SAARLAND UNIVERSITY
Faculty of Natural Sciences and Technology I
Department of Computer Science

MASTER'S THESIS

# Verifying the Internet Access of Android Applications

submitted by

Erik Derr

submitted December 22, 2011

Supervisor
Prof. Dr. Michael Backes
Dr. Matteo Maffei

Reviewer
Prof. Dr. Michael Backes
Dr. Matteo Maffei

Advisor
Sebastian Gerling

## Abstract

Google's mobile operating system Android has become the top-selling operating system on current smartphones. Its increasing popularity makes it also an attractive target for malware authors. Although Android's security model contains sophisticated mechanisms to harden attacks, the built-in security features often turn out to be insufficient to protect users from malicious applications or from undesired data leakage. This master's thesis presents a novel analysis approach to provide network communication transparency in Android applications. It performs comprehensive offline certification to verify Internet access that is commonly required by malicious applications to silently transmit sensitive user data to remote servers. To provide full transparency of the Internet usage of applications, the analysis recovers destinations and payload for any network connection and provides them in a comprehensive way to the user. This clearly helps users to understand application behavior and to decide whether a certain application is safe to be installed.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In 2008, Google published its mobile operating system Android. In contrast to competing operating systems like Apple's iOS or Microsoft's Windows Mobile, the Android system is completely open-source. Any manufacturer that is about to enter the smartphone market can use and customize Android without having to spent massive amounts of money for license fees. As a result, Android became very popular in short time. In the same year, the Open Handset Alliance [2], a consortium of various hardware and software companies, was founded to develop Android as open mobile platform.

With its increasing popularity, Android has become the top-selling operating system on current smartphones. Its market share has grown constantly over the last years to finally surpass Nokia's Symbian in 2011, which has been the market leader for years [32, 33, 42, 16].

Another important factor for the popularity of a mobile operating system is the distribution platform for applications. Shortly after publishing Android, Google announced the Android Market[1] as primary software store for Android applications. Current market statistics estimate a number of 340,000 applications in the official market in December, 2011[2]. Recently, Google announced that the Android market exceeded 10 billion app downloads with a growth rate of one billion app downloads per month [10]. With these numbers, Android reached a 44% worldwide share of mobile app downloads and overtook the current leader Apple [1].

Google's liberal application policy does also allow third-party marketplaces. Among the most popular ones is the Amazon Appstore, which opened in March, 2011[3]. Users and application developers benefit from this market diversity comparably. Developers reach a wider audience and can choose the market that provides the best conditions. On the other side, users are not bound to a single market. Competing markets have to offer attractive application prices to attract new customers. Users also benefit from special features like Amazon's *free app*

[1]`https://market.android.com`
[2]`http://www.appbrain.com/stats/number-of-android-apps`
[3]`http://www.amazon.com/appstore`

*a day* actions. However, the great popularity also has some downsides. The Android operating system and its application landscape have quickly moved into the focus of attackers and malware authors.

## 1.1 Motivation & Thesis Objectives

In the last years, the number of smartphone users has increased quickly. Currently, every fourth sold mobile device is a smartphone [33]. With increasing hardware capabilities, modern smartphones are no longer solely used for traditional tasks like phone calls and transmission of messages. They are increasingly often used for a wide range of functions that have previously been exclusive to computers like the usage of video portals, social networking, and even online banking. This also implies that users store more and more personal data on their smartphones.

A careful handling of sensitive user information is therefore essential for applications. However, most applications lack transparency and do not fully reveal what information is accessed and how it is processed or even is transmitted to remote servers. In order to prevent malicious applications from being uploaded in app stores, companies like Apple or Microsoft apply an application review process. Any submitted application that does not comply with their official content policy is declined. These reviews certainly do not detect all applications with malicious functionality, but they provide a basic security check.

Google does not apply any application vetting. Instead, the Android system uses a mechanism called mandatory access control to provide application security. Applications have to require permissions in order to access extended functionality or sensitive data. By omitting the manual review process, this approach allows a faster deployment of new applications and creates a more dynamic market landscape. But then, malware developers can register multiple accounts and can easily upload and publish malicious application packages.

The quickly rising Android application landscape also resulted in a dramatic increase of malicious code in the app store. With the constantly growing prevalence of Android smartphones, the market of mobile applications has evolved to a lucrative market for malware developers [24, 44]. In 2011, the number of malware reports escalated. Malware like DroidDream, Plankton, or DroidKungFu and its variants have impacted hundreds of thousands of Android users. Their common functionality includes the retrieval and subsequent transmission of sensitive user data.

The lack of application vetting along with the increasing popularity of Android, makes the official market a popular target for new malware. Since Google itself can not prevent malicious code from being uploaded and published, they installed a reporting mechanism. This way, users can inform Google about potential malware. Once confirmed, the concerning applications are removed from the market. In addition, Google can trigger a kill switch to remove the concerning software on all smartphones [45, 9, 41].

However, similar to signature-based detection systems, this only fights the effects and does not prevent zero-day malware. Therefore, it is crucial to detect malware before it is deployed to smartphones. Current Android malware reports [61, 44] state that many malicious applications require the Internet permission to communicate via the network and to transmit sensitive user data to remote servers. Thus, analyzing the Internet usage of Android applications would provide great benefit by making network communication completely transparent to users.

The goal of this thesis is to develop an analysis method for Android applications that performs a comprehensive verification of the Internet access. The first essential task is to devise a suitable analysis approach. Android's permission model and, in particular, the Internet permission is to be analyzed to check how applications can establish connections to remote servers. Finding a method for analyzing the Dalvik bytecode included in the application packages is then required to perform the actual application analysis. All network data sinks within the application have to be identified. Then, any data that reaches these methods has to be recovered. The most important information is the destination of outgoing requests. In order to provide full transparency to the user, another task includes the reassembly of data that leaves the smartphone via the network communication. Sensitive user data within a request payload is often a strong indicator for malicious intent or undesired data leakage. The revelation of the entire network communication is then used to verify the Internet usage of Android applications.

Another thesis objective constitutes the implementation of the analysis approach. A concrete tool, *Bati*, is to be developed to perform comprehensive offline-certification for Android applications. The analysis results are to be presented to the user in an informative and understandable manner. Based on the output, the user can then decide whether or not it is safe to install the analyzed application.

## 1.2 Outline

The remaining thesis describes in detail the analysis framework and the approaches used within. It is organized as follows:

*Chapter 2* introduces the mobile operating system Android. It gives a rough overview of the core system and important features. The runtime environment with Android's virtual machine and its bytecode are highlighted in particular, as they are essential basics for the analysis framework.

A high level overview of *Bati*, the analysis framework developed in this thesis, is then presented in *Chapter 3*. It briefly describes the functionality and tasks of components that are used within the analysis tool.

Before the actual information-flow analysis approach is described in detail, *Chapter 4* gives a brief introduction on common concepts of data-flow analysis, which will be used within the analysis framework. To facilitate understanding, each concept is illustrated with a descriptive example.

*Chapter 5* represents the core part of this thesis. It explains in detail how information-flow analysis techniques are applied to Android applications. In order to accomplish a comprehensive verification of the Internet access, multiple steps have to be performed. It starts with processing the application bytecode and generating the necessary data structures for the analysis. Backwards symbolic execution is then used to resolve register values that reach methods for network communication. Finally, this chapter describes how data is stored during analysis and is later assembled and output to the user.

The theory about the application analysis approach was presented in the last chapters. One goal of this thesis is to implement the devised techniques in a concrete tool that can be used for offline-certification. *Chapter 6* provides information about important implementation-specific details. This includes all third-party tools and algorithms that are used to perform certain tasks within the framework. Furthermore, necessary bytecode modifications and problems that occurred during bytecode processing are described in detail.

In order to show the effectiveness of the analysis framework, *Chapter 7* presents common programming paradigms and describes how Bati processes and evaluates them. Further, concrete analysis results are presented and their benefit to network communication transparency is highlighted.

*Chapter 8* presents current research that targets the conceptual improvement of the Android operating system and the detection of malware and information leakage. *Chapter 9* concludes this thesis by showing approach and implementation-specific limitations of the framework. It continues by summarizing the results and emphasizing the contributions of this thesis. Finally, topics for future work are proposed.

# Chapter 2

# The Android Operating System

Android is an operating system for mobile devices such as smartphones and tablet computers. It was originally developed by Android, Inc., which was acquired by Google back in July 2005. In November 2007, the Open Handset Alliance [2] was founded to develop Android as open mobile platform. Currently, the alliance is a consortium of 84 hardware, software, and telecommunication companies. In cooperation with the foundation, the first initial version of the SDK was published by Google. Most of the Android code is released under the Apache License in version two.

It took about one year until HTC released the first smartphone running Android. The HTC Dream, in the USA and parts of Europe also marketed as T-Mobile G1, became available in October 2008. At that time, Google also released most of the source code of Android (some Google apps were omitted). The software is hosted at the Android Open Source Project and the source code is freely available from a central Git repository[1].

The first official Android software version 1.0 was published in 2008. Since its original release there have been a number of updates. These updates typically target either the base operating system or the system applications that are shipped with Android. They include bugfixes, add new features or enhance existing ones. Each Android version has an official code name that is based on the name of a dessert. Examples include Cupcake (1.5), Eclair (2.0/2.1) and the latest release Ice Cream Sandwich (4.0).

The actual Android software stack comprises a Linux kernel, a middleware with libraries, a runtime environment, and an application framework for executing applications (commonly abbreviated by *apps*). *Figure 2.1* depicts a high-level view on the Android architecture.

[1]`https://android.git.kernel.org`

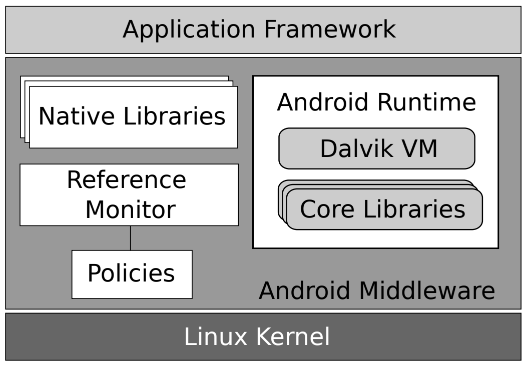


Figure 2.1: High-level view on the Android architecture

## 2.1 Kernel & Middleware

Current versions of Android use a customized version of the open-source Linux kernel series 2.6. By choosing Linux as base component, Android itself does not have to provide and implement core functionality, like process and thread management. It additionally benefits from the memory management and the proven driver model. The original Android is an ARM-based platform but there is ongoing work on porting it to the MIPS and x86 [63] architecture.

Android's security framework features a sandboxing mechanism to provide application security. This mechanism is implemented on operating system level in the underlying Linux. Each application in the user-space runs in its own instance of the virtual machine Dalvik, which itself is running in its own process. The Linux kernel assigns each VM process a unique user ID which allows the process to access only its own data and a very limited set of system features. It cannot interfere with data of other applications.

In order to support complex systems, applications must be able to communicate and share data. Because of the sandboxing approach, direct communication between processes is not possible. To enable controlled inter-process communication (IPC), Android uses the Binder framework[2]. Binder is a driver to facilitate IPC and has been integrated into the custom Android Linux kernel. It minimizes the processing overhead of IPC by supporting shared memory. Furthermore, it allows synchronous calls between processes.

Applications in Android are composed of components to provide modularity and to facilitate component reuse. *Figure 2.2* shows the communication between application components via Binder. Thereby, a reference monitor mediates the communication between components by using mandatory access control. The access control mechanism is based on permissions, which applications can request via a special manifest file (see *Section 2.3.1*).

The Android middleware includes libraries and APIs written in *C* and a runtime environment with the virtual machine Dalvik. In contrast to other Linux-based

[2]Android uses a custom implementation of OpenBinder[49]

systems, Android uses a custom libc implementation called Bionic, which is not compatible to the GNU libc. All native code must be compiled against Bionic.

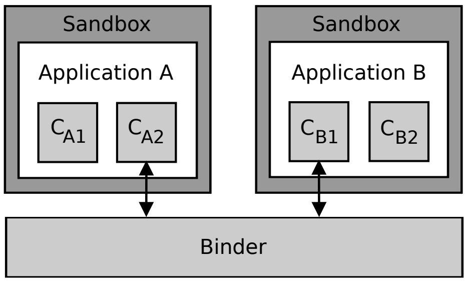


Figure 2.2: Inter-process communication

There are several reasons for not using the common libc library. Google wants to keep GPL-licensed software out of the Android user-space because this would restrict licensing of derivative works. Bionic is published under a permissive BSD license which only places minimal restrictions on how the software can be redistributed. It's a lightweight libc implementation optimized for a small memory footprint and embedded processors.

Since the application security is completely enforced on operating system level, the Dalvik VM itself is not concerned with runtime security. The virtual machine is not limited to executing dex bytecode. Through mechanisms like the Java Native Interface (JNI) or traditional Unix FIFOs, native code can be executed and thus the application can pop out of the virtual machine.

## 2.2 Android Runtime and the Dalvik Virtual Machine

The Android runtime consists of the Dalvik virtual machine [12] and some core libraries, which inherit almost all features provided by core libraries of the Java programming language. The virtual machine was originally written by Dan Bornstein and named after an Icelandic fishing village in which some of his ancestors lived.

There are some reasons on why Google prefers a custom virtual machine implementation for application virtualization to a well proven Java VM. Given the design decision to run each application in its own virtual machine and the natural resource limits from embedded devices, the VM must keep the memory footprint rather small. Furthermore, it needs to react quickly as applications are started and closed frequently. It should keep a high performance even if multiple instances are running. In order to meet these requirements, several optimizations have been devised and implemented.

Dalvik is capable of executing applications written in Java. It does not support Java directly, but the SDK provides a tool called dx which takes Java class files and converts them into dalvik executable (dex) bytecode. As of Android

2.2, Dalvik contains a JIT-Compiler for runtime bytecode optimization [14]. Performance tests showed that the Just-In-Time compiler delivers a speed gain up to 5 times faster in cpu-intensive workloads.

The Dalvik VM is not limited to executing dex bytecode. It is also capable of executing native code. Google offers a Native Development Kit[3] which can be used to generate native code for Android from C/C++ sources. In general, native code runs faster than bytecode executed by a virtual machine. Thus, it is a suitable choice for time-critical applications or for speeding up heavy computational tasks that appear for example in complex games. Intensive benchmarking [6] showed that native code runs up to 10 times faster than plain Java code. In Android applications, native code can be executed via JNI or named pipes. The JNI approach is preferable in data-intensive tasks, since the code runs in the same thread as the main application. No new process has to be spawned and the expensive Java I/O bottleneck is avoided.

### 2.2.1 The dx compiler

The dx compiler is the main tool to convert Java bytecode into dex bytecode. Given a set of class files, the dx compiler generates a single dex file named *classes.dex*. It uses the concept of shared constant pools to efficiently eliminate duplicates in the Java bytecode. Dex bytecode stores constant values only once, which dramatically reduces the size of the resulting bytecode.

*Table 2.1* shows a comparison of bytecode sizes of common system libraries and applications that are shipped with Android. The concept of shared constant pools results in a significant size reduction of the final bytecode[12].

| Code | Uncompressed JAR File (Bytes) | Compressed JAR File (Bytes) | Uncompressed Dex File (Bytes) |
|---|---|---|---|
| Common System Libraries | 21,445,320 (100%) | 10,662,048 (50%) | 10,311,972 (48%) |
| Web Browser App | 470,312 (100%) | 232,065 (49%) | 209,248 (44%) |
| Alarm Clock App | 119,200 (100%) | 61,658 (52%) | 53,020 (44%) |

Table 2.1: Size comparison of common Android system libraries and applications

### 2.2.2 The Zygote

Another optimization concept targets the memory usage and reaction times of the virtual machine. Traditional Java VM instances hold an entire copy of core library classes and associated heap objects. There is no memory sharing across VM instances. Additionally, a cold startup of virtual machines always takes excruciating long time. Cold startup means, that core libraries always have to be loaded before the actual application can be executed. In the context of mobile devices, it is common to start/stop applications and to switch between them. Long pre-loading sequences are not accepted by users.

[3]The NDK is available at `http://developer.android.com/sdk/ndk/index.html`

Since every application runs in its own VM instance, new virtual machines must be spawned quickly. In addition, the memory footprint must be kept minimal, as memory is always a very limited resource in mobile devices. Android uses a concept called Zygote [12, 23] to solve these issues. It provides both code sharing across VM instances and fast startup times of new VM instances. Zygote assumes that there are several core library classes and corresponding heap structures that are used by many applications. These structures are kept in memory and are usually accessed read-only, which means that applications will read this data, but never modify it. This observation is exploited to optimize the memory sharing across applications.

Zygote itself is also a VM process and is started by the init process at system boot time. During startup, Zygote spawns a Dalvik VM which pre-loads and pre-initializes common core library classes. Afterwards, it opens a socket and waits for requests from the runtime process to fork new child VMs for applications. This dramatically reduces the time necessary to fully initialize and start a new virtual machine.

The Zygote concept also offers mechanisms to handle rare cases in which applications write to shared memory. It uses a *copy-on-write* technique to allow such write operations. The concerning memory region is copied to the application VM process, that wants to write data to it. This prevents processes from interfering with each other and provides security across process boundaries.

### 2.2.3 Register-based Virtual Machine

Traditional JVMs, that operate on Java Bytecode, are usually stack-based virtual machines. Stack-based architectures are easier to implement but suffer from a weak performance and a larger memory overhead, since the stack data structure needs to be held in memory.

