type (constraint (2)) or that have at least one attribute of the given type (constraint (3)). The only constraint that requires different treatment is the permission condition. We have:

$$permission(i, f) = perm(i) \subseteq uses\_perm(f) \wedge perm(f) \subseteq uses\_perm(i).$$

The first clause is solved easily by using the method above. The second clause needs special treatment since the Intent attribute set is on the right-hand side of the set inequality and the method above is designed to solve constraints with the Filter attribute on the right-hand side. We verify the second clause after all other clauses have been verified by iterating through all candidate Filters as they are added to the final result set. Note that this does not impact performance, since the final set of candidate Filters is typically small. Further, this final iteration happens in any case due to the need to store the resulting links between the Intent being considered and the possible target Filters.

The reason why this optimized algorithm performs better than systematically iterating though all Filters is that most Filter attributes are precisely known. That is be-

Component Type	Component Count	Exported (% unprotected)	Permission
Activity	2823	620 (100%)	0
Broadcast Receiver	671	594 (91%)	52
Service	185	40 (58%)	17
Content Provider	19	19 (95%)	1
Total	**3698**	**1254 (94%)**	**69**

Table 7.1. Component statistics.

cause almost all Filters are known through a straightforward parsing of the application manifest file. Thus, finding all Filters with a given attribute in procedure FindFiltersWithAttribute is a fast process. The Filter attribute maps can be implemented with a hash table, allowing constant time access to all Filters with a given attribute at Line 4. Since most Intent attributes are also simple constants, the iteration at Lines 6-10 is not often performed. We will show performance comparisons between the naïve algorithm and the algorithm presented above in Section 7.3.

7.3 Approaches to Visualize ICC Links

We perform ICC linking experiments using a corpus of 350 applications already studied in [11]. They were selected at random among a set of 200000 applications downloaded from the official Play store between September 2012 and January 2013. We retarget them to Java bytecode using Dare [10] before running IC^3 [134] to infer Intent and Intent Filter values. 8 applications could not be processed (e.g., due to insufficient memory errors), so we report experiments for 342 applications.

Computing all links took 29 seconds using the algorithm described in Section 7.2. On the other hand, the naïve approach that consists in iterating over all Intents and Intent Filters took 696 seconds. Looking more closely at our data set, we notice that as we expected in most cases Intent and Filter attributes are simple constants, which means that we are able to verify individual constraints in constant time. Taking the intersection of the sets that verify different constraints (to account for the conjunction of several set constraints) takes time that is bounded by the size of the smallest set. Therefore, since several constraints are usually very restrictive (e.g., the data constraint) taking the intersection of sets is also a fast operation.

Table 7.1 shows a breakdown of the components by type. The *exported* column shows how many components are accessible to all applications. It also indicates the percentage of exported components that are not protected by permissions. The *permission* column gives the number of components protected by a permission. As previously reported [11], a large proportion of public components are not protected by a permission. Thus, a method to infer how components may communicate with one another is very important. The data set comprises 342 applications, with a total of 3698 components. Activities are the most common type of component, with 2823 instances found in our corpus.

7.3.1 Intent and Filter Links

We start by introducing a confidence metric for Intent links. The idea is that some Intents have very imprecise attributes, in the form of `.*` regular expressions. These match Filter attributes with any value, leading to many unfeasible links (i.e., false positives). Given a link between an Intent and an Intent Filter, let us denote the number of attributes that have value `.*` by wc. The total number of non-empty attributes is denoted by na. Then we define the confidence of the link to be

$$confidence = \begin{cases} 1 - \dfrac{wc}{na} & \text{if } na \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

Thus, a confidence of 1 implies a link where all attributes that account for the link are precise, resulting in a precise link. On the other hand, a confidence of 0 means a link where no attribute is known precisely. This implies a very likely false positive. Note that we do not count attributes that are always known precisely, such as the *exported* and *type* attributes.

Figure 7.3 shows links between Intents and Intent Filters. Intents and Filters that are not connected are not represented. This graph includes 2409 nodes and 8354 edges. Blue nodes represent Intent and red nodes are Intent Filters. Intents with the same attributes are connected to the same destinations, thus we merge similar Intents to a single node. On the graph in Figure 7.3, the size of Intent nodes is proportional to the number of Intents they represent. We do the same for Intent Filters. Note that this representation is more expressive and less cluttered than plotting all Intents and Intent Filters separately. Plotting all Intents separately would result in many more nodes. On the other hand, in our representation we have fewer nodes and the redundancy information is explicitly represented. Blue edges represent links with high confidence, whereas purple edges are

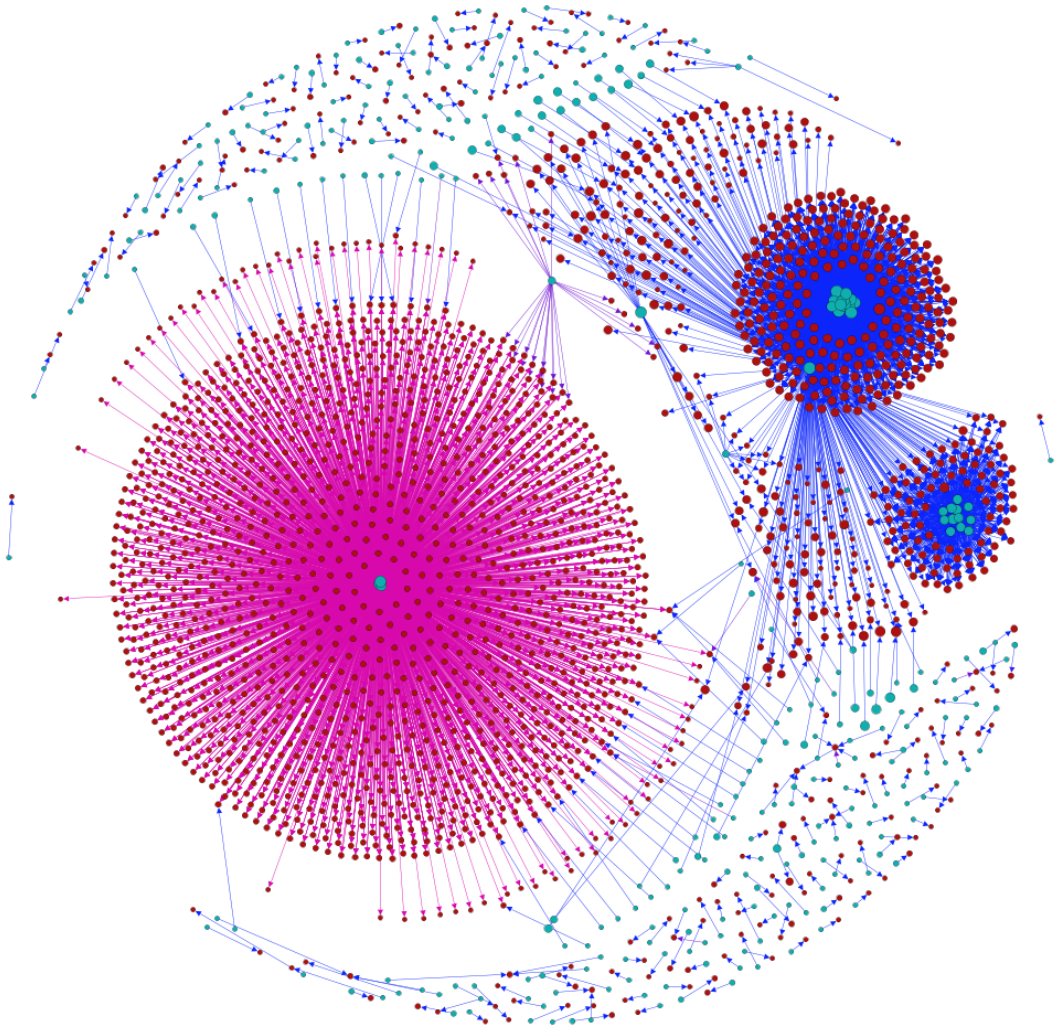


Figure 7.3. Links between Intents and Filters.

low-confidence links. Notice the large cluster with some Intents at the center connected to a large number of Intent Filters through purple, low-confidence edges. This confirms the intuition that Intents and Filters with imprecise `.*` attributes entail many false positives.

Figure 7.4 shows the links between Intents and Intent Filters after excluding links with a confidence of 0. Green edges represent explicit Intent links whereas red edges represent implicit Intent links. After excluding low-confidence links, only 4520 edges remain. Interestingly, implicit links form large clusters whereas explicit Intents are found in small graph components. That is because implicit Intents do not have a set target and

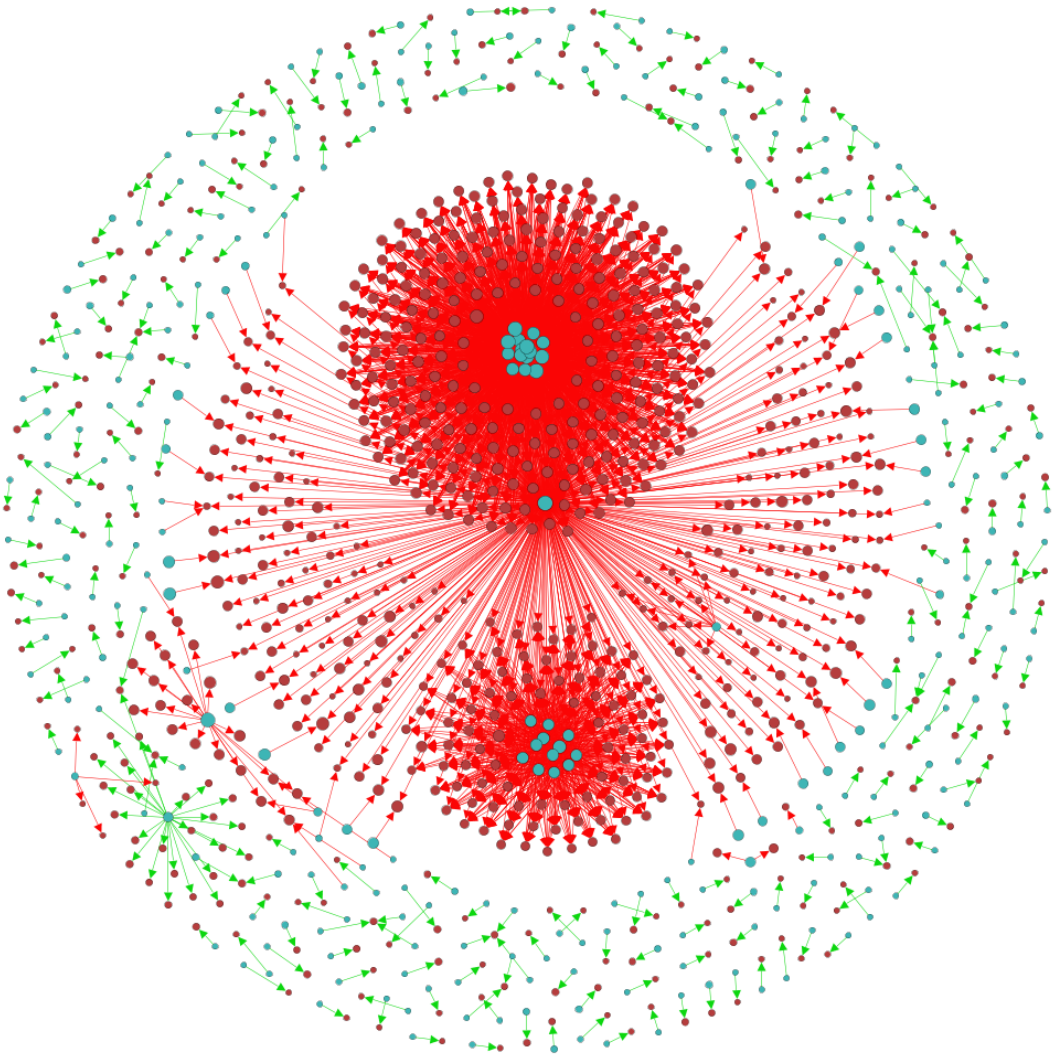


Figure 7.4. Links between Intents and Intent Filters without low-confidence links.

many Filters can potentially receive them. Looking more closely at the largest clusters, we observe that they are caused by a large number of Intents and Intent Filters that have similar attributes. These attributes are not all exactly the same, so the Intents and Filters do not get merged to a single node. The Intent Filters in these clusters have broad specifications, which makes them a target for many Intents. On the other hand, explicit Intents have a single target. In Figure 7.4, a few explicit Intents are connected to several Filters. That is because one of their attributes is not known precisely.

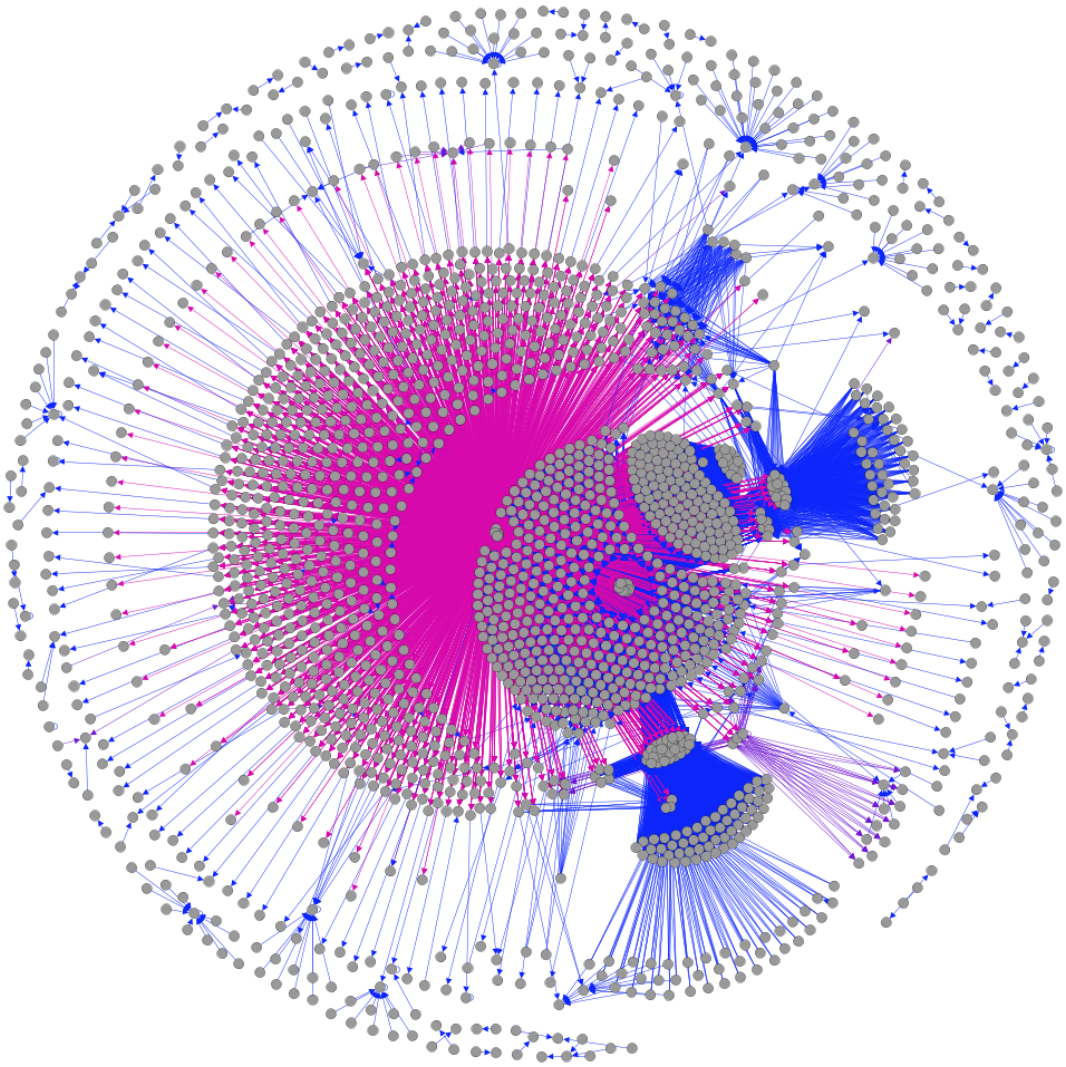


Figure 7.5. Links between components.

7.3.2 Component Links

Using the link between Intents, we can infer potential communication between components. Since a single component may send more than a single Intent, it is possible that several Intents account for the link between two components. On the other hand, a single Intent can originate from different components. That is because different components can call methods in the same class (e.g., utility functions). When two components are connected by n Intent links with confidence $conf_1, conf_2, \ldots, conf_n$, we define the confidence of the inter-component link to be $max\{conf_1, conf_2, \ldots, conf_n\}$. The idea is that if there exists a high-confidence Intent link between two components, then with high

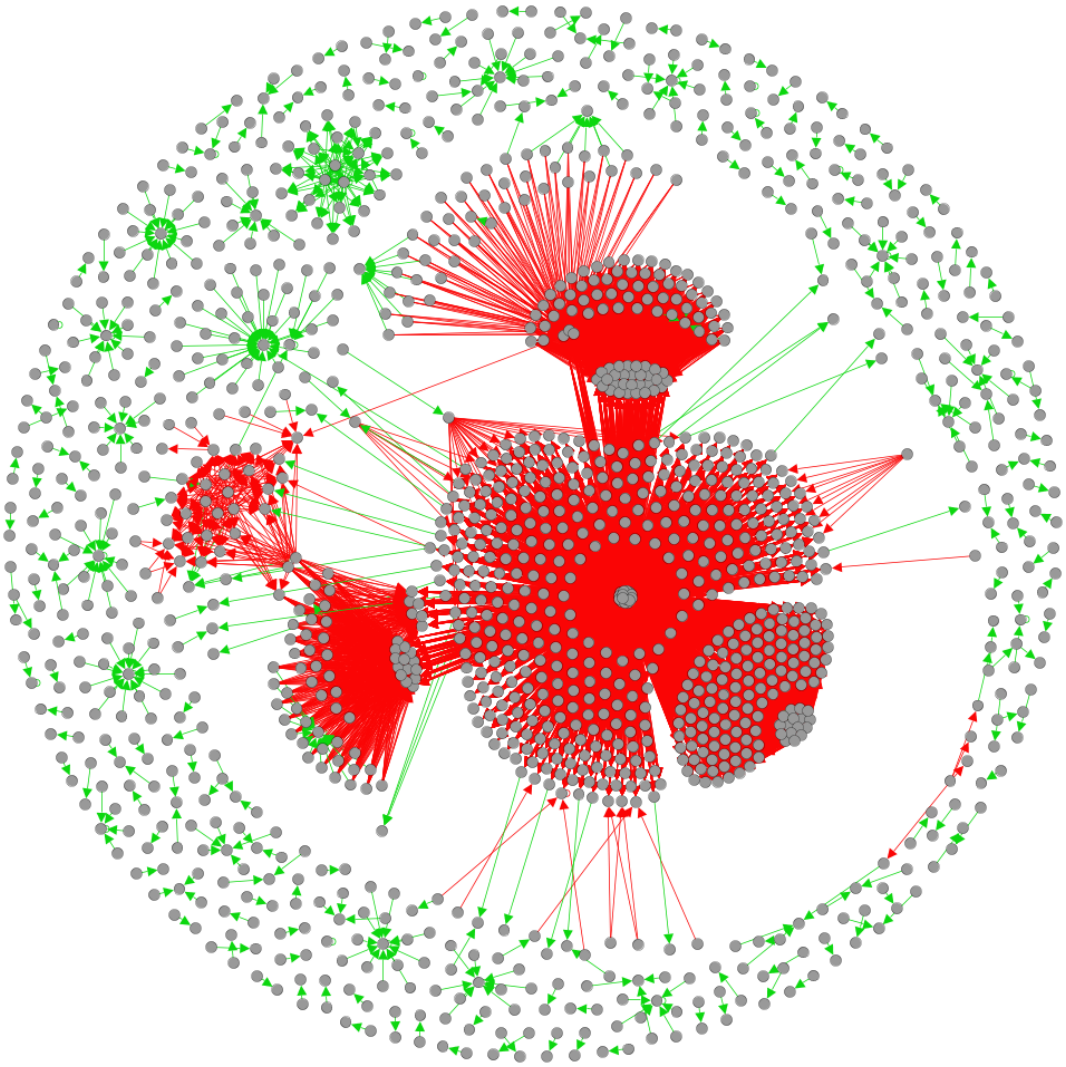


Figure 7.6. Links between components without low-confidence links.

likelihood there is an inter-component link. Figure 7.5 shows links between components, where blue edges represent high confidence whereas purple edges imply low confidence. Components without ICC links are not represented. There are 1920 components connected through ICC, with 22470 potential inter-component links.