Shi *et al.* [59] compared stack-based JVM architectures against a register-based architecture. They pointed out that register-based architectures require an average of 47% less executed VM instructions compared to a stack-based architecture. On the other hand, the code size for the register-based VM is about 25% larger than the corresponding code for the stack-based VM. After executing some standard benchmarks, they concluded that, on average, the register-based VM takes 32.3% less time to finish.

Given these results, the decision to implement a register-based virtual machine for Android seems to be appropriate. Android reduces the code size through the concept of shared constant pools. The resulting code reduction of 50% offsets the larger instruction size. Taking all aspects into consideration, the Dalvik VM ends up with an improved memory usage as compared to a stack-based VM.

### 2.2.4 Bytecode for the Dalvik VM

Android's custom register-based Dalvik VM also introduces a new bytecode format. As the remainder of this thesis will focus on the static analysis of this bytecode, it is necessary to introduce the basics of the bytecode format.

The dex bytcode currently includes 218 Instructions [52]. They can be grouped semantically, for example in move, invoke, or binary operation instructions.

Dex bytecode instructions are identified via an unique 2-digit hexadecimal ID. For user convenience there also exist human-readable mnemonics (see *Table 2.2*). Furthermore, the instruction format follows a strict argument ordering, i.e. the destination register is always placed before any source registers.

The default register size in Dalvik is 32-bit. Adjacent register pairs are used for 64-bit values of type *long* and *double*. In general, dex bytecode instructions do not necessarily preserve the data type. For example instruction *14* does specify an arbitrary 32-bit constant, which can be either an integer or a float value.

| Op | Mnemonic / Syntax | Arguments |
|---|---|---|
| 01 | move vA, vB | A: destination register (4 bits)<br>B: source register (4 bits) |
| 14 | const vAA, #+BBBBBBBB | A: destination register (8 bits)<br>B: arbitrary 32-bit constant |
| 9b | add-long vAA, vBB, vCC | A: destination register (8 bits)<br>B: first source register (8 bits)<br>C: second source register (8 bits) |
| bb | add-long/2addr vA, vB | A: destination and first source register (4 bits)<br>B: second source register (4 bits) |

Table 2.2: Dalvik bytecode instruction examples

For further optimization, the bytecode specification includes 2-address instructions for binary operations, indicated by a trailing *2addr*. These size-optimized instructions perform a binary operation with two registers. The destination register in which the result of the operation is stored, is also the first source register.

Binary operations on integers are very common in applications. The default binary instructions suffer from the fact that numeric constants have to be put into registers before they can be processed. Dalvik developers realized that this area has potential for optimization and thus implemented some additional opcodes.

Through dedicated binary operation opcodes, the Dalvik VM can execute integer operations with one literal operand directly. The literal value does not have to be put in an extra register first. This usually reduces the total number of registers used within a method. Furthermore, the performance is improved, since the additional move operation for the literal value can be omitted. These dedicated binary operations are allowed for 8-bit and 16-bit integer values and are labeled with a trailing *lit8* and *lit16*.

Working directly with the compact dex bytecode format is usually infeasible and inconvenient. Therefore, there exist disassembler, that translate the compact bytecode into a more convenient assembly language. The Android SDK comes with a default disassembler called dexdump, but there also exist some open-source dex disassembler which provide more functionality. More information about these tools is provided in *Section 6.1.*

## 2.3 Application Framework

Applications for Android are written in Java and are composed of a set of components. The most important component types are *activities*, *intents*, *services*, and *content providers.* An activity is a user interface screen. Android applications usually consist of several activities that differ in functionality and possibilities for user interaction. An intent is a mechanism for describing a specific action like "open a URL in the web browser". Services are tasks that run in the background without the user's direct interaction. A content provider is a set of data that can be accessed and modified by use of a previously defined custom API.

This component-based architecture provides a modular application structure and enhances component reuse. The definition and relationship of these components is specified in a manifest file called *AndroidManifest.xml.* Application developers must also use this file to specify any permission that extends the feature set of the application.

Another security feature used in Android is application signing. It allows the verification of the origin of an application. Developers use self-signed certificates to sign their application code. Applications signed with the same key are executed with the same user ID and are thus able to share information without additional permissions. Furthermore, the certificates ensure that application updates really come from the original application developer.

### 2.3.1 Permission Model

The official Android Market is not based on application vetting. Instead, Android uses a permission model called mandatory access control to protect applications and data provider. By default, applications cannot access sensitive information that is stored on the phone. The default policy also prevents communication between components that are not signed with the same key. Applications that need more than the basic set of features can explicitly request additional permissions via the Android manifest file. There exists a predefined set of permissions, but application developers are free to define new custom permissions.

Android's security model provides four different protection levels of permissions:

- **normal** permissions are low-risk permissions, which give an application access to some features that have minimal potential to cause damage to other applications, to the system, or to the user. During installation, this type of permission is automatically granted to the requesting application without asking the user for acknowledge. An example is the *SET_WALLPAPER* permission which allows an application to set the wallpaper.

- **dangerous** permissions give a requesting application access to an API which provides functionality to access private user data or to potentially dangerous control mechanisms. These permissions need the user's explicit approval when being requested during the application installation. An example is the *INTERNET* permission, which allows an application to open arbitrary network sockets that can be used to make connections and send data to remote servers.

- **signature** This type of permission is only granted by the system, if the requesting application is signed with the same certificate as the app that declared the permission. If the certificates match, the permission is automatically granted without asking for confirmation.

- **signatureOrSystem** This special type of permission is only granted to packages in the Android system image or to packages that are signed with the same certificates. This protection level is generally used by Google or vendors only.

Permissions declared in the manifest file are set at application install time. During runtime, the permissions are enforced by the Reference Monitor. After the installation of an application, there is no way to change or revoke permissions. Permissions can also not be granted dynamically at runtime. The following official statement gives some insight on why Google does not implement a more sophisticated permission management:

> *"Android has no mechanism for granting permissions dynamically (at run-time) because it complicates the user experience to the detriment of security."*[21]

MAC-based permission models are used in other mobile platforms as well, e.g. in the Java ME Platform (formerly known as J2ME). In contrast to Android, the Java ME model is slightly more flexible. It does not only offer a hybrid model of mandatory access control and application vetting, it does also allow the user to select permission options such as *always* and *ask every time* to provide access control at runtime.

### The Internet Permission

The main target of this thesis is the analysis of the Internet access of Android applications. The Internet permission is the main prerequisite for any application that transmits and receives data from servers in the Internet. Among the set of dangerous permissions, the Internet permission is handled in a special way. Network connectivity is exposed via API calls executed by the virtual machine, as well as via native code compiled with Bionic. In order to be able to control both cases, the permission is enforced through the underlying Linux kernel. Therefore, Android developers added a new Unix group *inet* to the system. Each executed application, that has been granted the Internet permission during install time, is added to this group. The Android kernel has been modified to perform the runtime checks by probing whether an application, that is trying to access the Internet, is member of this group.

The Internet permission is also a permission with a huge potential for misuse. Once granted to an application, the app can set up an arbitrary number of connections to remote servers. Data can even be sent and received without the user being aware of it at all.

Many applications require the Internet permission to be able to show advertisements to users, even if the application does not necessarily need connectivity for its core functionality. Other applications require the permission for offering cloud services or for communication with web applications that are hosted on remote servers, which especially makes sense for mobile devices with very limited computing power.

Several research studies did not only confirm that the Internet permission is often a prerequisite for malicious behaviour, they also showed that it is the most requested permission among the set of all permissions. Barrera *et al.* [5] analyzed 1,100 applications for their empirical study which included the top 50 free applications of all 22 categories in the Android Market. They found out that 62% of all applications request the Internet permission. An even more comprehensive analysis was done by the mobile security company SMobile Systems in June 2010 [61]. They collected information about 48,700 applications in the Android Market and reported that even 71% of all apps requested this permission.

This clearly shows that focusing on the Internet permission will give the most benefit, as it targets the security of almost three out of four apps in the Android Market.

# Chapter 3

# Bati - A high-level overview

In general, it should be assumed that applications never transmit sensitive user data without the user being aware of it. However, reality shows, that even non-malicious applications leak user data, for example to advertising servers. Providing network transparency for applications mitigates that problem, as potential leakage becomes apparent to users. This thesis takes on that problem by verifying the Internet access of Android applications. This includes the analysis of network connections that can be established by an application. Besides developing a theoretical model for the analysis, another task includes providing a concrete tool that applies the approach on applications. This chapter presents an abstract high-level description of the analysis tool, *Bati*, to be implemented.

Since the application source code is not available in general, the verification must be performed on the application binary. Android applications are written in Java and compiled to dex bytecode that is executed in the Dalvik virtual machine. Thus, the analysis must be applied on dex bytecode to recover information about the application's behavior. It is part of the application package and can therefore be easily extracted.

The analysis of applications using Bati comprises multiple steps. In a first step, it is to be checked whether the application includes routines to make connections to servers in the Internet at all. This is the necessary prerequisite for an application to exchange data with remote servers.

Once it is verified that the application contains network sinks, it is important to know the destination of outgoing requests. The destination usually comprises an IP address or domain name, a transmission protocol like HTTP or HTTPS, and an optional port specification. Furthermore, the analysis should verify the type of data that is exchanged with the remote server.

## 3.1 System Overview

*Figure 3.1* shows the high-level design of the system to perform the application analysis. Bati, the actual bytecode analyzer, is composed of several modules to increase flexibility and to ease reuse of components, for example in different analysis tools. Bati is the Icelandic word for recovery and was chosen in the style of the Icelandic fishing village Dalvik, which denotes the virtual machine used in Android. The tool implements a static analysis which is described in detail in the following chapters.

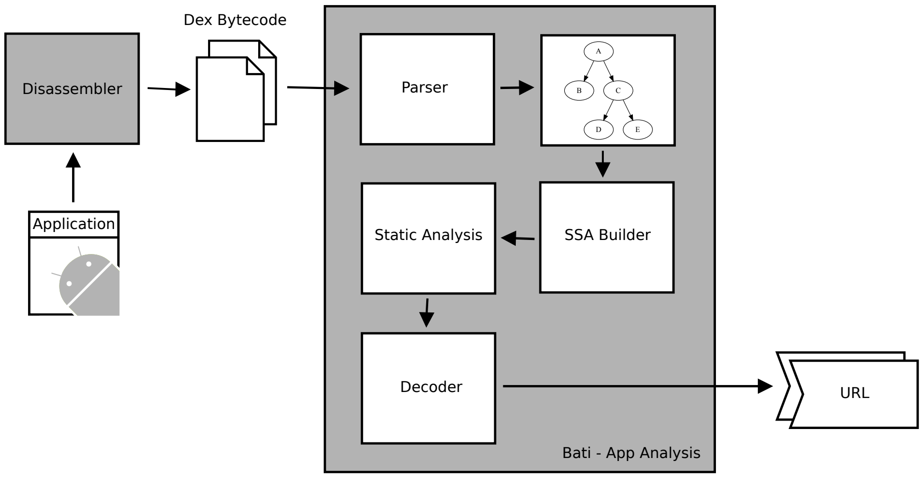


Figure 3.1: High-level view on the analysis tool

Working directly on the compact dex bytecode format is usually infeasible and inconvenient. Therefore, a disassembler translates in a pre-processing step the application bytecode into a more convenient assembly language. The disassembler is not part of the actual implementation. There exist some open-source disassembler for dex bytecode that offer comprehensive functionality and generate high quality assembly language. The output is a set of files containing the assembly language that is used as input for the analysis tool.

Bati comprises several modules, which perform the pre-processing of bytecode, the actual analysis, and the preparation of results that are output to the user.

The task of the first module is to parse the bytecode. Thereby, instructions are dissected into segments and stored for further processing. Within this step, lookup tables and acceleration structures are generated to improve the performance of the analysis. Furthermore, the bytecode is searched for locations at which the application creates connections to the Internet. These locations are considered as data sinks and are the starting point of the actual analysis. Before the verification of the Internet access takes place, the bytecode is transformed into data structures that facilitate the static analysis. In a first step, a control flow graph is generated from the bytecode. A control flow graph is a representation that shows, depending on certain conditions, how information flows through different parts of the application. In a subsequent step, the graph is used to generate a static single assignment form of the dex bytecode. This form is an intermediate representation that is generally used for optimization and validation. It requires modification of the bytecode by renaming registers and injecting custom instructions that provide additional information about data flow.

The static analysis starts at the data sinks found during parsing. Data sinks are method invocations used for initialising and establishing outgoing network connections. In order to open a connection, arguments for destination and payload have to be passed to the invocation. Resolving any data that reaches these network sinks is the main objective of the analysis algorithm. In order to accomplish this task, properties of the static single assignment form are exploited to trace information from predefined data sinks backwards towards data sources. Data sources might be data provider on the mobile phone, user input, or constant values in the bytecode.

All data fragments that are collected during the analysis, are assembled during the decoding step. The results are URLs including a host part that is usually followed by an argument string containing the concrete values that are to be sent to the remote server. Finally, the results are output to the user, who can in turn check whether sensitive information leaves the phone without previous authorization.

## 3.2 Modular Design & Generic Approach

The current system design provides benefit even beyond the actual objective of making application network usage transparent to the user. Its modular design enhances component reuse in other tools. Furthermore, modules can simply be replaced by components implementing a different technique or components offering optimized functionality.

The control flow graph is a common data structure that is used for many data flow analysis techniques. The static single assignment form constitutes the basis for various optimization and validation tasks, for example in the area of compiler construction. It could also form the basis for advanced dex bytecode optimization algorithms.

Backwards flow analysis is one of many techniques for static analysis. By replacing the analysis module with a different algorithm, the system can easily be modified to support other types of data analysis. All modules can thereby access and benefit from data structures created by preceding modules.

Another advantage of the system's static analysis module is that it implements a very generic approach. Tracing data back to its source does not depend on a certain data type. It's only a matter of defining data sinks, which represent the starting points of the analysis. The system's core task is to verify the Internet access of applications, however it can simply be extended to check the sending of SMS or to verify the destination of phone calls. It is only required to define the corresponding data sinks as new entry points for the algorithm.

Chapter 4

# Information Flow Analysis Primer

The basic idea behind data flow analysis is to derive information about the behavior of a program by only examining its static source code. Data flow analysis is a well-known approach to gather information about possible values calculated at various points in an application. It collects information about selected variables defined in the program and the way they are used. Values of variables are monitored while they propagate through parts of the program. Thereby, the control flow graph, a common data structure, is created to provide a high-level data flow representation of the target application. In particular, spots at which the control flow is split, like at conditionals or loops, can then easily be identified.

There exists a great range of approaches to perform data flow analysis. The following sections describe high-level approaches and concepts used in the area of flow analysis algorithms.

## 4.1 Static versus dynamic Program Analysis

The highest abstraction level for program analysis algorithms is the partition into static and dynamic analysis approaches. In dynamic program analysis, the target program is executed on a real or virtual processor. In general, it's very difficult to test all paths through a program, in particular if the source code is not available. For dynamic analysis to be effective, the applications must be executed with a large set of test inputs to achieve a high code coverage. API calls and expressions are executed by the host system during program execution, thus the dynamic analysis can precisely monitor modifications of values as they propagate through the application. Such a real-time analysis always incurs a non-negligible performance overhead that slows down the running application.

The second kind of program analysis is static analysis. In contrast to dynamic analysis, static approaches do not execute the target application. Instead, these algorithms work either on the source code or, if not available, on machine or bytecode. Static analysis is not able to evaluate or monitor third-party code that is not available, e.g. method invocations in private APIs. By exclusively working on source code, this kind of analysis is independent of the architecture and the system. It can be done both online and offline. The offline analysis approach can usually benefit from high-performance systems that accelerate the actual analysis.

In contrast to dynamic analysis, static analysis algorithms can report more detailed and comprehensive results on an application's data flow, because branch structures are often not available in dynamic analysis.

## 4.2 Symbolic Execution

Symbolic execution [39] is a common strategy that is used in program testing and validation. It refers to the analysis of programs by tracking symbolic rather than actual values. This kind of abstract interpretation is used to reason about all inputs that take the same path through a program. It's a common technique in static analysis that is used if no concrete values are provided and/or it is infeasible to test large sets of input values to get a satisfying coverage. Therefore, abstract symbolic values, which represent an entire class of input data, are used for traversing paths.

Symbolic execution is used to reason about a program path-by-path. It does not indicate if multiple inputs take the same path through the program. Compared to the input-by-input analysis done in dynamic program analysis, this approach might lead to superior results, due to improved performance and input coverage. A drawback of this kind of algorithms is the chance of a path explosion. Complex applications include many control structures like if-statements or loops and therefore generate a huge amount of paths that need to be analyzed. Symbolic execution techniques can be applied in both forwards and backwards direction.

Forward symbolic execution is a common approach that is taken for program analysis. Entry points of a program have to be defined and in subsequent steps all paths are analyzed until either predefined sinks are found or until some termination criteria is reached. It is usually unknown which paths contain a sink. Therefore, a forward-flow analysis algorithm needs to traverse all possible paths and filter out any path that does not end in a predefined sink. In complex applications this may lead to large graphs, whose processing is very time-consuming and resource-intensive.

The reverse way is taken in backwards symbolic execution. In a preprocessing step, the program code is searched for any predefined sinks. Once found, these locations are the starting point of the analysis. The control flow graph is then traversed backwards towards data sources like a file input or towards entry points of the program. This approach narrows down the set of paths dramatically, as only a subset of the program needs to analyzed. As compared to the forward analysis approach, each path that is processed, starts at a predefined sink and therefore produces a valid result.