As expected from Figure 7.3, there is a high number of low-confidence links. That is why we also present the inter-component links without low-confidence links in Figure 7.6. In this graph, there are only 1387 nodes connected through ICC and 14857 inter-component edges. On this graph, red edges represent explicit Intent links whereas green edges are implicit Intent links. Similarly to what we observe for Intent links, im-

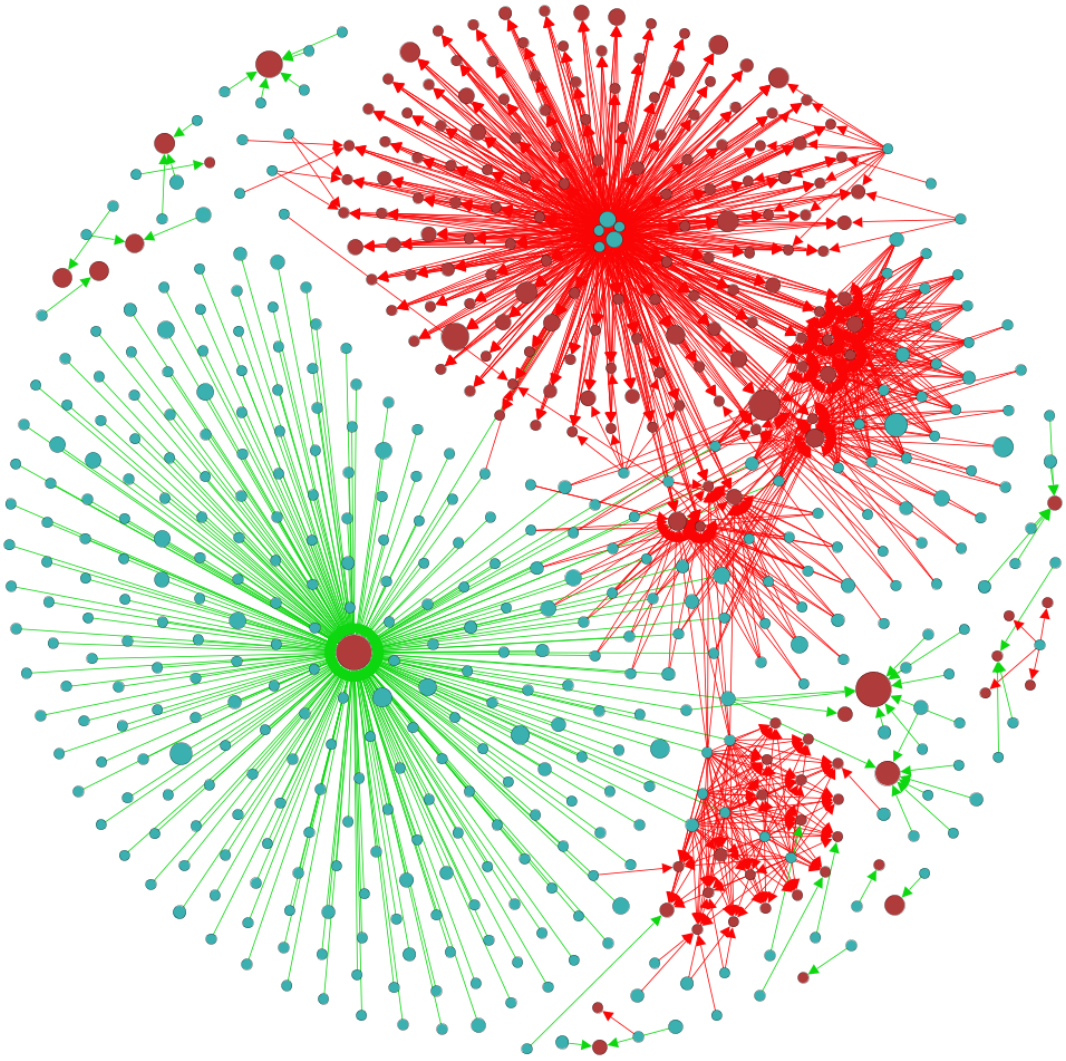


Figure 7.7. Condensed links between components without low-confidence links.

plicit links cause the formation of large clusters, whereas explicit links are observed in small graph components of a few nodes.

Many Intents and Intent Filters have similar values. As a result, we introduce a new representation for component links that is similar to Figure 7.4. The resulting graph is presented in Figure 7.7. In this graph, blue nodes represent source components and red nodes are destination components. This implies that a given component may be represented twice. Additionally, many components send or receive Intents with the same attributes. Therefore, when components send Intents with exactly the same attributes, we merge them to a single blue node. When components have the same Intent Filters,

we also merge them to a single red node. The intuition is that we merge components with semantically equivalent ICC interfaces. The code that constitutes the components themselves may be different, we are only merging based on ICC interface semantics. The size of each node is proportional to the number of components it represents. This representation is more informative than the one without merging, since it explicitly informs us about the presence of redundant information. The graph on Figure 7.7 does not include low-confidence links. It only comprises 575 nodes and 1607 edges. Thus this representation is very effective for condensing ICC information.

Green edges are explicit links and red edges are implicit links. We can observe a cluster where many explicit inter-component links point to a single node. That single node represents all the Activity components that are not exported and not protected by a permission. They also do not have any Intent Filter. The components that point to this single node are all the components that send an explicit Intent to transition the user interface from one screen to another one. This cluster confirms the intuition that this is a very common process in Android applications.

7.3.3 Application Links

We also visualize inter-application links in Figure 7.8 without low-confidence links. Red edges represent implicit links whereas green edges are explicit links. As with Intent and component links, a large cluster is caused by implicit links. The other nodes that are not connected all have self-loops, which represent intra-application inter-component communication.

Looking more closely at the cluster of implicit links, we notice that many links originate from a single application. More specifically, they originate from an Intent for which all attributes are empty. In order to show a clearer picture of inter-application communication, we show another graph that does not take into account empty Intents in Figure 7.9. Most of the edges are self-loops with explicit links, which shows that ICC is most often intra-application. This is expected, since as we showed in Figure 7.7 developers very often use explicit Intents to start an Activity within the same application to transition to a new screen.

Since Figure 7.9 includes some potential inter-application links, we have manually determined their cause. The single inter-application link that is due to an explicit Intent (represented by a green edge between two nodes) is caused by an Intent that is not precise. However, since its confidence is not 0, it is not excluded from the figure. The inter-application links that are related to implicit Intents (represented by red edges

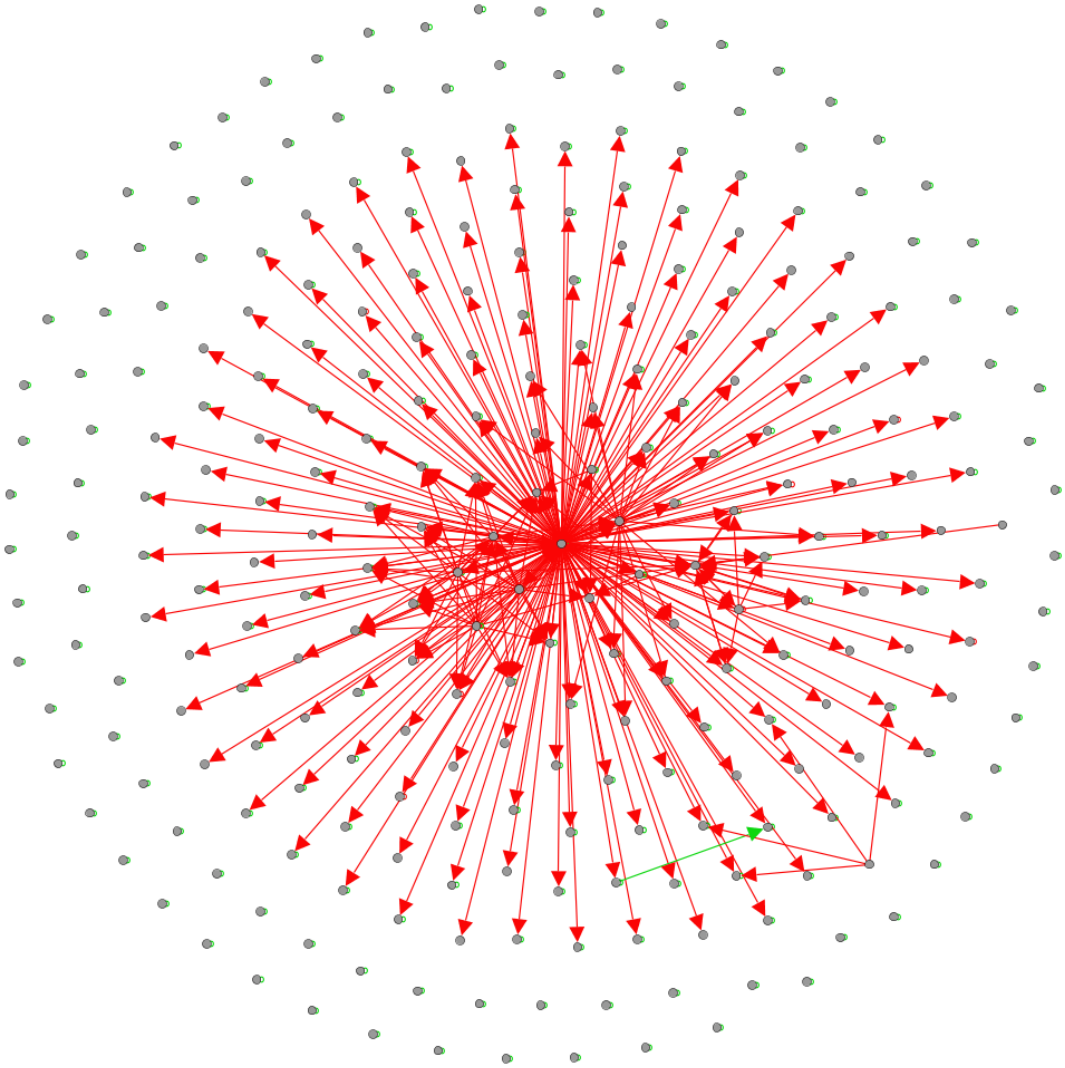


Figure 7.8. Links between applications without low-confidence links.

between nodes) have several causes. In one case, an application is sending an Intent with action `android.appwidget.action.APPWIDGET_UPDATE`, which is usually done to force an update of an application widget. However, in that particular application, the developer has omitted to restrict the Intent to their own application, which means that this Intent causes widgets from other applications to be updated. Two clusters of nodes are related to legitimate, intended inter-application communication. In this case, developers take advantage of some functionality that is provided by another application.

Two other clusters are due to string analysis imprecision for broadcast Intent attributes. We have noticed a common pattern in some advertisement libraries, where

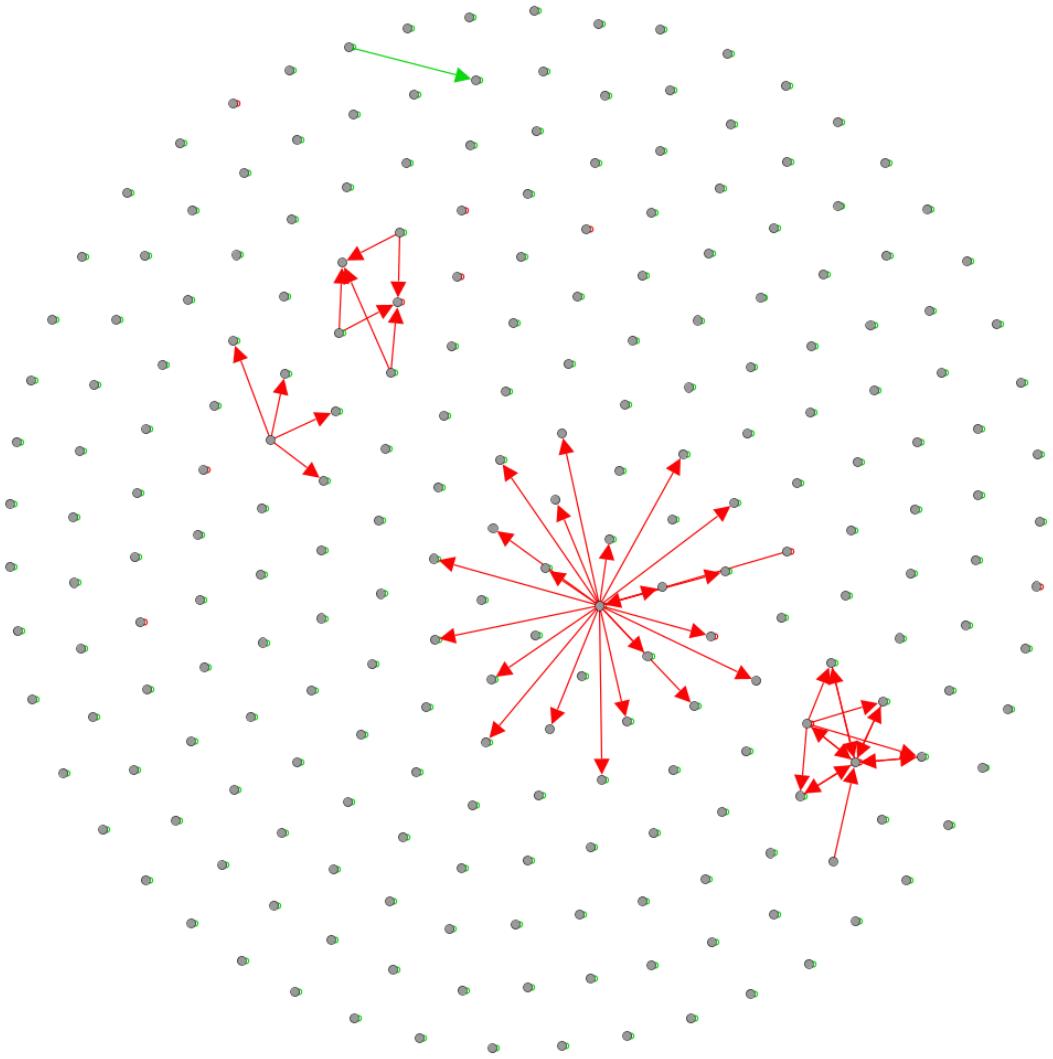


Figure 7.9. Links between applications without low-confidence links and empty Intents.

Intent broadcasts are targeted at internal dynamically-registered Broadcast Receivers. In order to ensure that the broadcast Intent is not caught by other applications, the Intent action strings are set to be a constant string concatenated with an application identifier. In some instances, the string analysis in IC^3 cannot determine the part of the action string that corresponds to the application identifier. In these cases, some Intents may be matched with dynamically registered Intent Filters in other applications that use the same advertisement libraries, even though these flows are not feasible in practice (assuming that the application identifier is not reused by another application). Note that malicious applications may intercept these messages, since the application identifier is

fixed for a given application. The Android platform provides a local Broadcast manager, which guarantees that Intent broadcasts are only sent within the same application. It also ensures that dynamically-registered Receivers only receive Intents from the same application.

Chapter 8

Directions for Inter-Component Communication Analysis

The growth of mobile application markets has accompanied the explosion of smart phones and tablets. Millions of users now rely on these applications to assist them in their daily life. The mobile application distribution model is very different from the desktop application model. In desktop applications, software was distributed by few companies and it was relatively easier for users to choose a trusted developer. On the other hand, in the mobile ecosystem applications are developed by many entities, from large software corporations to amateur developers to malicious programmers. The identity of the application developer is no longer an important factor when deciding which application to install.

The mobile ecosystem relies on communication between application components and between applications to share and reuse functionality. This participates in the explosive growth of application markets. Developers can focus on the features that provide added value to their applications instead of concerning themselves with providing the full chain of features that are needed to the user. Through the use of inter-component messages, they can rely on the operating system to find the missing links that provide the required functionality. Unfortunately, there are many ways that the missing links can be filled with malicious or vulnerable applications.

In this dissertation, we have provided effective means to infer the possible inter-component interactions in Android applications. In this chapter, we highlight some important aspects of this work and the implications of this work for the future of inter-component analysis.

8.1 Program Retargeting

Retargeting Android applications to Java bytecode is a prerequisite to our ICC analysis. Thus, it is important to reflect on why our program retargeting process is very reliable. This is particularly important, as the platform is evolving and the Dalvik runtime may eventually disappear [141]. The new Android Runtime (ART) works by translating Dalvik bytecode to machine code during a compilation process that occurs once on the phone or tablet. It is reasonable to expect that, after the Dalvik virtual machine disappears, applications will be distributed in a bytecode format that is optimized for translation to ART. This would be another intermediate representation between Java bytecode and machine code. If such a format is indeed used instead of Dalvik bytecode, we will need to make the retargeting process evolve to handle the new format.

As evidenced by the Dare system described in Chapter 4, program retargeting is possible when the target representation shares some fundamental characteristics with the original program. First, the type system is similar in Dalvik and Java bytecode. While some constants in Dalvik are more weakly typed, the fact that the Dalvik virtual machine enforces strong typing is key to our ability to map Dalvik typing to Java. For example, after an ambiguously-typed variable is used as an integer, it cannot be used as a floating-point variable. As a result, when we build a type constraint system for a given method, we obtain a system that can be solved, where constraints do not contradict one another.

It is also important that the target language share instruction semantics with the original one. This enables us to find mappings for instructions from Dalvik to Java. A few Dalvik instructions do not have trivial equivalents in Java, but Java bytecode can express the same semantics using a combination of instructions.

Finally, the way that we address the issue of unverifiable Dalvik bytecode is strong evidence of the similarities between the two platforms. We modify unverifiable Dalvik applications to make them Dalvik-verifiable and obtain Java-verifiable applications in almost all cases. The few cases where this is not true are related to very isolated corner cases where the Dalvik and Java verifiers differ. This indicates that the two platforms enforce very similar constraints on their respective bytecode formats.

8.2 Inter-Component Communication Analysis

We have developed the IC3 tool to determine the value of ICC objects. It relies on a generic IDE model that is instantiated as necessary using a textual description of the ICC

API. The simple fact that we generalized the *ad hoc* Epicc process described in Chapter 5 to a constant propagation problem in Chapter 6 informs us as to the reason why ICC analysis can be carried out in practice. In many cases, ICC objects are constants that can be computed statically. Admittedly, they are complex constants that require us to introduce the multi-valued, multi-field constant propagation problem.

For example, a common ICC use case is when developers transition the user interface between screens. In this case, a simple explicit intent is sent to the Activity that represents the new screen. In this case and many others, the target of the intent is known at compile time. In fact, one could imagine declaring most intents in an XML resource file in the same way that intent filters are described. The precompiled intent would then be loaded when needed. Dynamic intent creation is not needed in most cases, so this would be very commonly used to simplify both the development process and future ICC analyses.

8.3 Future Work

This dissertation has discussed the static ICC analysis of mobile applications. The purpose of these analyses is to improve the security of inter-component interactions. Since the approaches presented have several limitations, in this section we provide several directions to improve these static analyses. Since not all properties of applications can be decided statically, we also propose a hybrid approach to enforce security policies using both static analysis and runtime enforcement. This can achieve the end goal of improving inter-component security despite the theoretical and practical limitations of static analysis.

8.3.1 Improving ICC Analysis Precision

The static analysis of ICC described in Chapter 6 cannot infer precise ICC values in 14% of cases. We have found that the precision limitations are both due to cases that cannot be known statically, but also to constructs that are not currently handled by the string analysis or the main IDE analysis. These constructs can be handled by extending the current analyses. For example, this includes flows through Java collections such as lists or sets. Unfortunately, modeling these additional flows can be costly in terms of performance.

In particular, the string analysis used in IC3 works by generating a flow graph of string operations. The value of a string can be obtained by simply traversing the graph

```
1  public class MyStringClass {
2    private String[] mySuffixes;
3
4    public MyStringClass(String[] suffixes) {
5      mySuffixes = suffixes;
6    }
7
8    public String createAString(int index) {
9      return "MY_STRING_" + mySuffixes[index];
10   }
11 }
```

Figure 8.1. An example with string array field operations.

and evaluating the string operations that generate the final string. Generating the graph can be costly in time and in memory. Therefore, we propose to modify the analysis to allow for variable precision. For example, it might make sense in some case to precisely take flows through containers into account. In other cases (e.g., when many string array operations are used in the program), we may want the analysis to infer less precise but also less costly values. A challenge here is how to determine the appropriate amount of precision for each case. We expect that heuristics can be useful but we plan on considering other options as well. Possible heuristics include considering the number of flows to a list that need to be modeled or calculating the diameter of the flow graph.

For example, let us consider class MyStringClass from Figure 8.1. It contains a field that is an array of strings, which is set by the constructor. Method *createAString* creates and returns a string that is the concatenation of a constant string and one of the elements of the string array field. If we would like to determine the value that is returned by this method, a first solution is to do a coarse over-approximation of the second part of the string and to infer value `MY_STRING_.*`. If more precision is needed and deemed reasonable, then we can determine the possible values that the $mySuffixes$ field can contain. To be more precise (for example, if the application being analyzed is small), the analysis could also model which string is associated with a given array index.