## 4.3 Control Flow Graph

The control flow graph (CFG) is an essential building block for many static analysis techniques, e.g. in compiler optimization. A CFG is a directed graph that represents all paths that might be traversed through an application during its execution. In a CFG, each node represents a basic block, which is a straight-line piece of code or a set of instructions without any jumps or labels. Basic blocks end with a statement that splits the control flow, like *if* or *switch* conditionals, and a new basic block is created at a jump target/label. Most control flow graphs additionally have an unique entry node and an unique exit node.

*Figure 4.1* shows a control flow graph of a simple program. Starting at the conditional at the end of the first basic block, the control flow of the program is split into multiple paths. Depending on the value of $x$, either the left *then*-branch or the *else*-branch on the right is executed.

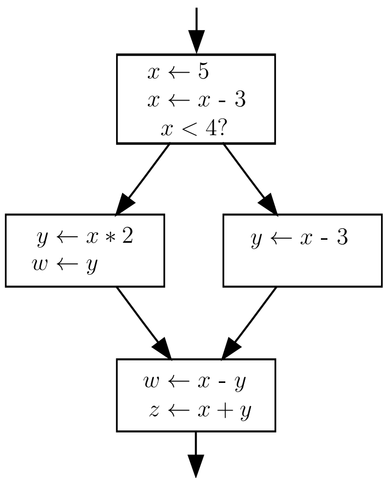


Figure 4.1: An example control flow graph

In the domain of control flow graphs there exist a number of concepts that describe the relationship between nodes. Important concepts include dominator and dominance frontiers which have applications in compilers and for computing the static single assignment form. The concept of dominator was first introduced by Reese T. Prosser in 1959 [54]. A formal definition of dominance is given by:

**Definition 4.1 (Dominance)** *Dom(b): A node n in the CFG dominates b if n lies on every path from the entry node of the CFG to b. Dom(b) contains every node n that dominates b. For x, y $\in$ Dom(b), either x $\in$ Dom(y) or y $\in$ Dom(x). By definition, for any node b, b $\in$ Dom(b). A node n strictly dominates b if n dominates b and n $\neq$ b.*

Often, it is also useful to know the immediate dominator of a node b. Intuitively speaking, this is the node of the dominator set, which is closest to b.

**Definition 4.2 (Immediate Dominator)** *IDom(b): For a node b, the set IDom(b) contains exactly one node, the immediate dominator of b. If n is b's* *immediate dominator, then every node in $\{Dom(b) - b\}$ is also in Dom(n).*

Another concept, especially important for computing the static single assignment form, is called dominance frontiers. Cytron *et al.* define the dominance frontier of a node b as

> *".. the set of all CFG nodes, y, such that b dominates a predecessor of y but does not strictly dominate y."*[19]

The dominance frontier is not only important for finding the exact places for $\phi$-functions in the SSA form (see *Section 4.4*), it has applications in other algorithms as well. It is also used to determine postdominance frontiers, which is an efficient method to compute control dependence.

## 4.4 The Static Single Assignment Form

In many forms of static analysis the choice of the data structure for input data has direct influence on the efficiency and power of the chosen algorithm. A poor choice of the data structure can have many negative implications. It can slow down the analysis dramatically or even inhibit forms of optimization or advanced analysis techniques. The static single assignment (SSA) form has been introduced [3, 56] to provide an improved representation of data and control flow of programs.

The SSA form is kind of an intermediate representation that is often used in compiler design for optimization and validation. A property of this form is that each variable is assigned exactly once. This is achieved by renaming variables. Each new definition of a variable is renamed by appending a sequential numeric index to its original name. Typically, an assignment instruction is considered a new definition. All variable uses are renamed by appending the index of the last definition of that variable. In the final SSA form, each use of a variable has exactly one definition. This means, that *use-def* chains are explicit in the SSA form because each chain only contains a single element. *Figure 4.2a* shows the previous control flow graph example with rewritten variable names.

The second essential step for translating a program into its static single assignment form is to place Phi functions. Just renaming all variables possibly creates ambiguity in join nodes at which control flow from multiple paths is merged. In the last basic block in *Figure 4.2a*, it is obviously not clear which definition of $y$ is used for the following operations, either $y_1$ from the then-branch or $y_2$ from the else-branch.

In order to resolve this ambiguity, the concept of Phi functions is used. Phi functions are special assignment statements that are inserted at the beginning of join nodes. A $\phi$-function has the form V $\leftarrow$ $\phi$(W, X, ..) where V,W,X are variables. The number of operands of the function directly corresponds to the number of immediate predecessor nodes of the node containing the $\phi$-function. As $\phi$-functions are placed in join nodes only, each function has at least two arguments. Thereby, the i-th operand of $\phi$ is associated with the i-th predecessor node. If the control flow reaches a $\phi$-function from the i-th predecessor, variable

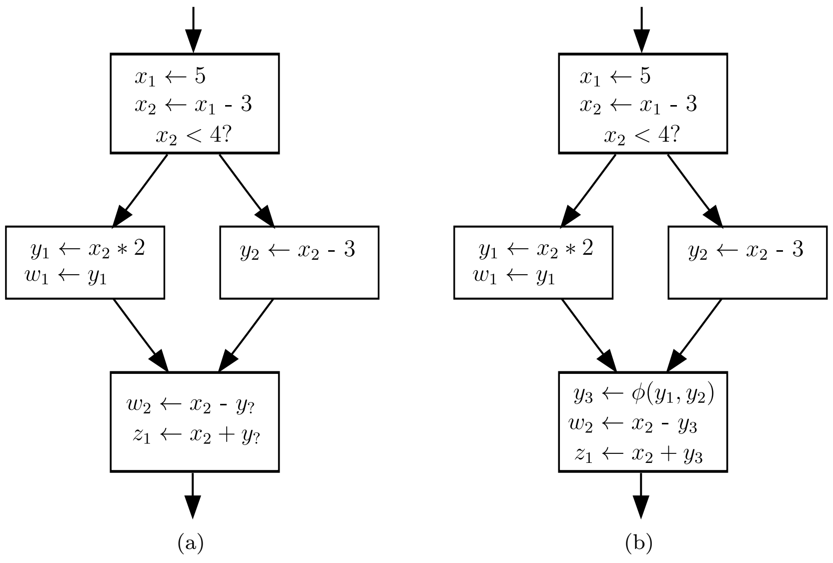


Figure 4.2: SSA form before and after placing the Phi function.

V is assigned the i-th operand. Each execution of a $\phi$ function only uses one of the operands, depending on the control flow.

*Figure 4.2b* shows the program after being fully translated into the SSA form. The inserted $\phi$-function resolves the ambiguity in the last basic block.

### 4.4.1 Creation of the SSA form

The static single assignment form is a powerful and attractive data structure in data flow analysis. It usually facilitates and accelerates analysis algorithms. However, the creation of the SSA form is difficult and tends to be inefficient. Over the years there has been a lot of research in making the generation of the SSA form more efficient and easier to implement. But besides all optimizations, each algorithm to translate an input into its SSA form includes the following two essential steps:

1. Placement of $\phi$-functions
2. Renaming of variables

A SSA algorithm starts with the placement of phi functions for merging the information flow of multiple paths. There exist various approaches to place $\phi$-functions at join nodes. Many algorithms use a control flow graph and compute the dominance frontier set to determine all target spots. This concept is based on the work of Cytron *et al.* [18, 19]. Computing dominance on the flow graph has been the bottleneck in first implementations. By using approved and more efficient algorithms to compute dominance information [40, 37, 17], the performance of the SSA generation could be increased significantly. Thereby, the concept of the SSA form has become a default technique in static program analysis. There also exist other techniques for placing $\phi$-functions, like approaches based on DJ graphs [60, 20] and merge sets [8].

The second step targets the renaming of variables such that each variable is defined exactly once. Typically, an assignment to a variable is considered a new definition. The renaming procedure is pretty straightforward and can be done in linear time. Only the current index per variable needs to be recorded.

# Chapter 5

# Information Flow Analysis on Android

The current state of the Internet permission allows an application to create an arbitrary number of connections to arbitrary remote servers. There is no limitation, neither in the number of connections, in the destination of the connections, nor in the kind of data that can be sent via connections. Unless the network traffic is monitored, apps can also establish connections in the background without the user being able to notice it.

The Internet permission is a necessary prerequisite for many malicious apps that try to send private user data to servers or to interfere with security critical applications like online-banking apps. However, there are also many applications that use this permission for legitimate reasons.

Unfortunately, there is no way for a user to decide whether a program uses the Internet permission correctly or not. Even at runtime, it is often not obvious which data leaves the phone and where it is sent. There is currently no transparency for the Internet usage of Android apps.

The goal of this thesis is to provide a tool to verify the Internet access of Android applications. Given an application, the tool's task is to retrieve information about connections that are established to servers in the Internet. In particular, the destination of the connections, that are host names and IP addresses, are from great interest. Finally, the user is interested in the kind of data that is sent to remote servers. Such an analysis tool will help users to detect malicious apps easier and faster, since the entire network communication of an application is revealed.

## 5.1 How to analyze Android Applications?

The first and most important step towards the verification tool is to choose an appropriate analysis technique. A possible approach could be based on dynamic program analysis. This includes executing the app on a real or virtual device.

One kind of dynamic analysis is used in TaintDroid [25]. Data originating from sensitive data sources are flagged with a taint marker to follow its propagation through the application (see also *Section 8.1*). The advantage of this kind of dynamic runtime analysis is that data which leaves the phone, for example via network, can be monitored precisely.

API calls and expressions do not have to be evaluated, the result is just to be observed and reported. But this convenience has also some drawbacks. Dynamic analysis usually requires a modification of the Android SDK to implement the monitoring mechanism. Furthermore, real-time analysis always incurs a non-negligible performance overhead, especially on mobile devices with very limited resources. TaintDroid only tracks explicit data flows to minimize this performance overhead.

Static analysis algorithms are directly applied on the application's bytecode. Algorithms that work directly on source or bytecode are not capable of analyzing libraries for which no code is available. However, there are also several advantages as opposed to dynamic approaches. By using symbolic execution and analyzing path information obtained by the control flow graph, static analysis algorithms can give detailed and comprehensive information about the data flow through a program.

Static analysis can also be done offline on high performance computers to accelerate the analysis. Executing Android applications on a x86 system is currently only possible by using the simulator that is shipped with the official SDK. This virtual device is quite slow and not meant to be used for heavy computation tasks. Being able to apply an analysis algorithm offline offers the possibility to even analyze a large number of applications in a short amount of time.

This kind of analysis also does not incur any runtime performance overhead. When used on a real device, the analysis just runs once without using resources during the application's runtime. Bytecode analysis also does not require modifications to the system image[1]. There is no need to implement functionality into the core system.

Dynamic analysis always suffers from the problem that the program needs to be executed. If an application is executed on a real device, there is always a chance that malicious actions take place. These actions can be monitored and reported, but not prevented in the first place. On the other side, static analysis takes place during installation-time of an application. If some malicious behaviour is detected, the user can still abort the installation without having experienced any negative effects.

Taking all these aspects into consideration, static bytecode analysis seems to be the appropriate choice to achieve the objectives of this thesis.

[1]Modifications are not necessary for the actual analysis. However, the Android security mechanism does not allow applications to access data of other apps. Therefore, a mechanism to access the bytecode of other applications must be implemented to run the analysis on an Android device.

### 5.1.1 Definition of Sinks

The static analysis algorithm directly operates on the dex bytecode. Before the analysis is executed, the starting points have to be specified. In the context of this thesis, these are data sinks at which data can leave the phone towards the Internet. In bytecode the sinks are usually method invocations of certain classes.

The Android/Java API contains several classes that can be used to create connections to remote servers. These classes also provide functionality to exchange data. The classes which are considered as a sink are:

- java.net.URI
- java.net.URL
- org.apache.http.client.methods.HttpGet
- org.apache.http.client.methods.HttpPost

URI and URL are both generic classes which are used to specify an Uniform Resource Identifier/Locator including an optional argument string to be sent. Instances of these classes can then be used to create HTTP, HTTPS, or FTP connections. The Apache classes can be used to create HTTP GET and POST requests as well. Instances of the Apache classes can not only be created using an existing URI instance, but also via a textual representation of a URI. Thus, these classes must also be considered a data sink to capture all cases.

After the specification of sinks is fixed, a technique is required to find these sinks in bytecode. Given a disassembled Android application and the predefined set of sinks, it is to be checked whether or not the application contains any of these sinks. Furthermore, their exact location needs to stored, as they are the starting point of the analysis.

The dx compiler translates any method call into an *invoke* bytecode instruction. There exist several types of *invoke* instructions to differentiate for example between static methods or super class methods. *Bytecode Listing 5.1* shows the dex bytecode that is generated for a new URL object that is initialized with the domain *http://www.foo.com* .

```
new-instance v0, Ljava/net/URL;
const-string v1, "http://www.foo.com"
invoke-direct {v0, v1}, Ljava/net/URL;-><init>(Ljava/lang/String;)V
```

Bytecode Listing 5.1: Dex bytecode for a URL instance

The dex bytecode instructions *new-instance* and *const-string* define new objects. The instruction of interest is the *invoke-direct* instruction. *invoke-direct* is used to invoke a non-static, non-overridable method–in this case the constructor of the URL class.

The invoke instruction format slightly differs from other instructions that have been explained in *Section 2.2.4*. The first argument is a *set* of registers. In non-static method calls, the first register of this set is the object instance on which the method is invoked. All subsequent registers are method argument registers. The URL constructor has a single argument of type String. It is passed via register *v1* that refers to the constant string *http://www.foo.com* .

### 5.1.2 Backwards Symbolic Execution

In a first step, all classes that can be used to make outgoing connections have been identified. Furthermore, it has been described how these sinks can be located in dex bytecode. Finally, an approach is required to recover all information that reaches the sinks. This does not only include the domain or host part of the URL, but also data that is sent to this destination. Symbolic execution is a good choice, because no concrete test inputs have to be generated. This approach uses symbolic values to traverse the paths through the program. It can be applied both in forward and backward direction.

In symbolic execution an entry point needs to be specified indicating the starting point of the analysis. In forward symbolic execution this is typically the main function of a program. However, Android applications do not have a single entry point. Instead, an application can have arbitrary entry points. Any *onCreate* constructor of an Activity, that can be displayed to the user, is considered an entry point. Furthermore, forward analysis potentially traverses the whole control flow graph of a program, since the paths that contain sinks are not known in advance. This slows down the analysis significantly, even more if the analysis is done on real devices whose ARM processors have very limited performance.

Backwards symbolic execution can exploit the fact that the location of sinks has already been determined. Thus, the control flow graph can be traversed backwards starting at the predefined sinks. This narrows down the search tree dramatically, since only a subset of the CFG needs to be analyzed. In many cases, backwards analysis will also output fragments of the final result faster than the forward approach. Usually, parts of the URL are created close to the sink, which is close to the connection that is used for data transmission. Thus, information about the destination can often be output quite fast, whereas parts of the argument string usually origin at various parts of the application and therefore require advanced analysis.

## 5.2 Building the Control Flow Graph

As explained in *Chapter 4*, an appropriate data structure is essential for any data flow analysis approach. The control flow graph is a well approved data structure for representing the control flow through a program.

For the actual analysis, the control flow graph is created on a per method level. This is also the granularity provided in dex bytecode. The resulting CFGs are relatively small in size (usually less than 100 basic blocks) and have therefore manageable complexity. The control flow graph is an intermediate representation that is used in subsequent steps as basis for the static single assignment form and for the actual backwards symbolic execution.

*Figure 5.1* depicts a syntax diagram that describes how a control flow graph is created from dex bytecode. The notation for the syntax diagram can be found in *Appendix A*. A control flow graph is composed of at least one *basic block*. An empty block is the simplest form of a basic block. A block may also start with exactly one label, which is usually the target of a jump instruction. An arbitrary number of *normal statements* can follow. Thereby, any instruction other than an *end statement* or a label is considered as a normal statement.

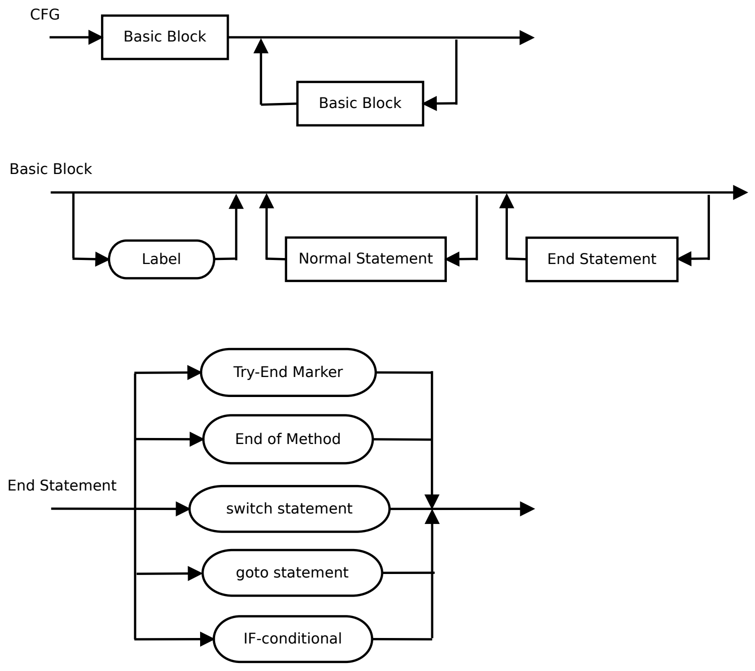


Figure 5.1: Syntax tree for the method control flow graph

There are several statements that end a basic block. A try-end marker indicates the end of the *try* block in a try-catch statement. When the end of a method is reached, the current basic block is considered to be complete, because the CFG is built on a per method basis. This is also the case if an exit instruction is encountered. These are instructions that leave the current method; either a *return* or a *throw* statement. Furthermore, *goto* and *switch* statements as well as *if*-conditionals end the current basic block. Usually, a single statement ends the current basic block. However, there are rare cases in which a throw statement is followed by a try-end marker. To cover these cases as well, the grammar has to allow more than one end statement.

### 5.2.1 Connecting the Basic Blocks

After having assembled a set of basic blocks from dex bytecode, the blocks have to be connected to create the control flow graph. There are multiple ways to link nodes. *Figure 5.2* depicts three basic cases. Nodes without special statements as last instruction are linked with their respective next node. Basic blocks that end with a *goto* jump instruction are linked with the block marked with the corresponding target label. If-conditionals have always a true and a false branch, thus if the current basic block ends with an if-conditional, it is linked to the associated *then* and *else* block.

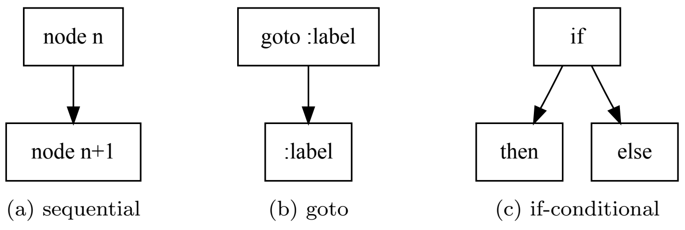


Figure 5.2: Basic cases for connecting nodes

Basic blocks that contain a switch statement have to be handled differently. Switch statements are statements that select among pieces of code–the *cases*–to be executed next, based on a given integral value. If no integral value matches, a default statement is executed. Therefore, the node ending with the switch statement has to be linked to all nodes representing a case, including the default case.

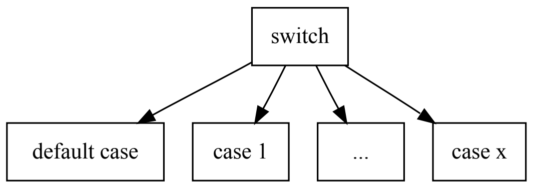


Figure 5.3: Connecting basic blocks in a switch statement

Applications in Android are written in Java. The Java programming language has a concept called *try-catch* that is used for error handling. A piece of code, the try-block, can be marked as *guarded area.* Whenever an error is thrown within this area, a previously defined catch-block is executed. A catch-block is a custom error handler for specific or generic errors, written by the developer. *Figure 5.4a* shows the control flow graph for a try-catch block. The try-node is linked to all error-handling catch-nodes, as well as to its successor node in case that no error occurs.

There also exists an extended version of try-catch, the *try-catch-finally* statement. The developer can include an additional finally-block, whose statements are *always* executed after try and catch-blocks. In programming languages without garbage collection and automatic destructor calls, the finally extension is important to guarantee the release of memory, regardless of what happens in the try-catch code. In Java, developers do not have to take care of memory, but the finally statement allows to do cleanup of parts other than memory, e.g. closing network connections or file handles.

Thus, the control flow graph needs to be created slightly different in presence of a finally-block (see *Figure 5.4b*). Instead of linking the try-block to its successor, it is linked to the finally-block. In addition, all catch-blocks are linked to the finally-block as well, because its execution is guaranteed under any circumstances.

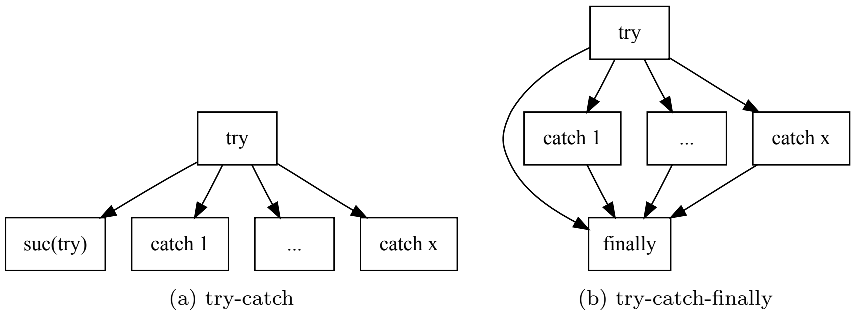


Figure 5.4: Connecting basic blocks in try-catch statements

In a final step, a unique end node is added to the graph. Any basic block that includes a statement to leave the current method, that is a *return* or a *throw* statement, is linked to that unique end node. This end node facilitates the backwards symbolic execution of custom methods. When a method is resolved backwards, the algorithm needs to look for any node with a return statement. Instead of storing the node indices, all return statements can be found in the predecessor nodes of the unique end node.

## 5.3 The SSA Form on Dex Bytecode

Given a set of predefined sinks as described in *Section 5.1.1*, the target argument registers are to be resolved during analysis. An example is the register associated with the string that is used in the URL class constructor. Starting from the location of the sink, the algorithm's task is to find the last instruction that assigned a new value to that target register.

The control flow graph provides information about the various paths through the application. However, finding the last assignment statement for a target register isn't an easy task. In high-level programming languages, variables are defined and used for a specific instruction or operation. In dex bytecode there are no variables. Instead, any operation is performed on registers. Thereby, registers are not chosen individually. The dx compiler generates the bytecode automatically. Register reuse is common within larger methods to keep the number of registers in use as small as possible. Reuse is even more encouraged through the fact that registers can hold values of arbitrary type.

Usually, it is not sufficient to stop when the last assignment of the target register is found. The values that reach a data sink could also be assembled by a series of computations. The analysis algorithm must know which register definitions belong to the current computation or if a certain definition is used in a totally different context.

Facing these problems, it becomes apparent that the CFG data structure is not sufficient for backwards symbolic execution. In contrast, the static single assignment form (see *Section 4.4*) provides this extra information about register definitions and uses. When resolving a register in the SSA form, there is exactly one definition and no register ambiguity. No context is required for the analysis algorithm to verify whether a certain register definition is associated with the currently resolved expression or with some completely different expression.

*Bytecode Listing 5.2* shows two dex bytecode snippets that are executed to create a new URL instance. The first version in *Bytecode Listing 5.2a* depicts default bytecode, as generated by the dx compiler. The string that is used in the constructor is defined multiple times in the code before, but the value is always stored in register *v1*. Without additional verification, it is not clear which definition is actually used for the URL instantiation.

The second code snippet in *Bytecode Listing 5.2b* shows the same bytecode after being translated into the SSA form. Each new definition of register *v1* is renamed by appending a unique numeric index. The use of *v1* in the constructor is renamed accordingly. The algorithm is now able to determine the correct value explicitly and efficiently.

```
const-string v1, "www.foo.com"
[...]
const-string v1, "www.bar.com"
invoke-direct {v0, v1}, Ljava/net/URL;-><init>(Ljava/lang/String;)V
```

(a) Normal dex bytecode

```
const-string v1_1, "www.foo.com"
[...]
const-string v1_2, "www.bar.com"
invoke-direct {v0_1, v1_2}, Ljava/net/URL;-><init>(Ljava/lang/String;)V
```

(b) Dex bytecode in SSA form

Bytecode Listing 5.2: Dex bytecode with/-out SSA form

### 5.3.1 Translating Dex Bytecode into SSA Form

In order to translate dex bytecode into a SSA form, *phi* functions need to be placed in join nodes of the CFG. Furthermore, all registers have to be renamed. The exact locations at which phi functions are required, can be determined by computing the dominance frontier for each node in the CFG. If a new variable or register is defined in a node N, then only that definition or redefinitions will reach every node which is dominated by N. Definitions of the same variable, originating from other paths, must only be accounted, if the dominated nodes are left and the dominance frontier is entered.

To integrate the theoretical concept of phi functions directly into dex bytecode, a new phi bytecode instruction had to be devised. A phi function normally has the form $v_A \leftarrow \phi(v_B, v_C, ..)$ where $v_A$, $v_B$, $v_C$ are registers. Transformed into a dex bytecode instruction, the phi function is defined as follows: *phi* $v_A$*,* $v_B$*,* $v_C$*,* .. . Similar to other instructions, the first register is the destination register.

It is followed by a number of argument registers, which directly corresponds to the number of immediate predecessor nodes. A phi function for a register *v2*, placed in a node with two predecessor nodes, looks as follows: *phi v2, v2, v2*. The renaming of the argument registers is done in a dedicated renaming step.

After all $\phi$-functions are placed, the renaming step is performed to complete the SSA form. New definitions of a register, that is if the register is assigned a new value, are renamed by appending a sequential numeric index that is recorded per register. Register uses are renamed by appending the index of the last definition of that register. Registers within a phi instruction are renamed the same way. Taking up the previous example, the phi instruction *phi v2, v2, v2* could be renamed to *phi v2_5, v2_3, v2_4*. This means that the last definition of register *v2* is *v2_3* in the path starting with the first predecessor and *v2_4* in the path starting with the second predecessor. Note, that the phi instruction itself is a new definition of the target register and therefore the destination register gets a new unique index.

In general, the SSA form is applied on high-level languages that use variables. In these languages, arguments of the phi function can always be renamed properly, because variables have to be defined before they can be used. In register-based bytecode this is different. Registers do not have to be defined, they are just used. Additionally, temporary values sometimes need to be stored in extra registers to perform certain operations. In more complex constructs, it can happen that registers are only used in some of the paths that end in a common join node. In a nested if-conditional as depicted in *Figure 5.5*, some target registers might be present in path if-2 because of some extra operations that require temporary registers. The same register might not necessarily be used in the then-1 branch.

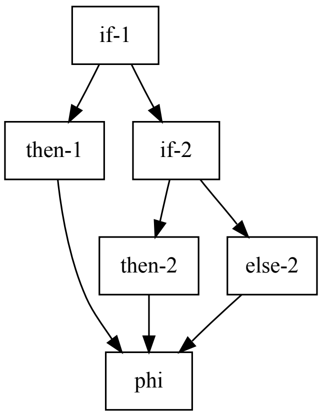


Figure 5.5: CFG of a nested if-conditional

In this case, the resulting phi function would look as follows: *phi* $v_A$*_i*, $v_A$*_?*, $v_A$*_j*, $v_A$*_k*. The renamed registers $v_A$*_j* and $v_A$*_k* directly correspond to the predecessors then-2 and else-2, respectively. But since register $v_A$ is not defined in then-1 or in any of its predecessors, it cannot be rewritten. Such cases are then marked with a capital $X$ to indicate that this register does not have a definition in the respective path. The resulting $\phi$-function is rewritten to *phi* $v_A$*_i*, *X*, $v_A$*_j*, $v_A$*_k*. An algorithm reaching such a phi instruction will not enter predecessor nodes marked with $X$ when resolving register $v_A$.

In order to complete the renaming step, a last bytecode modification needs to be done. In invoke instructions the first argument is a set of registers which includes the method arguments. In case of a non-static method invocation, it additionally includes the object instance register. The number of registers is important since the dex bytecode specification defines a maximum of five registers in this set.

If more than five registers are present, for example if the method has many arguments, a special set of invoke instructions is used. These instructions use a range representation for registers and are labeled with a trailing *range*. The range representation is defined as $\{v_A \text{ .. } v_B\}$, where $v_A$ and $v_B$ are the start and end register, respectively.

When ranges are processed, only the start and end registers are rewritten. During analysis, the rewritten names of the registers within the range are not available. The basic name of the registers is known, however, this information is not sufficient for the analysis algorithm to proceed. Therefore, range notations are expanded in a pre-processing step. Instead of renaming a range $\{v_A \text{ .. } v_F\}$ directly, it is first expanded to $\{v_A,\ v_B,\ v_C,\ v_D,\ v_E,\ v_F\}$ and then renamed in a subsequent step. This way, all information necessary to fully resolve all argument registers is available during analysis. Since the resulting bytecode is not executed by the Dalvik VM and is used for the analysis only, it doesn't matter if a set of argument registers contains more than five elements.

## 5.4 Backwards Symbolic Execution

The actual analysis starts at predefined sinks that have been previously located in the bytecode. The sinks are usually invoke instructions that have one or more argument registers. In case of the URL sink, the target register is the one associated with the string that is used in the constructor. The analysis algorithm starts with the rewritten target registers and searches the bytecode backwards for the corresponding definitions.

When the instruction with the matching destination register is found, the algorithm must make a decision on how to proceed with the analysis. This strongly depends on the type of instruction. If an instruction is found that assigns a constant value, the algorithm stops. Does the instruction again contain some source registers, the static analysis has to proceed and the corresponding values have to be resolved as well.

*Table 5.1* shows the resolver semantics that is used in the backwards symbolic execution algorithm. It includes all opcodes that can occur during register resolving. A trailing *op* denotes an entire family of instructions, for example *const-op* represents the set of all instructions that define a constant value. The resolver semantics describes how the analysis proceeds depending on the current instruction opcode. Opcodes can thereby be partitioned into direct and indirect opcodes. During backwards symbolic execution, only instructions labeled as *direct* opcodes are found during register resolving. This is due to the fact that certain instructions, which for example modify data structures or class fields, are never connected directly with a data sink. Thus, they will never appear in the search for a register definition. However, it turned out that the default symbolic execution approach lacks important information in certain situations, like the reconstruction of array data. In these cases, the bytecode is searched for *non-direct* opcodes explicitly. This enables a more precise and comprehensive analysis since these instructions provide additional information about objects and class fields.

| Op Format | Op Semantics | Direct | Resolver Semantics |
|---|---|---|---|
| const-op $v_A$, C | $v_A \leftarrow C$ | ✓ | $\varnothing$ |
| move-op $v_A$, $v_B$ | $v_A \leftarrow v_B$ | ✓ | $\rho(v_B)$ |
| move-result-op $v_A$ | $v_A \leftarrow R$ | ✓ | resolve(curInstr-1) |
| return-op $v_A$ | $R \leftarrow v_A$ | ✓ | $\rho(v_A)$ |
| unary-op $v_A$, $v_B$ | $v_A \leftarrow \otimes v_B$ | ✓ | $\rho(v_B)$ |
| binary-op $v_A$, $v_B$, $v_C$ | $v_A \leftarrow v_B \otimes v_C$ | ✓ | $\rho(v_B), \rho(v_C)$ |
| binary-op $v_A$, $v_B$ | $v_A \leftarrow v_A \otimes v_B$ | ✓ | $\rho(v_A\text{-}(i\text{-}1)), \rho(v_B)$ |
| binary-op $v_A$, $v_B$, C | $v_A \leftarrow v_B \otimes C$ | ✓ | $\rho(v_B)$ |
| phi-op $v_A$, $v_B$, $v_C$ [..] | $v_A \leftarrow \phi(v_B, v_C, ..)$ | ✓ | $\rho(v_B), \rho(v_C)[..]$ |
| sget-op $v_A$, $f_B$ | $v_A \leftarrow f_B$ | ✓ | $\mu(f_B)$ |
| sput-op $v_A$, $f_B$ | $f_B \leftarrow v_A$ | – | $\rho(v_A)$ |
| iget-op $v_A$, $v_B$, $f_C$ | $v_A \leftarrow v_B.f_C$ | ✓ | $\mu(v_B.f_C)$ |
| iput-op $v_A$, $v_B$, $f_C$ | $v_B.f_C \leftarrow v_A$ | – | $\rho(v_A)$ |
| new-array $v_A$, $v_B$, $f_C$ | $(f_C)[\,]\ v_B$ | – | $\rho(v_B)$ |
| aget-op $v_A$, $v_B$, $v_C$ | $v_A \leftarrow v_B[v_C]$ | ✓ | $\rho(v_C), \mu(v_B)$ |
| aput-op $v_A$, $v_B$, $v_C$ | $v_B[v_C] \leftarrow v_A$ | – | $\rho(v_A)$ |
| filled-new-array $\{v_B, v_C, ..\}$, $f_A$ | $R \leftarrow (f_A)[v_B, v_C, ..]$ | – | $\rho(v_B), \rho(v_C)[..]$ |
| new-instance $v_A$, $f_B$ | $(f_B)\ v_A$ | – | $\varnothing$ |
| invoke-op $\{v_B, v_C, ..\}, \mathcal{F}_A$ | $\mathcal{F}_A(v_B, v_C, ..)$ | – | $\mu(v_B), \mu(v_C), ..$ |
| invoke-op $\{v_D, v_E, ..\}, \mathcal{F}_C$ | $\mathcal{F}_C(v_D, v_E, ..)$ | – | $\mu(v_D), \mu(v_E), .., \mu(\mathcal{F}_C)$ |

Table 5.1: Resolver semantics in backwards symbolic execution

The default approach for resolving registers is to search for its corresponding definition and is indicated by $\rho()$ in the resolver semantics. The resolution of a register starts at the current instruction. The bytecode is processed backwards until a new definition of the register is found. In this context, definition means that the register is assigned a new value. This backwards processing continues until a constant value is found. Field references and data types marked with $\mu()$ have to be handled specially, since the default approach cannot be applied or only recovers little information. The resolution of method arguments strongly depends on the data types.

During analysis, it is possible that the beginning of a method is reached and the register is not yet fully resolved. This happens if the register belongs to a method argument. In this case, the application is searched for invocations of this method. When an invocation is found, the algorithm locates the target argument register and proceeds with the analysis at that point. If multiple invocations are found, the algorithm analyzes them consecutively. This also implies that more than one result is output for the current sink.

64-bit values must be handled explicitly. In the Dalvik virtual machine the register size is 32 bit. Adjacent register pairs are used for 64-bit values of type double or long. There are dedicated opcodes to handle 64-bit values, marked with a trailing *wide*. Although these opcodes handle 64-bit values, their instruction format does not include additional registers. Instead, opcodes only specify the first of both registers $v_A$, the second adjacent register $v_{A+1}$ is used implicitly.