8.3.2 Improving ICC Visualization

We propose several ideas to improve the visualization presented in Chapter 7. First, we suggest to include system applications when computing component links. This is because system applications are often solicited by third-party applications through ICC. This includes messages sent to browse a web page or to display a map centered at

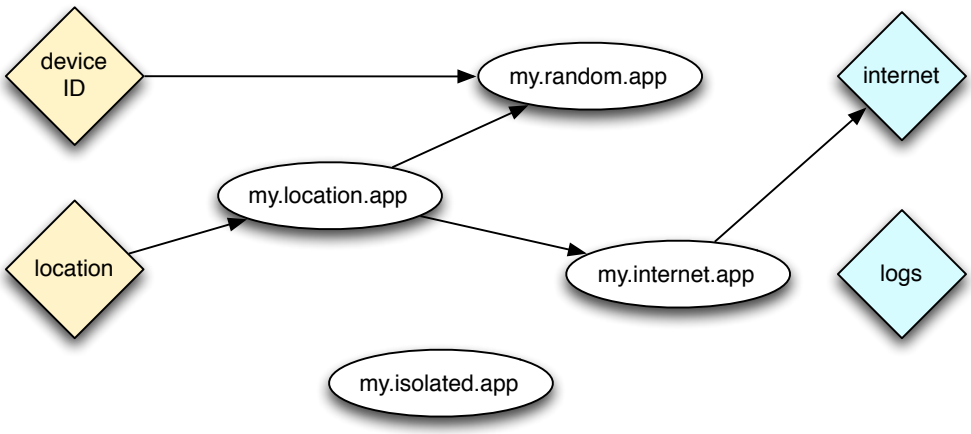


Figure 8.2. Visualization of application collusion.

particular coordinates. Further, system applications have been shown to be vulnerable to ICC attacks. This has been observed both in applications included in the default Android distribution [5] and in applications added by smart phone manufacturers [79]. We expect that this will provide us with a more comprehensive view of inter-component communication in the application ecosystem of a phone or a market.

Second, we would like to include information about data flows in the visualization of inter-component links. Some tools can already perform inter-component taint tracking using the ICC analyses described in this paper [135]. We propose to allow the filtering of edges to include only inter-component flows of sensitive data. Nodes representing sensitive data sources and sinks could be added to the representation. This could quickly inform users and developers that a potentially dangerous flow has been found. In particular, cases of application collusion could be found using this representation. Figure 8.2 illustrates this idea. On the left side of the graph we represent sensitive data sources. On the right side we have sensitive data sinks. The other nodes are applications and the edges show data-carrying inter-application flows. Data from the *location* source flows to `my.location.app`, which then sends the data to the `my.internet.app` and `my.random.app` application. The `my.internet.app` application subsequently leaks the location data to the *internet* sink. This is a case of application collusion. The `my.random.app` application receives sensitive data flows but does not leak the data to sensitive sinks, thus it is not a colluding application.

Finally, beyond modifications of the information that we represent, we would also like to improve the visualization itself. This includes both exploring alternative visualizations and enabling interactions with the current one. A common way to visualize software is by

using the software city metaphor, where a district represents a package and each building represents a class [142, 143, 144, 145]. The size and color of the buildings can be used to represent information (e.g., code size). We could imagine representing each component as a building and each application as a district. Using approaches such as [145], it is then possible to connect buildings using edges.

Another important aspect that should be improved to facilitate using visualization for security is the possible interactions with the representation. In particular, filtering links based on action, URI or any other attribute would allow developers to spot potential issues when using specific attributes. Also, filtering edges based on the presence of sensitive data could be to spot data leaks. Focusing the visualization on links to or from a single application is another desirable feature of a useful visualization tool.

8.4 Hybrid Enforcement of ICC Policies

We propose a hybrid process for enforcement of ICC policies. We suggest to allow the user to define policies related to inter-application communication. For example, a user may wish for an application with access to location data not to communicate with applications with Internet access. We propose the design of a system that will mediate application interaction. Several approaches perform similar or related functionality:

- *Installation policy enforcement:* The Kirin system [45] extends the installation process to disallow installation of applications with dangerous permission combinations, as expressed by a policy. The Kirin system is limited to combinations of permissions in single applications and does not provide guarantees for permission combinations resulting from sets of applications communicating with one another.

- *Application-centric policy enforcement:* The Saint system [60] provides applications with a fine-grained way to define policies for their ICC interfaces. This allows applications to protect themselves, but it relies on application developers to specify how their applications interact with other applications. This does not provide any guarantees to the user. On the other hand, we seek to allow the user to specify ICC policies.

- *Runtime enforcement:* The XManDroid system [82] provides guarantees using runtime enforcement, however the authors of that system observe that the performance overhead is noticeable. We seek to alleviate the performance limitation by limiting the number of runtime enforcement decisions. That is achieved by statically

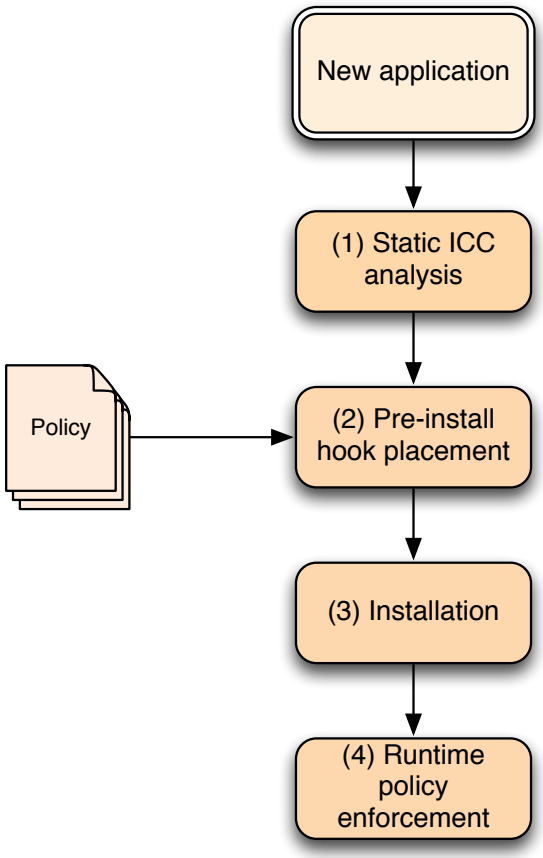


Figure 8.3. Hybrid Inter-Component Communication policy enforcement.

computing the information that can be known before installation.

In order to enforce such a policy, we propose a hybrid approach. The goal is to place enforcement hooks at entry points and exit points of applications. We assume that we have a set of applications that have a safe hook placement and are installed on a single phone. We assume that we also know the ICC interfaces for all these applications. Given a new application that gets installed on the phone, we are trying to determine a set of hooks that will allow enforcement of the policy at runtime. In order to limit the impact on performance, we want to limit the number of hooks that are inserted. For that purpose, we propose to use static analysis to determine the entry points and exit points that are known to be safe. Since static analysis yields an over-approximation of the program behavior, if no unsafe link is detected, then we can say that no unsafe link exists.

Figure 8.3 shows an overview of the proposed approach. In step (1), we compute the specifications of ICC objects for the new application. Step (2) inserts hooks using the information from the ICC analysis. Using the newly computed ICC information as well as the information from other applications, this step can estimate which ICC links are sure to be safe with respect to the specified policy. Hook placement only happens at locations that may be unsafe. This way, we are hoping to significantly limit the number of hooks that will be called at runtime. Next, the application is installed on the target system in step (3). Finally, using the inserted hooks, the policy is enforced at runtime in step (4). Steps (1) and (2) are computationally intensive and are therefore performed by an entity external to the phone. A good candidate would be the application market.

We expect to have to tackle a number of issues:

- *Hooks:* We will need to answer several questions related to the hooks. For example, we need to determine what actions they should perform. We also need to determine what the best placement strategy is, i.e., should they be placed at entry points or at exit points?

- *Policies:* We need to determine what policies can be enforced by our system. We envision limiting the union of permissions of different applications, but we may add provisions to only limit links that carry data between applications if we deem it reasonable. We will determine an appropriate language for policy specification.

- *Evaluation:* Evaluation may be challenging, due to the dynamic nature of the enforcement. We will consider a combination of automated [146] and manual testing. We intend on measuring the performance impact of our hooks, comparing it with an approach with complete runtime enforcement as well as no enforcement at all. We will measure the number of blocked inter-application communications.

8.5 Concluding Remarks

The popularity of smart phone and tablet platforms is closely related to the abundance of third-party applications that implement a wide variety of features. These applications often implement a small number of features and reuse functionality implemented in different applications. The inter-component interfaces of an application determine both the ways in which other applications activate it and which applications its inter-component messages can target. Understanding inter-component communication is therefore indispensable as a basis for the analysis of the security of mobile applications.

We found that developers use relatively straightforward inter-component messages in most cases. This makes it possible to analyze ICC by reducing it to a novel kind of constant propagation problem. Solving these constant propagation problems is done in a completely generic manner, which allows for easy extension as the ICC API evolves. We were able to perform static ICC analysis using existing program analysis techniques because the new mobile application formats are not fundamentally different from existing ones. Therefore we found that program retargeting is a reliable prerequisite to mobile application analysis. The techniques presented in this dissertation can be reused as new application runtime environments and novel inter-component communication mechanisms are developed.

Appendix

Opcode Map f_{uo}

For completeness, we present the complete definition of opcode map f_{uo} for $\mathcal{O}_{uo}$, which was partially defined in Section 4.6. It is presented in Tables A.1, A.2 and A.3.