Fortunately, in symbolic execution the second register does not matter. If this register would be resolved, either no result would be found or an incorrect one, in case the register has been used before for something else. Therefore, it is sufficient to just resolve the first register that is specified in the instruction. The resolver semantics does not change in these cases. However, if 64-bit values are used in method arguments, both registers are explicitly specified. In this case, only the first register must be resolved, whereas the second one has to be omitted. In order to determine whether 64-bit values are used within a method invocation, the method header must be parsed. It includes the full class name of arguments types and thus wide values can easily be identified.

The easiest case handled by the resolver semantics is when an instruction is found that defines a constant value. No further resolving is necessary and the algorithm stops. Besides the *const-op* and the *move-result-op* instructions, all other instructions include at least one register that must be resolved.

The resolving algorithm starts at the current instruction and continues until a constant value is found. An exception are the *move-result* opcodes. Method invocations via *invoke* opcodes and the *filled-new-array* opcode always return a result that must be stored in a register. Thus, any of these opcodes is immediately followed by a *move-result* opcode which performs this action. In backwards symbolic execution, the move-result instruction is always reached first. However, resolving the single argument register does not lead to the corresponding invoke instruction. There is no explicit connection between these two bytecode instructions. In order to continue with the correct method invocation, the analysis has to process the preceding instruction when a *move-result* opcode is reached.

Binary opcodes that perform an operation on two registers have an instruction format that needs to be handled specially. In these two-address opcodes, the first register is both destination and source register. Since this register is assigned a new value, it is considered a definition and gets a unique index during renaming. As this register is also considered a source register, it has to be resolved during analysis. But searching for this definition won't produce a result. The search has to be modified such that the register is found whose data is propagated to this instruction. This is achieved by searching for the register that has the same base name but an index decremented by one. Instead of searching for a register $v_A\_i$, the algorithm proceeds by searching for register $v_A\_(i-1)$. This way, the correct definition for the first operand is found.

The custom phi instruction requires special handling as well. Phi statements are always placed at the very beginning of join nodes in the CFG. They help to keep track of register definitions across multiple paths that originate at constructs like switch statements or if-conditionals. In phi statements, all source registers are resolved. In contrast to other instructions, a phi source register is only resolved within the path that starts with its associated predecessor node. Searching in other paths usually produces wrong results. Arguments of the $\phi$-

function marked with $X$ are not processed and considered a dead-end. When a $\phi$-instruction with multiple argument registers is reached, the algorithm resolves every register in its associated path. This also implies that for each analyzed path a distinct result is output. Additionally, the results are very likely to differ because different paths represent different control flow.

### 5.4.1 Resolving Class Fields

Values of static class fields are retrieved via *sget*-opcodes. Thereby, $f_B$ denotes the field reference to be accessed. Unfortunately, this reference is not a register that can be resolved by searching for its definition. The reference only provides information about the class and field name. In order to recover the concrete value at the time of the access, additional analysis is necessary. Therefore, the algorithm searches for all instructions that modify this specific field.

Static class fields can be modified via *sput*-opcodes. These opcodes are never found directly in backwards symbolic execution. When a static field access is reached via a sget-instruction, the algorithm searches for matching field modification opcodes explicitly. The algorithm starts by searching for the last sput-instruction that modified the associated class field, beginning at the current instruction. This search is path-sensitive to cover cases in which the field is updated differently in different paths of the CFG. If the header of the current method is reached and no matching sput-instruction was found, assignments to this class field have to be searched outside the method.

The complete application bytecode is searched for matching instructions, since static fields are not bound to a specific object instance. All sput-instructions found in this step are then resolved regularly. Multiple matching instructions are resolved consecutively. This again implies that several results are output for the current sink. Theoretically, a static class field reference can also be passed as method argument. In these rare cases, the algorithm searches for matching method invocations and proceeds with the class field recovery as described above.

Non-static class fields are accessed via *iget*-opcodes. In contrast to a static field access, iget-instructions additionally include the instance object whose field is to be accessed. However, applying the default backwards symbolic execution approach does not result in a concrete field value. Similar as for static fields, only information about class and field name can be retrieved. Thus, additional effort is required to recover the field value at the time of the access. Beginning from the current instruction, the target is the last *iput*-instruction that modifies the current instance field. The search is done path-sensitive and stops at the method border, if no match was found before.

In this case, there are two possibilities on how to proceed. If the object is the *this* reference, that is the current class instance, it is obvious that the current method is a class method. The application is then searched for any invocation of this class method and the algorithm continues with the analysis from the location of the invocation. The actual object register can then be extracted from the found invoke instruction. It is used to continue the search for class field modifying instructions as described before.

If the object reference is passed as method argument, the analysis has to proceed differently. The analysis continues with the search for all invocations of that method. In this case, the object register is part of the method argument list. Starting at the found method invocation, the search for matching iput-instructions then continues normally.

### 5.4.2 Resolving Arrays

Array data can be accessed via *aget*-instructions. The array element is stored in the destination register. The source registers include the array register and the index register.

*Bytecode Listing 5.3* shows a simple example containing an array definition and an array access. In the first lines a new integer array of size three is defined. Afterwards, the first two elements are set to 3 and 5, respectively. Finally, the first element from the array is retrieved.

```
const/4 v2, 0x3     // new integer array of size 3
new-array v0, v2, [I

const/4 v4, 0x0     // array[0] = 3
aput v2, v0, v4

const/4 v2, 0x1     // array[1] = 5
const/4 v3, 0x5
aput v3, v0, v2

aget v1, v0, v4     // v1 = array[0]
```

Bytecode Listing 5.3: Array definition and modification

The index register is resolved normally. Afterwards, the index of the element to be accessed is known. Resolving the array register directly leads to the definition of the array, which is the *new-array* instruction. It provides information about the array type and a reference of the size register. After resolving this information, the algorithm knows that there is an integer array with three elements. Thus, all meta information is available, but the actual elements of the array are still unknown. It is not possible to reconstruct the array data once the definition is reached.

In order to provide the actual element values, the aget-instruction has to be resolved specially. The algorithm tries to reconstruct the array data at the time of the access. Starting at the aget-instruction, the algorithm records any modification on the target array. The bytecode is processed backwards until the array definition is reached. Only the most recent modification of an array element is used to reconstruct the array. Any previous values are out of date at the time of the access. The index register is resolved normally to determine the target element.

By using the reconstructed array, it is possible to evaluate the array access. Usually, the element index is resolved first. Thus, it would be sufficient to reconstruct only the target element and not the whole array. However, the index is, in general, not necessarily reducible to a numeric value. In symbolic execution the index could also be an unresolvable method invocation that returns an integer value. In this case the concrete element index is unknown and the complete array data must be reconstructed to provide detailed information.

In *Bytecode Listing 5.3* the index register *v4* is resolved to 0. The array reconstruction records the value 3 and 5 for the first two elements. Once the definition is reached the array type and size is also available and thus the reconstructed array is [3,5,X]. The *X* that denotes the third element indicates an unknown or uninitialized value.

Arrays can also be defined and pre-filled by use of a single bytecode instruction. The *filled-new-array* opcode specifies an array type and a set of argument registers which are used to initialize the array elements. The argument registers can be resolved normally without any special handling. Note, that this opcode is only reached indirectly via the *move-result* instruction that immediately follows.

### 5.4.3 Resolving Method Invocations

Methods or functions are important programming constructs that are nearly used in any application to separate code into smaller units that perform a specific task. In symbolic execution, methods have to be classified into open-source and closed-source methods. All methods whose bytecode is available in the application are classified as open-source or custom methods. This does not only include all methods implemented by the app developer, but also third-party code. A prominent example is the AdMob SDK, that is used to display advertisements. All other methods are closed-source or API methods. These methods are not necessarily closed-source in general, but from the view of the application, their bytecode is not available. An example is the official Android API which is not closed-source. However, an application does not include the source-code from all APIs that are used within the app.

Static analysis algorithms are not capable of analyzing closed-source methods in depth, since their bytecode is not available. But the arguments used for the method invocation can be resolved. As explained in *Section 5.1.1*, function calls use an *invoke* bytecode instruction. The first argument of this kind of instruction is a set of registers. In case of non-static method invocations, it includes the object register on which the method is invoked. Additionally, the set includes registers for all arguments that are passed to this function call. The register set can be resolved normally, that is by searching for the respective definitions. But comprehensive information would then not be recovered for certain argument types. Thus, they need to be checked and processed accordingly.

If a register is associated with an array argument, the algorithm handles it the same way like in *aget*-instructions. Direct resolving would only discover meta information about the array. Putting some additional effort into reconstructing the array content, results in the concrete element values at the time of the function call.

In non-static method invocations, the first register in the argument set is the object register, that refers to the object on which the method is invoked. Resolving this register directly leads to the *new-instance* instruction, which is the object definition. This opcode does not provide any useful information, since the object type can be extracted from the method header as well. The instruction is also a dead-end, as it does not contain any more registers to resolve. Thus, not even the object initialization or constructor call can be found this way.

But the algorithm should not only provide information about the constructor. The analysis should go beyond the initializer, as usually the object's call history is of great interest. A call history is a simple, chronologically ordered list of method invocations executed on a target object, starting at the constructor call and ending at the currently processed invocation. This construct gives valuable context information about an object that reaches a data sink. In general, the interesting arguments in data sinks are strings or numeric values. If an object is propagated to a sink, it is converted to a string value, for example through the *toString* method. This is also the first and only invocation that is encountered during normal analysis. Without a call history, only this piece of information is output to the user.

*Example 5.4* shows an bytecode excerpt in which a new class instance is created. The class object is initialized and in subsequent instructions some class methods are invoked. Finally, a string representation of the instance is retrieved and used to initialize a StringBuilder instance[2].

```
new-instance v1, $Class;
...
invoke-direct {v1, v2}, $Class;-><init>($Arg1)V
...
invoke-virtual {v1, v3, v4}, $Class;->$Method($Arg1, $Arg2)V
...
invoke-virtual {v1}, $Class;->toString()Ljava/lang/String;
move-result-object v6
...
invoke-direct {v8, v6}, Ljava/lang/StringBuilder;
                                  -><init>(Ljava/lang/String;)V
```

Bytecode Listing 5.4: Method invocations on a class object

If the string argument in the last invocation is to be resolved, the algorithm processes the move operation and proceeds with the associated *toString* invocation on an instance of type *$Class*. Resolving the object register *v1* directly leads to the *new-instance* instruction which does not include any more information. Thus, the result of the analysis would only provide little information about the string origin:

$Class;→toString()

Creating a call history on the object, referred to by register *v1*, provides a lot more information. Starting from the *toString* invocation the algorithm records any class function call on the target object. The search is done path-sensitive and stops when the object definition is reached. After assembling the results, a more comprehensive and detailed view on the state of the object is provided

[2]In dex bytecode, the StringBuilder class is used to perform any operation on strings, including assembling string values from fragments.

at the time of the conversion to a string representation. The final call history looks as follows:

$Class;→<init>(μ($Arg1))→$Method(μ($Arg1), μ($Arg2))→toString()

During the construction of the call history, the arguments of the found invocations are resolved as described before. Object registers that are passed as method arguments in static and non-static function calls are resolved the same way.

### Analysis of Custom Methods

Open-source or custom methods are more interesting for static analysis than API methods. Their bytecode is available in the application package, and thus a deep analysis of the functionality is possible. In a first step, the application is searched for the implementation of the function. In backwards symbolic execution, the analysis usually starts at the end of the method, that is the unique end node in the CFG. All predecessor nodes end with an instruction that exits the function, like *return* and *throw*-instructions. These are the locations at which the algorithm continues the analysis. Note that the resolution of multiple exit instructions will also result in multiple results, as each instruction is analyzed separately.

In general, resolving *throw*-instructions does not give any valuable information for the currently analyzed sink. Usually, these instructions lead to an error message that is displayed to the user in case of an error. Therefore, the algorithm does not process *throw*-instructions.

Resolving *return*-instructions will exhibit the behavior and functionality of the method. This acquired knowledge can then be used in combination with the obtained information about the passed method arguments to deduce an even more accurate result for the current sink. Note that in non-static custom method invocations the object register is *not* resolved, that is no call history is generated. This is due to the fact that the method bytecode is available and thus a comprehensive analysis is possible. During analysis of open-source methods, the algorithm stops when the method header is reached. The functionality of the method has then been reconstructed and the invocation can be evaluated by inserting the resolved argument values.

## 5.5 Assembling & Encoding of Results

The previous section described how backwards symbolic execution works. *Table 5.1* specifies how the analysis proceeds when a certain type of instruction is reached. But, it does not provide information on how fragments of the result, collected during analysis, are stored and assembled. Just merging all token to a string and output it as final result does not give a proper view on the data that reaches the predefined sink.

In case of a binary operation, it is not sufficient to just store the resolved values of the operands. In order to provide complete information, the type of operation needs to be stored additionally. For example, the result of an addition is different to the result of a multiplication of two operands. Without storing meta information, the data type of values is lost as well, which makes an evaluation pretty difficult. Therefore, not only the resolved values have to be stored, but also meta information about instructions to be able to output accurate and detailed information about an operation.

In general, expressions cannot be evaluated directly during analysis. An operation with two constant values is easy to evaluate, but an operand itself could be another expression. In order to evaluate the enclosing expression, the inner expression needs to be resolved first. During analysis, it is not clear whether the current set of token is sufficient to evaluate an expression completely or if some important information is yet to be resolved.

Thus, we need a proper and sophisticated encoding to store all necessary information during the symbolic execution. Information should be stored in a compact way such that the results can be assembled efficiently and be output after the analysis stops. Encoding is necessary not only for unary and binary operations, but for any instruction for which simple storing of resolved register information is not sufficient. In these cases, additional meta information about the opcode is required. In order to handle complex control flow, arbitrary nesting of different instruction encodings must be supported.

Before the actual encoding schemes are explained, the term *result* is defined in the context of the static analysis. A result is a list of token that is created and extended during the backwards symbolic execution. A token is either a constant value like a string or numeric value, or a special marker that is used to encode meta information. In dex bytecode, strings are always written in quotes, thus there is no way to interfere with encoding markers. All numeric values are stored in a hexadecimal representation in dex bytecode and start with a *0x*. Therefore, the first digit in opcode identifiers is always greater than zero.

For the resulting list structure, three groups of instructions need a proper encoding scheme. These groups are expressions, arrays and method invocations. The following sections introduce the encoding schemes for all groups. They are presented as syntax diagrams which show the essential parts while omitting trivial details for the sake of simplicity. This includes for example the syntax format of strings or hexadecimal values.

### 5.5.1 Encoding Expressions

Expressions include unary and binary expressions as well as definitions of constant values. *Figure 5.6* shows the encoding scheme for expressions. It includes the root syntax diagram *expression* as well as diagrams for *constants* and complex *values*.

A trivial expression, is an expression that defines a *constant* value. There are opcodes to define string and integer constants as well as multi-purpose opcodes to define a fixed-size value of arbitrary type. Any numeric value in dex bytecode is stored in hexadecimal notation. Thus, constant values have to be encoded in order to store the type information. The most compact way of storing this meta data is to use the 2-digit hexadecimal opcode that is unique among all instructions[3].

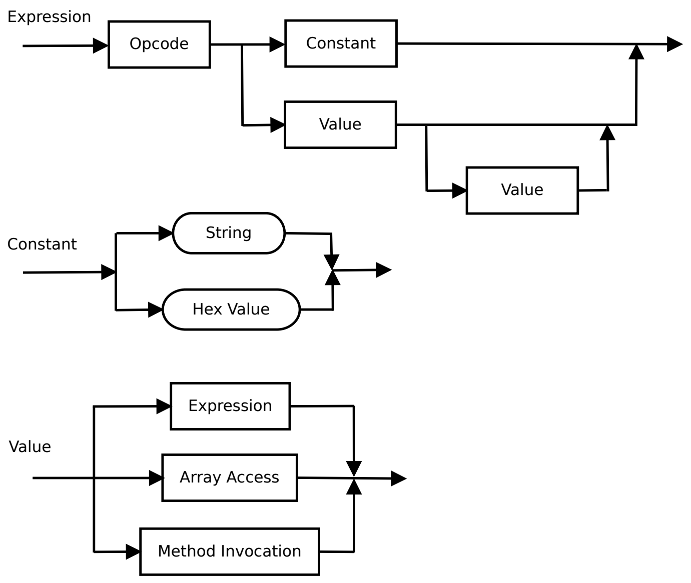


Figure 5.6: Expression encoding

Unary and binary operation instructions are encoded the same way by prepending the opcode identifier. This provides information for both data type and kind of operation. The opcode ID is followed by one or two *values* depending on the type of operation. Values in unary and binary operations can be another expression, the result of an array access, or the return value of a method invocation. The encoding scheme for expressions does even allow invalid combinations, like strings in operations or method invocations that return non-numeric values. For the sake of simplicity, the encoding scheme does not have restrictions on complex structures or on return values of certain operations. However, these invalid combinations only exist in theory. In practice, the dex bytecode guarantees that the operands are numeric values. If one of the operands would not contain a numeric value, the dx or the javac compiler would have generated an error.

Using the opcode for storing meta information works fine for all opcodes, except for the two multi-purpose opcodes. The opcodes *14* and *18* define values of arbitrary data type of 32-bit and 64-bit size, respectively. This means that a 32-bit register can either hold an integer or float value, whereas a 64-bit value might have type long or double. The Dalvik virtual machine does not care about the concrete data type, the size of the value is the only crucial information. During conversion from JVM bytecode to dex bytecode, the type information is lost and there is no way to recover the actual data type by analyzing the opcode in isolation.