Tyde Opcode	**Java Opcode**
nop	nop
move, move/from16, move/16, move-wide, move-wide/from16, move-wide/16, move-object, move-object/from16, move-object/16, move-result, move-result-wide, move-result-object, move-exception	$\mathcal{J}_\epsilon$
return-object	areturn
const-wide/16, const-wide/32, const-wide, const-wide/high16	ldc2_w
const-string, const-string/jumbo, const-class, const-class/jumbo	ldc
monitor-enter	monitorenter
monitor-exit	monitorexit
check-cast, check-cast/jumbo	checkcast
instance-of, instance-of/jumbo	instanceof
array-length	arraylength
new-instance	new
throw	athrow
cmpl-float	fcmpl
cmpg-float	fcmpg
cmpl-double	dcmpl
cmpg-double	dcmpg
cmp-long	lcmp
aget-object	aaload
aget-boolean, aget-byte	baload
aget-char	caload
aget-short	saload
aput-object	aastore
aput-boolean, aput-byte	bastore
aput-char	castore
aput-short	satore
iget, iget/jumbo, iget-wide, iget-wide/jumbo, iget-object, iget-object/jumbo, iget-boolean, iget-boolean/jumbo, iget-byte, iget-byte/jumbo, iget-char, iget-char/jumbo, iget-short, iget-short/jumbo	getfield
iput, iput/jumbo, iput-wide, iput-wide/jumbo, iput-object, iput-object/jumbo, iput-boolean, iput-boolean/jumbo, iput-byte, iput-byte/jumbo, iput-char, iput-char/jumbo, iput-short, iput-short/jumbo	putfield

Table A.1. Opcode map f_{uo} for set $\mathcal{O}_{uo}$ (part 1).

Tyde Opcode	**Java Opcode**
sget, sget/jumbo, sget-wide, sget-wide/jumbo, sget-object, sget-object/jumbo, sget-boolean, sget-boolean/jumbo, sget-byte, sget-byte/jumbo, sget-char, sget-char/jumbo, sget-short, sget-short/jumbo	getstatic
sput, sput/jumbo, sput-wide, sput-wide/jumbo, sput-object, sput-object/jumbo, sput-boolean, sput-boolean/jumbo, sput-byte, sput-byte/jumbo, sput-char, sput-char/jumbo, sput-short, sput-short/jumbo	putstatic
invoke-virtual, invoke-virtual/range, invoke-virtual/jumbo	invokevirtual
invoke-super, invoke-super/range, invoke-super/jumbo, invoke-direct, invoke-direct/range, invoke-direct/jumbo	invokespecial
invoke-static, invoke-static/range, invoke-static/jumbo	invokestatic
invoke-interface, invoke-interface/range, invoke-interface/jumbo	invokeinterface
neg-int	ineg
neg-long	lneg
neg-float	fneg
neg-double	dneg
int-to-long	i2l
int-to-float	i2f
int-to-double	i2d
long-to-int	l2i
long-to-float	l2f
long-to-double	l2d
float-to-int	f2i
float-to-long	f2l
float-to-double	f2d
double-to-int	d2i
double-to-long	d2l
double-to-float	d2f
int-to-byte	i2b
int-to-char	i2c
int-to-short	i2s

Table A.2. Opcode map f_{uo} for set $\mathcal{O}_{uo}$ (part 2).

Tyde Opcode	**Java Opcode**
add-int, add-int/2addr, add-int/lit16, add-int/lit8	iadd
sub-int, sub-int/2addr, rsub-int, rsub-int/lit8	isub
mul-int, mul-int/2addr, mul-int/lit16, mul-int/lit8	imul
div-int, div-int/2addr, div-int/lit16, div-int/lit8	idiv
rem-int, rem-int/2addr, rem-int/lit16, rem-int/lit8	irem
and-int, and-int/2addr, and-int/lit16, and-int/lit8	iand
or-int, or-int/2addr, or-int/lit16, or-int/lit8	ior
xor-int, xor-int/2addr, xor-int/lit16, xor-int/lit8	ixor
shl-int, shl-int/2addr, shl-int/lit8	ishl
shr-int, shr-int/2addr, shr-int/lit8	ishr
ushr-int, ushr-int/2addr, ushr-int/lit8	iushr
add-long, add-long/2addr	ladd
sub-long, sub-long/2addr	lsub
mul-long, mul-long/2addr	lmul
div-long, div-long/2addr	ldiv
and-long, and-long/2addr	land
or-long, or-long/2addr	lor
xor-long, xor-long/2addr	lxor
shl-long, shl-long/2addr	lshl
shr-long, shr-long/2addr	lshr
ushr-long, ushr-long/2addr	lushr
add-float, add-float/2addr	fadd
sub-float, sub-float/2addr	fsub
mul-float, mul-float/2addr	fmul
div-float, div-float/2addr	fdiv
rem-float, rem-float/2addr	frem
add-double, add-double/2addr	dadd
sub-double, sub-double/2addr	dsub
mul-double, mul-double/2addr	dmul
div-double, div-double/2addr	ddiv

Table A.3. Opcode map f_{uo} for set $\mathcal{O}_{uo}$ (part 3).

Bibliography

[1] Arthur, C. (2012), "Feature phones dwindle as Android powers ahead in third quarter," *The Guardian*, available at `http://www.guardian.co.uk/technology/2012/nov/15/smartphones-market-android-feature-phones`.

[2] IDC, "Smart Connected Device Market by Product Category, Shipments, Market Share," Available from `http://images.techhive.com/images/article/2013/03/idc-smart-devices-100030871-orig.png`.

[3] Enck, W., M. Ongtang, and P. McDaniel (2009) "Understanding Android Security," *IEEE Security & Privacy Magazine*, **7**(1), pp. 50–57.

[4] Chin, E., A. P. Felt, K. Greenwood, and D. Wagner (2011) "Analyzing Inter-Application Communication in Android," in *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[5] Felt, A. P., H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin (2011) "Permission Re-delegation: Attacks and Defenses," in *Proceedings of the 20th USENIX Conference on Security*, SEC '11, USENIX Association, Berkeley, CA, USA.

[6] Schlegel, R., K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang (2011) "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*.

[7] Marforio, C., H. Ritzdorf, A. Francillon, and S. Capkun (2012) "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, ACM, New York, NY, USA, pp. 51–60.

[8] ZDNET, "Android OEMs slow to roll out Bluebox Security patch," Available from `http://www.zdnet.com/android-oems-slow-to-roll-out-bluebox-security-patch-7000018012/`.

[9] McDaniel, P. and W. Enck (2010) "Not So Great Expectations: Why Application Markets Haven't Failed Security," *IEEE Security & Privacy Magazine*, **8**(5), pp. 76–78.

[10] Octeau, D., S. Jha, and P. McDaniel (2012) "Retargeting Android Applications to Java Bytecode," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE '12, ACM, New York, NY, USA, pp. 6:1–6:11.

[11] Octeau, D., P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon (2013) "Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis," in *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, USENIX Association, Berkeley, CA, USA, pp. 543–558.

[12] Sagiv, M., T. Reps, and S. Horwitz (1996) "Precise interprocedural dataflow analysis with applications to constant propagation," *Theoretical Computer Science*, **167**(1-2), pp. 131–170.

[13] Proebsting, T. A. and S. A. Watterson (1997) "Krakatoa: Decompilation in Java (Dose Bytecode Reveal Source?)," in *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, COOTS '97, USENIX Association, Berkeley, CA, USA.

[14] van Vliet, H. (2001), "Mocha, the Java Decompiler," `http://www.brouhaha.com/~eric/software/mocha/`.

[15] Miecznikowski, J. and L. Hendren (2001) "Decompiling Java using staged encapsulation," in *WCRE '01: Proceedings of the 8th Working Conference on Reverse Engineering*, IEEE Computer Society.

[16] Miecznikowski, J. and L. J. Hendren (2002) "Decompiling Java Bytecode: Problems, Traps and Pitfalls," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, Springer-Verlag, London, UK, pp. 111–127.

[17] Naeem, N. A. and L. Hendren (2006) "Programmer-friendly Decompiled Java," in *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, IEEE Computer Society, pp. 327–336.

[18] Vallee-Rai, R., E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan (2000) "Optimizing Java Bytecode using the Soot Framework: Is it Feasible?" in *International Conference on Compiler Construction, LNCS 1781*, pp. 18–34.

[19] "JAD Java Decompiler Download Mirror," `http://http://www.varaneckas.com/jad`.

[20] Dupuy, E., "JD Java Decompiler," `http://java.decompiler.free.fr/`.

[21] Cifuentes, C. (1994) *Reverse Compilation Techniques*, Ph.D. thesis, Queensland University of Technology.

[22] Cifuentes, C. and K. J. Gough (1995) "Decompilation of Binary Programs," *Software – Practice and Experience*, **25**(7), pp. 811–829.

[23] Cifuentes, C. (1996) "Interprocedural Data Flow Decompilation," *Journal of Programming Languages*, **4**, pp. 77–99.

[24] Cifuentes, C., D. Simon, and A. Fraboulet (1998) "Assembly to High-Level Language Translation," in *Proceedings of the International Conference on Software Maintenance*, ICSM '98, IEEE Computer Society, Washington, DC, USA, pp. 228–.

[25] Schwartz, E. J., J. Lee, M. Woo, and D. Brumley (2013) "Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, USENIX, Washington, D.C., pp. 353–368.

[26] Bellamy, B., P. Avgustinov, O. de Moor, and D. Sereni (2008) "Efficient Local Type Inference," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, ACM, New York, NY, USA, pp. 475–492.

[27] Gagnon, E., L. J. Hendren, and G. Marceau (2000) "Efficient Inference of Static Types for Java Bytecode." in *SAS'00*, pp. 199–219.

[28] Knoblock, T. B. and J. Rehof (2001) "Type elaboration and subtype completion for Java bytecode," *ACM Transactions on Programming Languages and Systems*, **23**, pp. 243–272.

[29] Johnson, R. T. (2007) *Verifying Security Properties using Type-Qualifier Inference*, Ph.D. thesis, EECS Department, University of California, Berkeley.

[30] Myers, A. C. (1999) "JFlow: Practical Mostly-static Information Flow Control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, ACM, New York, NY, USA, pp. 228–241.

[31] Lim, J. and T. Reps (2008) "A System for Generating Static Analyzers for Machine Instructions," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 36–52.

[32] Ramsey, N. and J. Davidson (1999) "Specifying instruction semantics using λ–RTL," Unpublished manuscript.

[33] Morrisett, G., D. Walker, K. Crary, and N. Glew (1998) "From System F to Typed Assembly Language," in *The Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[34] Kildall, G. A. (1973) "A Unified Approach to Global Program Optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, ACM, New York, NY, USA, pp. 194–206.

[35] Kennedy, K. (1981) "A survey of data flow analysis techniques," *Program Flow Analysis: Theory and Applications*, pp. 5–54.

[36] Wegman, M. N. and F. K. Zadeck (1985) "Constant Propagation with Conditional Branches," in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, ACM, New York, NY, USA, pp. 291–299.

[37] Callahan, D., K. D. Cooper, K. Kennedy, and L. Torczon (1986) "Interprocedural Constant Propagation," in *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, ACM, New York, NY, USA, pp. 152–161.