However, it is possible to retrieve the data type through context parsing. By analyzing method invocations that use this value, the data type can be extracted

[3]The hexadecimal opcodes can be looked up at `http://s.android.com/tech/dalvik/dalvik-bytecode.html`

from the method header. In order to store this additional information, new custom opcodes have to be defined. Existing opcodes cannot be exploited to store the type information without changing the instruction semantics. For each of the multi-purpose opcodes, two new opcodes are devised. The semantics remain unchanged, but the type information is provided additionally.

*Table 5.2* depicts the four custom opcodes. The opcode identifier deliberately include non-hexadecimal characters. This way, they won't interfere with future changes in the bytecode specification. The templates are used during evaluation as fallback. If the actual value is not resolvable, outputting the template with the resolved operand provides the most detailed information. This is described in depth in *Section 5.6.*

| Opcode | Template | Description |
|---|---|---|
| 1w | (int) $1 | 32-bit integer value |
| 1x | (long) $1 | 64-bit long value |
| 1y | (float) $1 | 32-bit float value |
| 1z | (double) $1 | 64-bit double value |

Table 5.2: New custom opcodes for encoding type information

Note, that in case of unary and binary operations, the opcode always specifies the resulting data type. Thus, the type of the operands is known implicitly, even if one of the operands is defined via a multi-purpose opcode.

The custom opcodes are not used directly when the algorithm reaches a multi-purpose instruction. Instead, the custom IDs are added to the current list of token when type information is exposed in method calls during parsing. Since these opcodes are idempotent, it doesn't matter if they are added multiple times during analysis.

The following token list represents the encoding for an example integer operation; the value 5 is subtracted from the value 10:

91, 12, 0xa, 12, 0x5

The opcode for the subtraction is followed by the encoded operands. Opcode 12 describes a 4-bit signed integer value. This type information is used for both converting the hexadecimal values into more readable integer values and for evaluating the expression.

## 5.5.2 Encoding Arrays

Resolving arrays is handled specially during analysis in order to reconstruct the elements at a certain point in time. Fragments obtained during array reconstruction need to be encoded properly, such that the array elements can be identified and put at the correct location within the array. *Figure 5.7* depicts the encoding scheme for both *array definitions* and *array accesses.*

The enclosing markers of the array definition encoding are *Array* and *Array-End.* Within these markers, the reconstructed element information and array

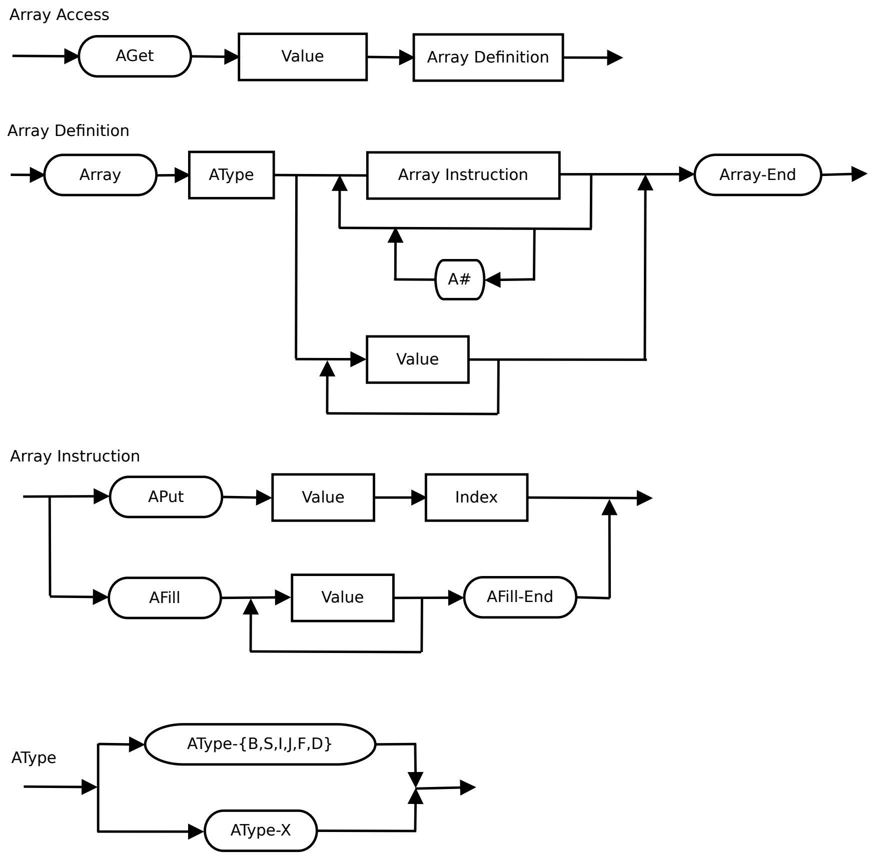


Figure 5.7: Array and array access encoding

meta data is stored. The array type is extracted from the corresponding new-array instruction. It is encoded via special *AType* markers. There are dedicated markers for all primitive types like integer, float or double values. Any non-primitive class type is marked with *AType-X*. Object types do not have to be discriminable, since the type information is only required for converting primitive types correctly. The type marker is followed by at least one *array instruction*.

In dex bytecode there are only two types of array element modifying opcodes. The aput-opcodes modify the element at the specified index. The instruction is encoded with a *APut* marker followed by the resolved value register and the resolved index register.

The second one is the *fill-array-data* opcode. It performs a bulk initialization of an array by setting multiple elements at once. The values to be filled into the array are stored in a dedicated table within the bytecode. The number of elements used for initialization does not necessarily have to be equal to the array size. Thus, the encoding does not only require a start marker *AFill* but also an end marker *AFill-End*.

Path-sensitive reconstruction in the array definition scheme is encoded by use of the *A#* marker. It separates the array instructions of different paths.

There is also an atomic instruction to define and fill a new array. The *filled-new-array* returns a reference of a newly created array that is being filled with values specified by the argument registers. The corresponding encoding is shown in the second path of the *array definition* diagram. Instead of using array instructions to describe the modifications of elements, the element values are directly placed after the initial marker. This is because the values can be directly resolved and the algorithm does not need to search for element modifying instructions.

Besides the array definition, there is another diagram that describes the encoding for *array accesses*. Values are retrieved from arrays via aget-opcodes. In order to evaluate these instructions correctly, the index of the element to be accessed and the array content are required. The encoding scheme is pretty straightforward. An *AGet* marker indicates an array access. The index argument and the encoded array are placed behind.

For more clarification, a simple array encoding is presented in the following. It shows the token list generated for the integer array [1,3]:

Array, AType-I, APut, 12, 0x3, 12, 0x1, AFill, 12, 0x1, 12, 0x7, AFill-End, Array-End

The list includes two array instructions. A fill-array-data opcode has been used to initialize the array with the values 1 and 7. Afterwards, an aput-instruction modified the value of the second element. Note that the encoding always starts with the array instruction that is closest to the sink. In this example, storing the array data type would not be necessary since the information is already encoded through opcode 12. However, it is required if the data type is not exposed through the value encoding, for example if multi-purpose opcodes are used.

### 5.5.3 Encoding Method Invocations

There are two types of method invocations, custom and API method invocations. Additional semantic analysis can be applied to open-source methods, as their bytecode is available within the application. Therefore, both kinds of invocations need to be encoded differently. *Figure 5.8* depicts the syntax diagrams for *method invocation* and *argument* encoding. An argument can be either a *value* or an *array definition.*

When processing API method invocations, the algorithm resolves all method arguments. The method itself cannot be resolved and is considered a dead-end. However, storing the full method header also provides significant information. The method header includes the class name, the method name, and the argument list. For the sake of simplicity a distinct diagram for the *method header* is omitted.

The encoding uses enclosing markers *MA-Start* and *MA-End.* The start marker is followed by the method header and the resolved argument registers, if any. The argument list in the method header is replaced by a counter indicating the number of arguments. Furthermore, the return type is removed.

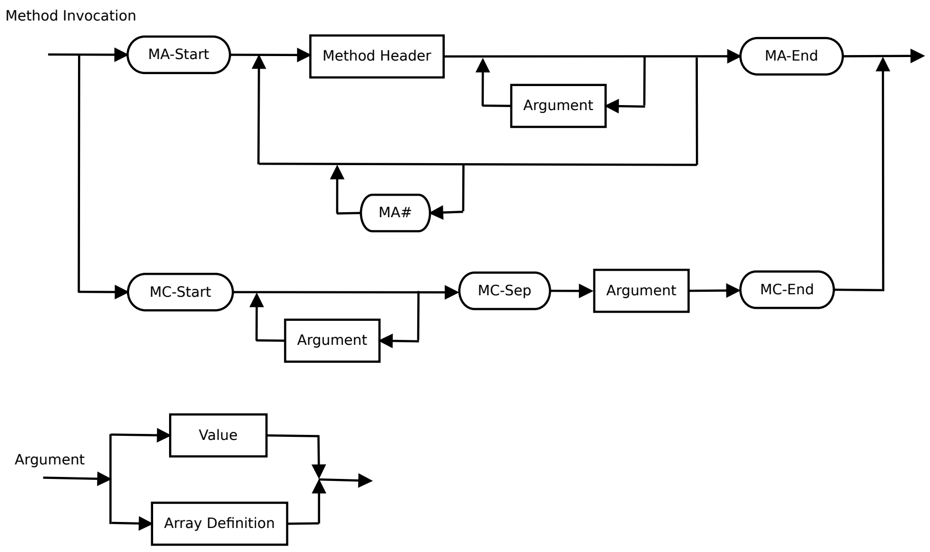


Figure 5.8: Encoding of method invocations

The following example shows the encoding of an append function of the String-Builder class. It has been invoked with a single string argument *foo*:

MA-Start, "Ljava/lang/StringBuilder→append(#1)", 1a, "foo", MA-End

Call histories are encoded as nested method invocations. Any subsequent invocation is placed right after the last argument of the previous invocation. Since call histories are constructed path-sensitive, the encoding must take this into consideration. The *MA#* marker separates method invocations of different paths. By using the separator, a distinct call history can be created for each path. The following token list presents a path-sensitive call history encoding. $\text{Inv}_i$ describes an invocation including method header and arguments:

MA-Start, $\text{Inv}_1$, $\text{Inv}_2$, MA#, $\text{Inv}_3$, $\text{Inv}_4$, MA# $\text{Inv}_5$, MA-End

The encoding describes two distinct call histories. The first two method invocations are part of both histories. Then the control flow splits and the MA# marker separates the invocations for both paths. Thus, the first history includes the invocations $\text{Inv}_1$, $\text{Inv}_2$, $\text{Inv}_3$, and $\text{Inv}_4$, whereas the second one includes the invocations $\text{Inv}_1$, $\text{Inv}_2$, and $\text{Inv}_5$. Note that the first element is again the invocation that is reached first during the backwards symbolic execution.

Custom methods can be fully resolved and thus have to be encoded slightly different. The resolved method invocation arguments are placed right after the start marker *MC-Start*. The *mandatory MC-Sep* marker separates the arguments from the resolved method. Resolving a custom method starts from the unique end node and stops once the method border is reached. Intuitively speaking, the algorithm resolves each register that is returned from this method, for example via a return opcode. Method argument registers are resolved until the beginning of the method is reached. Then, they are replaced by a placeholder that includes the position of the argument within the argument list. For example, the second method argument is then replaced by a marker §2. During evaluation, the placeholders are replaced by the associated resolved argument values. Finally, the whole encoded method invocation is evaluated and the result is output.

### 5.5.4 Multi-path Encoding

Only trivial methods have sequential control flow. Control flow splitting instructions like switch statements or if-conditionals create additional paths in the CFG. In backwards symbolic execution there are various situations in which multiple paths have to be followed and resolved. This also implies multiple results for the currently processed sink. In order to be able to distinguish resolved token of different paths correctly, places at which the control flow splits must be encoded properly.

Intensive testing showed that multi-path encoding via marker is both complex and infeasible for typical applications. In general, the effort for decoding the result is disproportionate to the effort necessary for encoding. Furthermore, the representation of complex nested structures is usually infeasible in terms of performance and also interfered with non-multi-path encodings. The previous encoding schemes described some exceptions, for example the path-sensitive array reconstruction via markers. The reason for that will be explained later in the section.

In general, a more sophisticated approach is necessary to encode multi-path results properly. The most obvious solution is to encode these structures in a graph. A directed, acyclic graph (DAG) provides the best representation for this kind of encoding. A DAG is a directed graph that has no directed cycles. This matches the assembly of results during backwards symbolic execution best. The graph can be constructed node by node during analysis with only little effort. It is also a memory efficient representation since shared subpaths are only stored once.

In combination with the previous encoding schemes, the algorithm uses an hybrid approach of graph and list encoding. Any node in the DAG contains a list of encoded tokens found during analysis. If the control flow splits, new nodes are created to represent the various paths. There are a few cases that cause the resolver to follow multiple paths:

- **Phi statements** are placed in join nodes to merge control flow from multiple paths. When the algorithm reaches such a statement, the argument registers have to be resolved in their associated path. The number of outgoing links that have to be created, equals the number of arguments of the $\phi$-function.
- **Class field access** resolution does not necessarily create new nodes in the target graph. But if the access cannot be resolved in the current method, the algorithm searches the application for accesses of this class field. This may produce multiple results, which must be resolved sequentially.
- **Method resolving** If the algorithm resolves a register within a method and does not find an associated definition, then the register is a method

argument. In order to resolve the value of this register, the algorithm searches the app for invocations of this method. This potentially generates multiple new paths, depending on the number of invocations found.

If the algorithm reaches a point at which the control flow splits (see *Figure 5.9*), a new node in the target graph is created and linked with the current node. Any token that is obtained in the new path is stored in this new node. As backwards symbolic execution performs a depth-first search, the algorithm completely resolves this path before other paths are processed. A new node is created for each new path that is about to be analyzed and an outgoing edge is generated from the split node to the new path node.

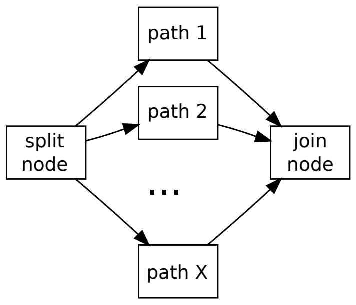


Figure 5.9: Target graph construction

If all paths have been processed, a join node is created and connected to the last node of each path. This common end node merges the control flow after all paths have been resolved. It is necessary to be able to continue building the graph in case there are multiple registers to be resolved at the same instruction. Registers are resolved sequentially, thus resolving the first register will result in a target graph (see the light gray nodes in *Figure 5.10*). Once the first register is complete, the second register is resolved. Instead of building a dedicated graph for the second register and trying to merge them afterwards, any new nodes are appended to the current graph (dark gray nodes). Tokens for the second register are then stored at the first join node.

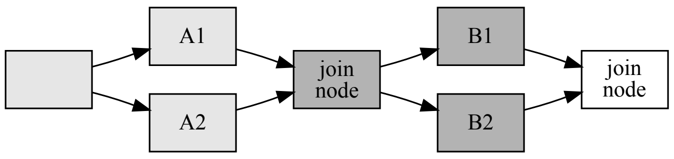


Figure 5.10: Example target graph

In general, this approach will overestimate the final results in some cases. *Figure 5.10* depicts an example target graph for two registers resolved from the same instruction. There are four distinct paths through the graph which lead to four results. If however A1 and A2 are the same nodes as B1 and B2, respectively, there are only two valid results. In this case, the graph encodes some combinations that won't appear during execution of the application. This overestimation originates from building the cross product of all results of the first register with all results of the second register. Other approaches to create and merge dedicated graphs for each register are both complex and error-prone.

*Listing 5.1* shows the code of a simple Java method with a URL sink. When the URL constructor is processed and the *url* variable is resolved, a phi instruction right after the conditional will indicate three paths. In each path, the argument that is appended to the domain *www.foo.com* is set differently.

```
private URL test(){
    int i = 5;
    String url = "www.foo.com";

    if (i < 10)
        url += "/?key=" + 7;
    else {
        if (i > 1)
            url += "/?key=" + i;
        else
            url += "/?key=" + i*i;
    }

    return new URL("http://" + url);
}
```

Listing 5.1: Java method with a URL sink

The graph shown in *Figure 5.11* depicts the target graph for the Java method. Each node contains a list of token that represents the encoded values obtained during analysis.

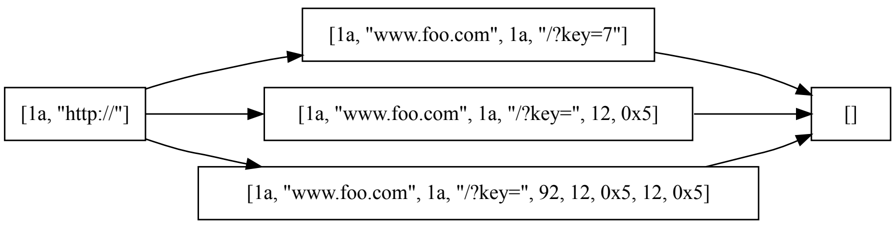

[1a, "http://"]
[1a, "www.foo.com", 1a, "/?key=7"]
[1a, "www.foo.com", 1a, "/?key=", 12, 0x5]
[1a, "www.foo.com", 1a, "/?key=", 92, 12, 0x5, 12, 0x5]
[]

Figure 5.11: Target graph that shows a simple multi-path encoding

*Section 5.4* stated that certain opcodes and array reconstructions are performed path-sensitive. This means that multiple paths have to be analyzed. However, this multi-path analysis is encoded in the token list and not within the graph. The reason for this is that this kind of multi-path encoding is instruction-sensitive. It can simply be expressed within the encoding of a single instruction/structure.