[38] Wegman, M. N. and F. K. Zadeck (1991) "Constant Propagation with Conditional Branches," *ACM Transactions on Programming Languages and Systems*, **13**(2), pp. 181–210.

[39] Metzger, R. and S. Stroud (1993) "Interprocedural constant propagation: an empirical study," *ACM Letters on Programming Languages and Systems*, **2**(1-4), pp. 213–232.

[40] Grove, D. and L. Torczon (1993) "Interprocedural Constant Propagation: A Study of Jump Function Implementation," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, ACM, New York, NY, USA, pp. 90–99.

[41] Sharir, M. and A. Pnueli (1981) "Two approaches to interprocedural data flow analysis," *Program flow analysis: theory and applications*, pp. 189–234.

[42] Merlo, E., J.-F. Girard, L. J. Hendren, and R. d. Mori (1993) "Multi-Valued Constant Propagation for the Reengineering of User Interfaces," in *Proceedings of the 1993 International Conference on Software Maintenance (ICSM '93)*, IEEE Computer Society.

[43] Enck, W. (2011) "Defending Users Against Smartphone Apps: Techniques and Future Directions," in *Proceedings of the 7th International Conference on Information Systems Security*, ICISS '11, Springer-Verlag, Berlin, Heidelberg, pp. 49–70.

[44] Au, K. W. Y., Y. F. Zhou, Z. Huang, P. Gill, and D. Lie (2011) "Short Paper: A Look at Smartphone Permission Models," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, ACM, New York, NY, USA, pp. 63–68.

[45] Enck, W., M. Ongtang, and P. McDaniel (2009) "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM Conference on*

Computer and Communications Security, CCS '09, ACM, New York, NY, USA, pp. 235–245.

[46] Barrera, D., H. G. Kayacik, P. C. van Oorschot, and A. Somayaji (2010) "A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, ACM, New York, NY, USA, pp. 73–84.

[47] Peng, H., C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy (2012) "Using Probabilistic Generative Models for Ranking Risks of Android Apps," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM, New York, NY, USA, pp. 241–252.

[48] Sarma, B. P., N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy (2012) "Android Permissions: A Perspective Combining Risks and Benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, ACM, New York, NY, USA, pp. 13–22.

[49] Bartel, A., J. Klein, Y. Le Traon, and M. Monperrus (2012) "Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, ACM, New York, NY, USA, pp. 274–277.

[50] Au, K. W. Y., Y. F. Zhou, Z. Huang, and D. Lie (2012) "PScout: Analyzing the Android Permission Specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM, New York, NY, USA, pp. 217–228.

[51] Felt, A. P., E. Chin, S. Hanna, D. Song, and D. Wagner (2011) "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, ACM, New York, NY, USA, pp. 627–638.

[52] Zhou, Y., Z. Wang, W. Zhou, and X. Jiang (2012) "Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proceedings of the Network and Distributed System Security Symposium*.

[53] Saltzer, J. H. and M. D. Schroeder (1975) "The protection of information in computer systems," *Proceedings of the IEEE*, **63**(9), pp. 1278–1308.

[54] Wei, X., L. Gomez, I. Neamtiu, and M. Faloutsos (2012) "Permission Evolution in the Android Ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, ACM, New York, NY, USA, pp. 31–40.

[55] Sellwood, J. and J. Crampton (2013) "Sleeping Android: The Danger of Dormant Permissions," in *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '13, ACM, New York, NY, USA, pp. 55–66.

[56] Vidas, T., N. Cristin, and L. F. Cranor (2011) "Curbing Android Permission Creep," in *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)*.

[57] Sbirlea, D., M. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar (2013) "Automatic detection of inter-application permission leaks in Android applications," *IBM Journal of Research and Development*, **57**(6), pp. 10:1–10:12.

[58] Jeon, J., K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein (2012) "Dr. android and mr. hide: fine-grained permissions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, pp. 3–14.

[59] Pandita, R., X. Xiao, W. Yang, W. Enck, and T. Xie (2013) "WHYPER: Towards Automating Risk Assessment of Mobile Applications," in *22nd USENIX Security Symposium (USENIX Security '13)*, USENIX, Washington, D.C., pp. 527–542.

[60] Ongtang, M., S. McLaughlin, W. Enck, and P. McDaniel (2009) "Semantically Rich Application-Centric Security in Android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, IEEE Computer Society, Washington, DC, USA, pp. 340–349.

[61] Nauman, M., S. Khan, and X. Zhang (2010) "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, ACM, New York, NY, USA, pp. 328–332.

[62] Conti, M., V. T. N. Nguyen, and B. Crispo (2011) "CRePE: Context-related Policy Enforcement for Android," in *Proceedings of the 13th International Conference on Information Security*, ISC'10, Springer-Verlag, Berlin, Heidelberg, pp. 331–345.

[63] Xu, R., H. Saïdi, and R. Anderson (2012) "Aurasium: Practical Policy Enforcement for Android Applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Security '12, USENIX Association, Berkeley, CA, USA, pp. 27–27.

[64] Sunshine, J., S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor (2009) "Crying Wolf: An Empirical Study of SSL Warning Effectiveness," in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM '09, USENIX Association, Berkeley, CA, USA, pp. 399–416.

[65] Felt, A. P., E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner (2012) "Android Permissions: User Attention, Comprehension, and Behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, ACM, New York, NY, USA, pp. 3:1–3:14.

[66] Jung, J., S. Han, and D. Wetherall (2012) "Short Paper: Enhancing Mobile Application Permissions with Runtime Feedback and Constraints," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, ACM, New York, NY, USA, pp. 45–50.

[67] Kelley, P. G., S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall (2012) "A Conundrum of Permissions: Installing Applications on an Android Smartphone," in *Proceedings of the 16th International Conference on Financial Cryptography and Data Security*, FC '12, Springer-Verlag, Berlin, Heidelberg, pp. 68–79.

[68] Livshits, B. and J. Jung (2013) "Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*, USENIX, Washington, D.C., pp. 113–130.

[69] Felt, A. P., S. Egelman, M. Finifter, D. Akhawe, and D. Wagner (2012) "How to Ask for Permission," in *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, HotSec'12, USENIX Association, Berkeley, CA, USA.

[70] Lin, J., S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang (2012) "Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, ACM, New York, NY, USA, pp. 501–510.

[71] Yang, L., N. Boushehrinejadmoradi, P. Roy, V. Ganapathy, and L. Iftode (2012) "Short Paper: Enhancing Users' Comprehension of Android Permissions," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, ACM, New York, NY, USA, pp. 21–26.

[72] Hardy, N. (1988) "The Confused Deputy: (or why capabilities might have been invented)," *SIGOPS Operating Systems Review*, **22**(4).

[73] Wang, T., K. Lu, L. Lu, S. Chung, and W. Lee (2013) "Jekyll on iOS: When Benign Apps Become Evil," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*, USENIX, Washington, D.C., pp. 559–572.

[74] Fuchs, A., A. Chaudhuri, and J. Foster (2009) "SCanDroid: Automated Security Certification of Android Applications," Unpublished manuscript.

[75] Reps, T., S. Horwitz, and M. Sagiv (1995) "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, ACM, New York, NY, USA, pp. 49–61.

[76] Lu, L., Z. Li, Z. Wu, W. Lee, and G. Jiang (2012) "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM, New York, NY, USA, pp. 229–240.

[77] Grace, M., Y. Zhou, Z. Wang, and X. Jiang (2012) "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS '12)*.

[78] Chan, P. P., L. C. Hui, and S. M. Yiu (2012) "DroidChecker: Analyzing Android Applications for Capability Leak," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, ACM, New York, NY, USA, pp. 125–136.

[79] Wu, L., M. Grace, Y. Zhou, C. Wu, and X. Jiang (2013) "The Impact of Vendor Customizations on Android Security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, ACM, New York, NY, USA, pp. 623–634.

[80] Zhou, Y. and X. Jiang (2013) "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Annual Symposium on Network and Distributed System Security (NDSS '13)*.

[81] Bugiel, S., L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry (2011) "Practical and Lightweight Domain Isolation on Android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, ACM, New York, NY, USA, pp. 51–62.

[82] Bugiel, S., L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi (2011) *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*, *Tech. Rep. TR-2011-04*, Technische Universitat Darmstadt, Germany.

[83] Dietz, M., S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach (2011) "Quire: Lightweight Provenance for Smart Phone Operating Systems," in *20th USENIX Security Symposium*.

[84] Smalley, S. and R. Craig (2013) "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *20th Annual Network and Distributed System Security Symposium (NDSS'13)*.

[85] Loscocco, P. and S. Smalley (2001) "Integrating Flexible Support for Security Policies into the Linux Operating System," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, pp. 29–42.

[86] Bugiel, S., S. Heuser, and A.-R. Sadeghi (2013) "Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC '13, USENIX Association, Berkeley, CA, USA, pp. 131–146.

[87] Felt, A. P., M. Finifter, E. Chin, S. Hanna, and D. Wagner (2011) "A Survey of Mobile Malware in the Wild," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, ACM, New York, NY, USA, pp. 3–14.

[88] Zhou, Y. and X. Jiang (2012) "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, IEEE Computer Society, Washington, DC, USA, pp. 95–109.

[89] Enck, W., P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth (2010) "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI' 10, USENIX Association, Berkeley, CA, USA.

[90] Gilbert, P., B.-G. Chun, L. P. Cox, and J. Jung (2011) "Vision: Automated Security Validation of Mobile Apps at App Markets," in *Proceedings of the Second International Workshop on Mobile Cloud Computing and Services*, MCS '11, ACM, New York, NY, USA, pp. 21–26.

[91] Hornyack, P., S. Han, J. Jung, S. Schechter, and D. Wetherall (2011) "These Aren't the Droids You're Looking for: Retrofitting Android to Protect Data from Imperious Applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, ACM, New York, NY, USA, pp. 639–652.

[92] Zhou, Y., X. Zhang, X. Jiang, and V. W. Freeh (2011) "Taming Information-stealing Smartphone Applications (on Android)," in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, Springer-Verlag, Berlin, Heidelberg, pp. 93–107.

[93] Beresford, A. R., A. Rice, N. Skehin, and R. Sohan (2011) "MockDroid: Trading Privacy for Application Functionality on Smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, ACM, New York, NY, USA, pp. 49–54.

[94] Zheng, C., S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou (2012) "SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, ACM, New York, NY, USA, pp. 93–104.