Examples are the array reconstruction and the path-sensitive creation of the call hierarchy. For an array reconstruction, the algorithm searches for array element modifying instructions in the area between current instruction and array definition. Thereby, the resolved instructions are encoded by path using the separation marker *A#*. In order to create a call history, the encoding of invocations is nested and paths are separated via a dedicated marker.

## 5.6 Decoding and Evaluation

Once the analysis is completed, the data that reaches the sink is available as encoded graph. In order to output the final URL strings, several steps have to be performed. The first essential step includes the dissection of the multipath encoding to obtain a set of encoded lists. Therefore, all paths through the graph, beginning at the start node and ending at the last common end node, have to be dumped. Instead of recording the node indices per path, the node token lists are stored and merged to a single list. Each final list represents an encoded value for the specified sink. The token lists are then to be decoded and any expression is to be evaluated, if possible. Finally, the results are output to the user.

Formally, the path decoding is done by performing the following three steps:

- Decoding opcodes and evaluating operations (expressions)
- Decoding arrays and array accesses
- Decoding method invocations (both custom and API methods)

### 5.6.1 Decoding Expressions

Decoding expressions requires connecting the opcode meta information with the actual operand values. In order to provide a comprehensive textual view on expressions, templates for each opcode have been devised. These templates illustrate a human readable presentation of the operation, including all information necessary to evaluate the expression. *Table 5.3* shows a fraction of the set of all templates.

| Opcode | Mnemonic / Syntax | Template | Operand $1 | Operand $2 |
|---|---|---|---|---|
| 15 | const/high16 $v_A$, C | (int) $1 | C | – |
| 7c | not-int $v_A$, $v_B$ | (int) ~$1 | $\rho(v_B)$ | – |
| 7f | neg-float $v_A$, $v_B$ | (float) -$1 | $\rho(v_B)$ | – |
| 90 | add-int $v_A$, $v_B$, $v_C$ | (int) $1 + $2 | $\rho(v_B)$ | $\rho(v_C)$ |
| 98 | shl-int $v_A$, $v_B$, $v_C$ | (int) $1 ≪ ($2 & 31) | $\rho(v_B)$ | $\rho(v_C)$ |
| 9d | mul-long $v_A$, $v_B$, $v_C$ | (long) $1 * $2 | $\rho(v_B)$ | $\rho(v_C)$ |
| a9 | div-float $v_A$, $v_B$, $v_C$ | (float) $1 / $2 | $\rho(v_B)$ | $\rho(v_C)$ |
| cc | sub-double/2addr $v_A$, $v_B$ | (double) $1 - $2 | $\rho(v_A)$ | $\rho(v_B)$ |
| d6 | or-int/lit16 $v_A$, $v_B$, C | (int) $1 \| $2 | $\rho(v_B)$ | C |
| df | xor-int/lit8 $v_A$, $v_B$, C | (int) $1 ^ $2 | $\rho(v_B)$ | C |

Table 5.3: Selected templates for expressions

The templates include placeholders for the operands. During decoding, the template is selected based on the opcode that has been used for encoding. The placeholders are then replaced by the concrete operand values. This provides a first high-level view on the expression. The next step towards the evaluation is to convert the operand values. Any numeric value in Dalvik is stored as hexadecimal value. All templates include information about the result type of the operation. Thus, it is possible to convert the hex values into the corresponding data type. Finally, the operation can be evaluated and the resulting value can be output.

The conversion of data types fails if an operand is not a number, but a closed-source method invocation that returns a number. In this case, the operand is replaced by the invocation and the template is output as fallback. This way, the user gets the most detailed information possible.

However, placing an unresolvable expression directly into the final URL, reduces the readability dramatically due to the increased length of the result. Furthermore, the important high-level view on the URL is mixed with low-level information about a specific value. Thus, the expression is not placed directly into the result. Instead, a label is placed at the position of the expression and the unresolved data is put in an extra line. This improves the readability of the result by abstracting low-level parts away from important parts of the URL without omitting information.

### 5.6.2 Decoding Arrays

Decoding arrays and array accesses requires reconstructing the elements at the time of usage/access. Given a list of token, the decoding algorithm extracts any sublist which encodes an array construct. These sublists can either start with the *Array* or *AGet* token. Any other marker associated with arrays can only appear within the array encoding. The *AType* marker provides information about the array element type. In case of a primitive type, this information is used to convert the element values accordingly.

In the simplest form of array encoding, all element values are given directly in the correct order. This is usually the case when the *filled-new-array* opcode has been used to define and fill an array with a single instruction. Given an array encoding

$$\text{Array AType-}x\ val_1\ val_2\ val_3\ val_4\ \text{Array-End}\,,$$

the decoder outputs [$val_1$, $val_2$, $val_3$, $val_4$], whereas the values have been converted according to the specified type.

The other form of array encoding doesn't reveal the element values directly. Instead, array instructions, as depicted in *Figure 5.7*, describe how element values changed in the region between array definition and a certain instruction. The reconstruction assembles the array by starting from scratch and subsequently applying the encoded instructions. At the time of usage/access only the last value of each element matters. Due to the backwards traversal, the instructions are ordered from array usage to array definition. Thus, an element in the reconstructed array is only modified, if it hasn't been set before.

Decoding array instructions is pretty straightforward. The *aput*-encoding simply sets a value at a given index. In order to decode an *AFill* instruction, multiple values are set in the target array, starting at index zero.

*Table 5.4* shows an example reconstruction via array instructions. Note, that the array size is not necessary for the actual reconstruction and therefore it is not included in the encoding scheme. The size of the reconstructed array is adapted dynamically, if the current size does not suffice for processing the next instruction. Increasing the size might result in unknown values, that haven't been set so far. These elements are labeled with $x$.

| Instruction | Reconstructed Array |
|---|---|
| ∅ | [ ] |
| APut 3 2 | [x, x, 3] |
| APut 8 0 | [8, x, 3] |
| APut 7 4 | [8, x, 3, x, 7] |
| APut 4 2 | [8, x, 3, x, 7] |
| AFill 2 2 2 AFill-End | [8, 2, 3, x, 7] |

Table 5.4: Reconstruction of an array

The algorithm starts with an empty array. The first aput instruction sets the third element of the array. Thus, the array size must be increased before the value can be set. Values for the first two elements are unknown at that point in time and are therefore marked with an $x$. The next aput instruction sets the value eight at index zero. The subsequent aput extends the array size again. The next instruction then tries to set an element for which a value is already known. This fails, since the algorithm prevents overwriting the existing value. Finally, the fill instruction tries to set the first three elements of the array. Only the unknown second element is replaced by the given value.

After the reconstruction there is a chance that some of the elements are still unknown. This does not cause problems, because such a situation can also be caused in a Java program. If these elements would be accessed by the application, a runtime exception would be thrown.

The array reconstruction is performed path-sensitive. Instructions of different paths are separated by the $A\#$ marker. For each path, a distinct array is assembled. The number of results, that are finally output, is directly related to the number of reconstructed arrays. If, for example, the reconstruction outputs two arrays, the whole token list must be cloned and the corresponding array encodings must be replaced by the respective arrays.

The encoding scheme does also provide an encoding for array accesses. The *AGet* encoding simply includes an encoded array and the index of the element to be retrieved. Its decoding requires the array to be decoded first, before the element at the specified index can be accessed and output.

### 5.6.3 Decoding Method Invocations

The decoding of method invocations includes both custom and API methods. The encoding schemes provide information about the arguments passed to the function. During decoding, the arguments used in the invocation are resolved first. In case of an API method invocation, the argument header definition is replaced by the resolved argument values. Given the method header of a StringBuilder class method, e.g. Ljava/lang/StringBuilder;→append(Ljava/lang/String;), the method argument type is replaced by the resolved string value, for example Ljava/lang/StringBuilder;→append("foo").

A call history provides a chronological list of invocations executed on a class instance object. Histories are encoded as nested closed-source method invocations. Due to the backwards analysis approach, the invocations are encoded in reverse order which means that the final invocation represents the first executed method on the target object. This is usually the constructor call which will be placed at the very beginning of the call history.

Given a set of invocations and the associated resolved arguments, a call history is created as follows:

$$\text{\$Class;}\rightarrow\text{<init>}([\mu(Arg_1,..)])\rightarrow\text{\$Method1}([\mu(Arg_1),..])\rightarrow\text{\$Method2}([\mu(Arg_1),..])$$

It starts with the class name followed by the constructor invocation and its corresponding arguments, if any. Any additional invocation, indicated by the right arrow, is appended to the current history string. This results in a compact and readable representation of a call history. *Table 5.5* depicts a simple example with a StringBuilder object. The invocations are already in chronological correct order.

| Method invocation header | Resolved Argument |
|---|---|
| Ljava/lang/StringBuilder;→<init>(Ljava/lang/String;) | "http://www.foo.com/" |
| Ljava/lang/StringBuilder;→append(Ljava/lang/String;) | "?key=" |
| Ljava/lang/StringBuilder;→append(I) | 12345 |
| Ljava/lang/StringBuilder;→toString() | ∅ |

Table 5.5: Set of invocations to form a call history

The history is composed of four invocations. The method invocation header includes type information of the arguments. In a first step, this type information is replaced by the resolved argument values. Finally, the history string is assembled as described before. The following history represents the output for the example invocation set (the line wrap is due to the limited horizontal space):

Ljava/lang/StringBuilder; → <init>("http://www.foo.com")
→ append("?key=") → append(12345) → toString()

The generation of a history is performed path-sensitive. The *MA#* separation marker is used to distinguish between invocations of different paths. Similar to the reconstruction of arrays, multiple paths lead to multiple distinct call histories. For each history, the token list needs to be duplicated and the corresponding encoding must be replaced with the respective history.

Placing a decoded API method invocation or a call history directly into the URL would cause a massive blowup of the result. Furthermore, it makes the view on the essential parts of the URL difficult. Therefore, any invocation in the result is replaced by a label during a post-processing step. The corresponding label along with the actual invocation is then printed in an extra line to improve the readability of the result, similar as it is done with unresolvable expressions.

### Custom Method Invocations

Decoding open-source method invocations starts with the decoding of argument values. Thereby, the marker *MC-Sep* separates the encoding of the method functionality and the encoded arguments passed to the invocation. Resolving a function starts at the unique end node of its CFG and stops once the method header is reached. Any unresolved method argument is subsequently replaced by a marker indicating its exact location within the argument list.

The evaluation of custom methods starts by replacing these markers with their associated resolved argument values. Then, the method functionality can be fully resolved. Any expression is evaluated and finally a value of the method's return type is output.

# Chapter 6

# Implementation

The last chapter described the theory behind the information flow analysis on Android applications. This chapter describes in detail how these concepts are realized in a concrete tool called *Bati*. The analysis tool is a standalone Java application. The design and implementation is highly modular and a later port to Android is possible with only little effort. *Figure 6.1* shows a high-level view on the static analysis approach.

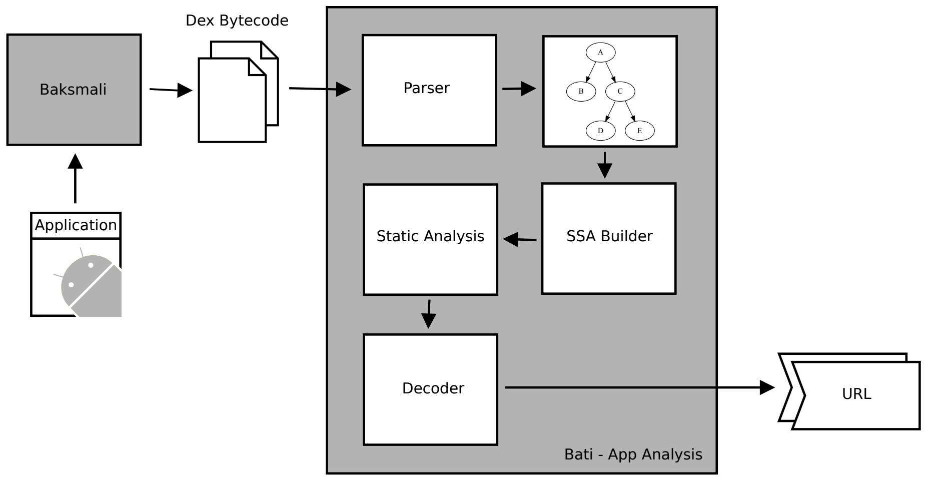


Figure 6.1: High-level view on the static analyzer

The following sections describe specific implementation details of the individual modules. Besides explaining the actual techniques that are used to solve certain tasks, aspects on how to save memory, to build acceleration structures, or to improve the performance are pointed out as well.

## 6.1 Disassembling and Parsing

The first step towards the analysis is to retrieve the application's bytecode. Therefore, a disassembler is used to translate the compact dex class file into a more convenient assembly language. The Android SDK includes a default disassembler called dexdump which provides only limited functionality. Some data structures like initialization tables for arrays are not dumped. In addition to that, the output is not very intuitive and quite hard to read and process.

Besides the default disassembler, there are two popular open-source disassemblers which offer a more comprehensive functionality. These tools are dedexer [48] and baksmali [36]. Both disassemblers fully support the dex bytecode instruction set and generate a Jasmin-based output which is much more readable[1] Furthermore, both tools include odex[2] support in their latest versions.

The tools are easy to handle and offer a great set of functionality. However, baksmali has a better support and a larger community. Its code base is more structured and it includes a distinct comprehensive parsing library for dex bytecode. Therefore, we chose baksmali for the disassembling task.

Disassembling an Android application produces a set of dex bytecode files, one for each class used in the app. In order to process the output files and to extract important information, the bytecode needs to be parsed. In a first step, Bati's parsing module reads in the application bytecode. In the process, the instructions are parsed and stored in basic blocks which are used in a later step to generate the control flow graph. Dumping the instructions block-wise also improves the readability of bytecode, because control flow sources and targets can be identified without effort. In contrast to this, baksmali outputs bytecode instructions separated by a blank line, which provides no visual assistance in identifying control flow.

*Bytecode Listing 6.1* shows the dex bytecode of the simple program presented in *Section 4.3.* The output on the left shows the default output format that is used by the disassembler baksmali. The format on the right is the block-wise output format that is generated by Bati. Instructions belonging to the if-block and else-block can easily be identified.

Besides parsing bytecode instructions, the module also parses data of array initialization tables and switch branch offset tables. Information stored in branch offset tables is required to be able to connect basic blocks in switch statements correctly. Array initialization tables include values that are used for bulk array initialization, for example through the *fill-array-data* instruction. This kind of array initialization is only performed for arrays of primitive types like integer or float. *Bytecode Listing 6.2* depicts a float array with its corresponding supplemental data block that is used to set multiple array elements directly.

The instruction and the data block are connected via a label. The *array-data* annotation includes three elements of type float. Each line contains 4 bytes = 32 bit for the specification of one float value. Thereby, the trailing *t* indicates a

[1]Jasmin is a popular assembler for the Java Virtual Machine. It takes Java classes, written in a simple assembler-like syntax, and converts them into binary Java class files.

[2]odex files are optimized dex files which are generated at runtime by Dalvik's JIT compiler

```
const/4 p2, 0x5

add-int/lit8 p2, p2, -0x3

const/4 v0, 0x4

if-ge p2, v0, :cond_0

mul-int/lit8 p3, p2, 0x2

move p1, p3

:goto_0
sub-int p1, p2, p3

add-int/lit8 p4, p3, 0x2

return-void

:cond_0
const/4 v0, 0x3

sub-int p3, p2, v0

goto :goto_0
```

(a) Default baksmali output

```
const/4 p2, 0x5
add-int/lit8 p2, p2, -0x3
const/4 v0, 0x4
if-ge p2, v0, :cond_0

mul-int/lit8 p3, p2, 0x2
move p1, p3

:goto_0
sub-int p1, p2, p3
add-int/lit8 p4, p3, 0x2
return-void

:cond_0
const/4 v0, 0x3
sub-int p3, p2, v0
goto :goto_0
```

(b) Block-wise output

Bytecode Listing 6.1: Difference between dex output formats

byte literal. The actual byte values are stored in reversed order and thus the first element is 0x3fb47ae1. The three elements are the hexadecimal representation of the values 1.41, 3.75, and 2.22, respectively.

```
new-array v1, v2, [F
fill-array-data v1, :array_0

:array_0
   .array-data 0x4
       0xe1t 0x7at 0xb4t 0x3ft
       0x0t 0x0t 0x70t 0x40t
       0x7bt 0x14t 0xet 0x40t
   .end array-data
```

Bytecode Listing 6.2: Array definition with initialization table

During the initial phase, information extracted from the bytecode is also used to generate lookup maps and acceleration structures. They are used during the construction of the SSA form and during the actual analysis to reduce the need for re-parsing data. Among the most important data structures is the one providing information about the various method invocations within the application. It includes which custom methods are invoked in which methods and whether a method is invoked recursively. The differentiation between open-source and closed-source method invocations is based on the full class name. Classes that are included in the public Android API[3] are classified as closed-source methods.

[3]http://developer.android.com/reference/packages.html

Another data structure assembles data about Java interfaces within the application and classes that implement them. This knowledge is important during analysis, if an *invoke-interface* instruction is reached and the algorithms needs to search for classes that implement this method to be able to resolve the call.

Storing data about assignments to class fields increases the performance when the resolver algorithm searches the application for new definitions of that class field. Without this data structure, the complete bytecode must be re-parsed during analysis to obtain this information.

Another essential task of the parsing module is the search for predefined sinks. If an application does not contain any sinks of interest, the tool can stop the analysis right after parsing. Otherwise, if at least one sink is found, its location is stored, such that the static analyzer can directly start at that very instruction. The location of an instruction is stored as compact 4-tuple that is composed of numeric IDs for the class, method, basic block, and instruction. Since the control flow graphs are created on a per method level, the method ID refers to the associated CFG.