[95] Gibler, C., J. Crussell, J. Erickson, and H. Chen (2012) "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, Springer-Verlag, Berlin, Heidelberg, pp. 291–307.

[96] Weiser, M. (1981) "Program Slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, IEEE Press, Piscataway, NJ, USA, pp. 439–449.

[97] Yang, Z. and M. Yang (2012) "LeakMiner: Detect Information Leakage on Android with Static Taint Analysis," in *Proceedings of the 2012 Third World Congress on Software Engineering*, WCSE '12, IEEE Computer Society, Washington, DC, USA, pp. 101–104.

[98] Book, T., A. Pridgen, and D. Wallach (2013) "Longitudinal Analysis of Android Ad Library Permissions," in *Proceedings of the 2013 Mobile Security Technologies (MoST) Workshop*.

[99] Grace, M. C., W. Zhou, X. Jiang, and A.-R. Sadeghi (2012) "Unsafe Exposure Analysis of Mobile In-app Advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, ACM, New York, NY, USA, pp. 101–112.

[100] Egele, M., C. Kruegel, E. Kirda, and G. Vigna (2011) "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*.

[101] Kim, J., Y. Yoon, and K. Yi (2012) "ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications," in *MoST 2012: Workshop on Mobile Security Technologies 2012*.

[102] Arzt, S., S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel (2014) "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM, New York, NY, USA, pp. 259–269.

[103] Reps, T., S. Horwitz, and M. Sagiv (1995) "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, ACM, New York, NY, USA, pp. 49–61.

[104] Zhang, M. and H. Yin (2014) "AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications," in *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS '14)*.

[105] Rasthofer, S., S. Arzt, and E. Bodden (2014) "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks," in *Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium*.

[106] Cortes, C. and V. Vapnik (1995) "Support-vector networks," *Machine Learning*, **20**(3), pp. 273–297.

[107] Enck, W., D. Octeau, P. McDaniel, and S. Chaudhuri (2011) "A Study of Android Application Security," in *Proceedings of the 20th USENIX Conference on Security*, SEC '11, USENIX Association, Berkeley, CA, USA, pp. 21–21.

[108] Shekhar, S., M. Dietz, and D. S. Wallach (2012) "AdSplit: Separating Smartphone Advertising from Applications," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security '12)*, USENIX, Bellevue, WA, pp. 553–567.

[109] Pearce, P., A. P. Felt, G. Nunez, and D. Wagner (2012) "AdDroid: Privilege Separation for Applications and Advertisers in Android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, ACM, New York, NY, USA, pp. 71–72.

[110] Genaim, S. and F. Spoto (2005) "Information Flow Analysis for Java Bytecode," in *Verification, Model Checking, and Abstract Interpretation* (R. Cousot, ed.), vol. 3385 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 346–362.

[111] Livshits, V. B. and M. S. Lam (2005) "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th USENIX Security Symposium - Volume 14*, SSYM '05, USENIX Association, Berkeley, CA, USA.

[112] Bernardeschi, C., N. De Francesco, G. Lettieri, and L. Martini (2004) "Checking secure information flow in java bytecode by code transformation and standard bytecode verification," *Software – Practice and Experience*, **34**(13), pp. 1225–1255.

[113] Liu, Y. and A. Milanova (2008) "Static analysis for inference of explicit information flow," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '08, ACM, New York, NY, USA, pp. 50–56.

[114] Liu, Y. and A. Milanova (2009) "Practical static analysis for inference of security-related program properties," in *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC '09)*, pp. 50–59.

[115] Liu, Y. and A. Milanova (2010) "Static Information Flow Analysis with Handling of Implicit Flows and a Study on Effects of Implicit Flows vs Explicit Flows," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, IEEE Computer Society, Washington, DC, USA, pp. 146–155.

[116] Google, "Dalvik VM: Code and documentation," `http://code.google.com/p/dalvik/`.

[117] WALA, "T.J. Watson Libraries for Analysis," `http://wala.sourceforge.net`.

[118] HP-Fortify, "Source Code Analyzer (SCA)," `https://www.fortify.com/products/hpfssc/source-code-analyzer.html`.

[119] Tiuryn, J. (1990) "Type Inference Problems: A Survey," in *MFCS '90: Proceedings of the Mathematical Foundations of Computer Science 1990*, Springer-Verlag, pp. 105–120.

[120] Milner, R. (1978) "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, **17**, pp. 348–375.

[121] Octeau, D., W. Enck, and P. McDaniel (2010) *The ded Decompiler*, *Tech. Rep. NAS-TR-0140-2010*, Network and Security Research Center, Pennsylvania State University, USA.

[122] Meyer, J., D. Reynaud, and I. Kharon (2004), "Jasmin Home Page," `http://jasmin.sourceforge.net/`.

[123] Lindholm, T. and F. Yellin (1999) *Java Virtual Machine Specification*, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[124] Rehof, J. and T. A. Mogensen (1996) "Tractable Constraints in Finite Semilattices," in *Science of Computer Programming*, Springer-Verlag, pp. 285–300.

[125] "dex2jar - Tools to work with android .dex and java .class files," Available at `http://code.google.com/p/dex2jar/`.

[126] Octeau, D., P. McDaniel, and W. Enck (2011), "ded: Decompiling Android Applications," `http://siis.cse.psu.edu/ded/`.

[127] Oracle (2004), "Maxine VM," `https://wikis.oracle.com/display/MaxineVM/Home`.

[128] Davi, L., A. Dmitrienko, A.-R. Sadeghi, and M. Winandy (2011) "Privilege Escalation Attacks on Android," in *Proceedings of the 13th International Conference on Information Security*, ISC '10, Springer-Verlag, Berlin, Heidelberg, pp. 346–360.

[129] Bugiel, S., L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry (2012) "Towards Taming Privilege-Escalation Attacks on Android," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS '12)*.

[130] Bodden, E. (2012) "Inter-procedural Data-flow Analysis with IFDS/IDE and Soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, ACM, New York, NY, USA, pp. 3–8.

[131] Vallée-Rai, R., E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan (2000) "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?" in *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, Springer-Verlag, London, UK, UK, pp. 18–34.

[132] Christensen, A. S., A. Møller, and M. I. Schwartzbach (2003) "Precise Analysis of String Expressions," in *Proceedings of the 10th International Conference on Static Analysis*, SAS '03, Springer-Verlag, Berlin, Heidelberg, pp. 1–18.

[133] Lhoták, O. and L. Hendren (2003) "Scaling Java points-to analysis using SPARK," in *Proceedings of the 12th international conference on Compiler construction*, CC'03, Springer-Verlag.

[134] Octeau, D., D. Luchaup, M. Dering, S. Jha, and P. McDaniel (2013) *Android Inter-Component Communication Analysis with the COAL Constant Propagation Language*, *Tech. Rep. NAS-TR-0170-2014*, Institute for Network and Security Research, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA.

[135] Li, L., A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel (2014) "I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis," *CoRR*, **abs/1404.7431**.

[136] Lad, M., D. Massey, and L. Zhang (2006) "Visualizing Internet Routing Changes," *IEEE Transactions on Visualization and Computer Graphics*, **12**(6), pp. 1450–1460.

[137] Henry, S. and D. Kafura (1981) "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, **SE-7**(5), pp. 510–518.

[138] Grady, R. (1994) "Successfully applying software metrics," *Computer*, **27**(9), pp. 18–25.

[139] Aiken, A. (1999) "Introduction to Set Constraint-based Program Analysis," *Science of Computer Programming*, **35**(2-3), pp. 79–111.

[140] Gelade, W. and F. Neven (2012) "Succinctness of the Complement and Intersection of Regular Expressions," *ACM Transactions on Computer Logic*, **13**(1).

[141] Android Police, "Meet ART, Part 1: The New Super-Fast Android Runtime Google Has Been Working On In Secret For Over 2 Years Debuts In KitKat," `http://www.androidpolice.com/2013/11/06/meet-art-part-1-the-new-super-fast-android-runtime-google-has-been-working-on-in-secret-for-over-2-years-debuts-in-kitkat/`.

[142] Wettel, R. and M. Lanza (2007) "Visualizing software systems as cities," in *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*, IEEE, pp. 92–99.

[143] Steinbrückner, F. and C. Lewerentz (2010) "Representing Development History in Software Cities," in *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, ACM, New York, NY, USA, pp. 193–202.

[144] Wettel, R., M. Lanza, and R. Robbes (2011) "Software Systems As Cities: A Controlled Experiment," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, ACM, New York, NY, USA, pp. 551–560.

[145] Caserta, P., O. Zendra, and D. Bodénes (2011) "3D hierarchical edge bundles to visualize relations in a software city metaphor," in *6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011)*, IEEE, pp. 1–8.

[146] Google Android Developers, "UI/Application Exerciser Monkey," Available from `http://developer.android.com/tools/help/monkey.html`.

Vita

Damien Octeau

EDUCATION

Pennsylvania State University, University Park, PA 2014
Ph.D. in Computer Science and Engineering
Advisor: Dr. Patrick McDaniel

Pennsylvania State University, University Park, PA 2010
M.S. in Computer Science and Engineering
Advisor: Dr. Patrick McDaniel

Ecole Centrale de Lyon, Ecully, France 2010
Diplôme d'ingénieur (Master's degree in Engineering)

Ecole Centrale de Lyon, Ecully, France 2007
B.S. in Engineering

HONORS AND AWARDS

AT&T Graduate Fellowship, 2013

Best Research Artifact Award, 20th International Symposium on the Foundations of Software Engineering (FSE), 2012

USENIX Security Symposium Travel Grant, 2009, 2011, 2013

EXPERIENCE

Research Assistant 2009 - 2014
Pennsylvania State University *University Park, PA*
Performed research in mobile application analysis and security.

Intern Summer 2013
Google Inc. (Security Team) *Mountain View, CA*
Designed and implemented a system to analyze Linux binaries in a virtualized environment.

Intern Summer 2011
Google Inc. (Mobile Apps Lab Team) *Mountain View, CA*
Designed and implemented tools to analyze and visualize experimental data about user proximity.

Teaching Assistant Fall 2010
Pennsylvania State University *University Park, PA*
Assisted students in labs and office hours for CMPSC 122, the course in intermediate programming in C++. Performed grading of homework assignments and exams.

Intern Summer 2008
Osiatis France *Mérignac, France*
Designed and implemented a collaborative php/MySQL application to manage the follow up on all IT issues for a major client. Trained the local team to use and modify it.