## 6.2 The SSA Builder Module

Building the SSA form is an essential prerequisite for an efficient backwards symbolic execution. Without this intermediate form the tracing for register definitions would become infeasible and extremely complex. Translating bytecode into the SSA form usually requires two steps. Placing phi functions to handle locations where control flow is merged from multiple paths and the renaming of registers, such that each register is defined exactly once.

However, depending on the chosen approach, additional steps have to be done to create the necessary data structures. Besides the common approach to place phi functions by using node dominance information, there also exist approaches based on DJ graphs and merge sets (see *Section 4.4.1*).

The current implementation of Bati uses an approach based on control flow graphs. The CFG is not only used for constructing the SSA form, it is also an important data representation during analysis. If registers have to be resolved path-sensitive, the graph helps to keep track of data flow. Computing dominance on a CFG might be slightly slower than comparable approaches, however their performance gain usually shows up only on larger graphs. Based on empirical evidence, control flow graphs per method have less than 100 nodes. Thus, neither approach is significantly faster in this scenario.

The actual generation procedure for the SSA form includes the following steps:

1. Generate the control flow graph
2. Calculate the reverse postorder
3. Compute dominator
4. Compute dominance frontiers
5. Place phi functions
6. Rename registers

The basic blocks have already been created during parsing. In order to create the control flow graphs, the blocks have to be connected. The way basic blocks are connected strongly depends on the block type and is described in detail in *Section 5.2.1.*

The graph generating algorithm additionally includes a module to visualize the CFG. This module creates a textual representation of the flow graph and uses the open-source graph visualization software *GraphViz*[4] to generate a picture of the CFG. The GraphViz binaries can be used transparently via a Java wrapper API[5]. The visualization plugin is not necessary for creating the SSA form. Instead, it is some kind of visual debugging that can be used to generate a picture of the control flow graph and the various paths within.

## 6.2.1 Determining Node Relationship

The next steps gather and compute information about the node relationship necessary to create the SSA form. Computing dominance on a CFG has a long history in literature. There exist multiple approaches which vary in computation time and complexity. One of the best-known and most widely used algorithms is the Lengauer-Tarjan algorithm [40]. It computes dominance in $O(E * log(N))$, where E is the number of edges and N the number of nodes. Another technique was proposed by Cooper *et al.* [17]. Their iterative algorithm uses well-engineered data structures and computes dominance in $O(N^2)$.

Although the algorithm of Lengauer-Tarjan has faster asymptotic complexity, it requires unreasonably large graphs ($\geq$ 30.000 nodes) to show a performance advantage compared to the iterative algorithm. Small graphs benefit from the improved performance of the iterative solution. Additionally, it is a very memory efficient algorithm.

Therefore, the analysis tool implements the iterative approach as proposed by Cooper *et al.* [17]. In the following, the algorithm is explained in detail along with the necessary data structures.

Dominance is computed by the iterative *Algorithm 6.1.* In order to apply this algorithm, nodes have to be traversed in reverse postorder. This can be determined by calculating the postorder first and then reversing the resulting sequence in a subsequent step. The postorder graph traversal is the following:

1. Right subtree
2. Left subtree
3. Root

In general, it is possible to start with either the right or the left subtree. In the actual implementation, the right subtree is traversed first. This implies that in reverse postorder the left subtree is traversed first, which means that in if-conditionals the then-branch is processed before the else-branch.

[4]Hosted at `http://www.graphviz.org/`

[5]Bati uses an open-source Java wrapper created by Laszlo Szathmary. The source is freely available at `http://www.loria.fr/~szathmar/off/projects/java/GraphVizAPI/index.php`

**Algorithm 6.1**: Iterative algorithm to determine dominance

```
for all nodes n do {initialize the dominator array}
  doms[n] ← $\varnothing$
end for
doms[start_node] ← start_node
changed ← true

while changed do
  changed ← false
  for all nodes n in reverse postorder (except start_node) do
    new_idom ← first (processed) predecessor of n

    for all other predecessors p of n do
      if doms[p] $\neq \varnothing$ then {i.e. if it is already calculated}
        new_idom ← intersect(p, new_idom)
      end if
    end for

    if doms[n] $\neq$ new_idom then
      doms[n] ← new_idom
      changed ← true
    end if
  end for
end while
```

By using a reverse postorder sequence, the graph traversal starts at the leaf nodes. It ensures that a parent node is not traversed before all child nodes have been processed. This is important since dominance information can only be computed at a parent node if dominance has been already computed for all child nodes. The algorithm does also work without reverse postorder traversal. However, additional iterations might then be required to compute the complete dominance information.

Recall that a node $n$ dominates a node b if n lies on every path from the root node to b. The set of dominator of b is denoted by Dom(b). The immediate dominator of b, denoted IDom(b), is the single node of Dom(b) that is next to b. Furthermore, a dominator tree is defined for all nodes except the root node $n_0$ as

$$\mathrm{Dom}(b) = \{b\} \cup \mathrm{IDom}(b) \cup \mathrm{IDom}(\mathrm{IDom}(b)) \ldots \{n_0\} .$$

Instead of keeping distinct dom sets, the algorithm uses a compact dominator tree to manage dominator sets. The sets are implicitly represented by the tree. This data structure provides several benefits. It is a compact scheme since IDom(b) is stored only once. Modifications can be done in-place, that is without having to copy data or to allocate new data structures.

In the algorithm, the tree is implemented as single array *doms*. For any node $b$, the inclusion into its dom set dom(b) is done implicitly. The array element doms[b] holds IDom(b). Element doms[doms[b]] then holds the subsequent entry, which is IDom(IDom(b)). The dominator set of a node b can then be reconstructed by walking through the doms array, starting at b.

The main loop in the algorithm updates the doms array. Starting at leaf nodes, the algorithm computes the IDom of the current node by performing intersections with its predecessor nodes. The intersection routine is depicted in *Algorithm 6.2.*

**Algorithm 6.2**: Two-finger algorithm for node intersection

```
Require: Nodes n1, n2
  finger1 ← n1
  finger2 ← n2

  while finger1 $\neq$ finger2 do
    while finger1 $<$ finger2 do
      finger1 ← doms[finger1]
    end while

    while finger2 $<$ finger1 do
      finger2 ← doms[finger2]
    end while
  end while

  return finger1
```

It implements a *two-finger* algorithm. Given two nodes $n_1$ and $n_2$, one can imagine two fingers pointing at the associated dominator sets. Each finger moves independently until both fingers point at the same element. Note, that the comparisons are on postorder numbers and nodes higher in the dominator tree have higher postorder numbers. Thus, the finger pointing at the smaller element is moved backwards.

The resulting set starts with the found element. The dominator set is assembled by traversing the array backwards until the root element is reached. The computation for a node continues until its dominator set doesn't change in two subsequent iterations. The algorithm stops when the complete doms array has been computed.

## Dominance frontier

After having computed dominance information, all necessary prerequisites are in place to compute the dominance frontiers. The dominance frontiers describe the exact places at which phi functions have to be placed during SSA construction. Recall that the dominance frontier of a node $n$ was defined as the set of all CFG nodes b, such that n dominates a predecessor of b, but does not strictly dominate b.

Given this definition and the dominance information, *Algorithm 6.3* can now be applied to compute the dominance frontiers. In a first step, the method selects all join nodes within the graph at which control flow of multiple paths is merged. Formally, a join node $n$ is a node with at least two predecessors. For each predecessor p the dominator tree is traversed starting at p. The algorithm stops when the immediate dominator of node n is found. The join node is in the dominance frontier of each traversed node except for its immediate dominator. The remaining dominator of n are shared by n's predecessors as well.

**Algorithm 6.3**: The dominance-frontier algorithm

```
for all nodes n do
  if the number of predecessors of n $\geq 2$ then
    for all predecessors p of n do
      runner $\leftarrow$ p

      while runner $\neq$ doms[n] do
        add n to runner's dominance frontier set
        runner $\leftarrow$ doms[runner]
      end while
    end for
  end if
end for
```

### 6.2.2 Modifying Dex Bytecode

The final steps towards the translation into the static single assignment form are the placement of phi functions and the renaming of registers. It starts with instrumenting the bytecode by adding $\phi$-functions at the nodes determined by the dominance-frontier computation. However, this information includes only the target nodes. It is not clear which registers in these nodes require a phi function.

This kind of information has to be collected in an additional pre-processing step. In each target node, registers that are assigned a new value are recorded. A distinct $\phi$-function is necessary for any element of this resulting set of registers. The classification of instruction registers is accelerated by use of a predefined hashmap.

An argument in a bytecode instruction can either be a register, a literal, or a numeric value. For the translation into the SSA form, non-register arguments are of no interest. Register arguments instead, can be further separated into registers that define a new value and source registers. This register classification is fix and can be predefined for all opcodes. *Table 6.1* shows an excerpt of the classification map.

A dex bytecode instruction has up to three arguments. A set of registers, that is for example used in invoke instructions, is considered as a single argument. Registers within a set are always source registers. A dash sign denotes that either the instruction does not have this argument or it is some constant value that does not have to be processed. Destination registers can only appear as first argument.

The hashmap allows classification of instruction registers in constant time. The set of registers for which $\phi$-functions have to be created can then be computed in linear time by processing each bytecode instruction in the target node exactly

| Mnemonic / Syntax | Argument 1 | Argument 2 | Argument 3 |
|---|---|---|---|
| const-wide/32 $v_A$, C | Destination | – | – |
| move/from16 $v_A$, $v_B$ | Destination | Source | – |
| return-void | – | – | – |
| throw $v_A$ | Source | – | – |
| invoke-direct $\{v_A, ..\}$, C | Source | – | – |
| new-array $v_A$, $v_B$, C | Destination | Source | – |
| aget-boolean $v_A$, $v_B$, $v_C$ | Destination | Source | Source |
| aput-object $v_A$, $v_B$, $v_C$ | Source | Source | Source |
| mul-int $v_A$, $v_B$, $v_C$ | Destination | Source | Source |

Table 6.1: Register classification table

once. Phi functions are inserted at the very beginning of basic blocks. The number of arguments equals the number of predecessor nodes.

### Renaming registers

Once the bytecode has been instrumented, all registers have to be renamed in a final step. This is done in $O(n)$ whereas n is the total number of bytecode instructions in the application. In order to generate the SSA form, destination and source registers have to be renamed differently. Register types are looked up in the register classification table to determine which renaming scheme has to be applied.

As already explained in *Section 4.4*, registers that are assigned a new value, the destination registers, are renamed by appending an underscore along with a sequential numeric index. For each register a distinct index is kept on a per method level. Indices start with value one and are incremented each time a new definition of that register is encountered. In the process, register v3 will be renamed to v3_1 $\rightarrow$ v3_2 $\rightarrow$ v3_3 .. during this phase. Note that argument registers within the method header are always considered as a new definition.

Renaming source registers is slightly more difficult. While destination registers are renamed by using a global, per method index, source registers have to be renamed differently. If a register is renamed, the last definition for this register must be found on the current path. The target register is then assigned the name of the last definition. The search for matching definition registers usually requires traversing the CFG backwards, or at least traversing the already processed instructions in the current node backwards.

In order to speed up this computation, a local register map is generated during renaming of destination registers. The map stores the maximum index per register and basic block. By using this map, constant time lookups can be performed to avoid costly path traversals.

An anomaly of transforming register-based bytecode into the SSA form is, that in some cases it is not possible to rename certain argument registers of a phi function. In high-level programming languages, variables always have to be defined before they can be used. Registers in bytecode do not have to be defined explicitly. They are just used when necessary. Thus, it is possible that temporary values in one path are stored in a register that is not present in a different path (see also *Section 5.3.1*).

In such cases, no matching definition for the phi argument register can be found in the associated path. In order to mark these special locations, the register is renamed to *X*. If the resolver algorithm reaches such a phi function, the corresponding path is not processed.

## 6.3 Resolver Implementation

The theory behind the symbolic execution of dex bytecode has already been explained in *Section 5.4*. This section describes implementation specific details of the analysis algorithm.

During the parsing step, all sinks within the application have been located and their exact positions have been stored. However, due to the translation into the SSA form, some of these positions might have become incorrect. If phi functions have been inserted into basic blocks that contain sinks, the instruction indices of the sinks have to be updated. This is accomplished by adding the number of inserted phi functions to the instruction ID of the sinks. After updating all sinks, the resolver algorithm is able to access the instructions associated with the sinks.

Once the sink locations are verified, the actual backwards symbolic execution can be applied. Sinks are processed consecutively. The algorithm starts at the bytecode instruction associated with the sink and resolves all source registers. Instructions are processed according to the resolver semantics depicted in *Table 5.1*.

### 6.3.1 Loop Detection

During analysis, several kinds of loops can occur. In order to proceed with the analysis, several mechanisms have to be implemented to handle these loops and to provide information on how to resolve these situations.

One of the most important functions of the resolver is to find the matching definition for a given source register. When an instruction with arguments is processed, the next step usually includes the search for corresponding definitions of the argument registers. This is done by a path-sensitive backwards search of the application bytecode, starting at the current instruction. Path information is extracted from the previously created control flow graph.

In case that the CFG contains directed loops, it is possible that the tracing for a definition register ends in an infinite loop. Such a loop usually starts at a join node, if for example phi instruction arguments are resolved. If an infinite loop is detected during resolution of an argument, the algorithm stops, returns back to the phi instruction and continues its search in the path associated with the next argument register.

The actual loop detection is done by using a list of visited nodes. Once the tracing for a definition register starts, the visited list is cleared. It is updated each time a new node is visited and a loop detection warning is reported, if the current node is already in the visited list.

The simplest form of a loop includes only two bytecode instructions. It occurs when a for-loop, as shown in *Listing 6.1*, is processed. Internally, each loop is converted into a conditional-jump construct in dex bytecode. Thus, if the loop condition is true, the body is executed and in the end of the body a jump instruction puts the control flow back to the beginning of the loop.

```
String str = "someString";
for (int i = 1; i < 5; i++)
    str += i;
```

Listing 6.1: Simple for-loop

The corresponding bytecode in SSA form for the for-loop is depicted in *Bytecode Listing 6.3.* A phi instruction at the end of the loop indicates the two paths of the conditional. The *const*-instruction defines the initial value of the loop variable. Problematic is the integer addition instruction afterwards. It increments the loop variable by one for each iteration. Obviously, the source register of the addition instruction equals the destination register of the phi instruction and vice versa.

```
const/4 v0_1, 0x1
add-int/lit8 v0_3, v0_2, 0x1
phi v0_2, v0_1, v0_3
```

Bytecode Listing 6.3: Dex bytecode of a simple for-loop

Another rather rare case of a loop was encountered during resolution of a method argument register. The loop is formed by multiple phi instructions pointing at each other, that is phi argument registers pointing at the destination register of another phi function within the same method. *Figure 6.2* shows a phi-loop with register v2_5 as starting point encountered during register resolving.

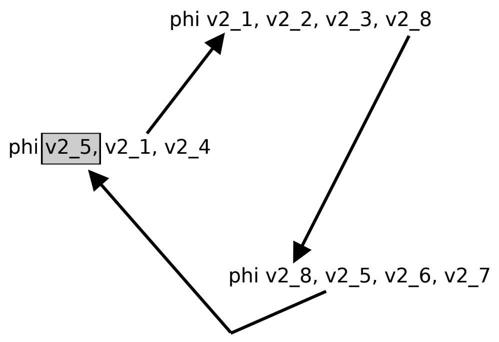


Figure 6.2: Phi loop during resolution of a method argument register

This kind of a loop is detected and prevented by a special phi visited list. The list is initialized each time a new method argument is resolved and updated each time a new phi argument register is processed. This is possible due to the unique naming of registers within a method. If a loop is detected, the algorithm terminates the current path and proceeds with the next argument of the last processed phi instruction.

Loops cannot only appear on register or node level. During testing, loops also appeared on method invocation level. Furthermore, this kind of loops can be categorized into two types, recursive method invocations and invocation chain loops.

Recursive method invocations can only be detected in custom method invocations since their bytecode is available for analysis. Recursive invocations are easily detectable. If a new invocation is reached, the corresponding method name is compared to the name of the currently analyzed method.

Recursive invocations must be treated differently in terms of resolving. In contrast to non-recursive custom method invocations, they cannot be reduced to a single value after all registers and the bytecode of the function have been resolved. The actual analysis starts with the resolution of method arguments. The recovered tokens encode the values of the initial invocation. Then, the method is resolved starting from its return instructions. Once the method header is reached, any unresolved function argument is replaced by a placeholder value. The recursive invocation within the function is treated like an API invocation in order to prevent a loop.

```
private int recursiveSum(int n) {
    return (n > 1)? n + recursiveSum(--n) : 1;
}
```

Listing 6.2: Recursive function to calculate the sum of the first $n$ natural numbers

*Listing 6.2* shows a simple recursive function to compute the sum of the first $n$ natural numbers. If it is invoked with a value x, the sum

$$x + \ldots + 2 + 1 = \sum_{i=1}^{x} n_i$$

is calculated recursively. Applied to this example, the analysis algorithm first reassembles the argument values that are passed to the initial invocation. Then, the function itself is resolved, whereas method argument registers are not resolved beyond the method border but instead replaced by markers. As a result, an invocation *recursiveSum(4)* is resolved as

4 + !ClassName;→recursiveSum($Arg1 - 1) .

Besides recursive method invocations, loops can also be formed by a sequence of method invocations that is repeatedly executed. However, sequences of invocations can appear multiple times in a program without forming a loop. In order to be able to distinguish real loops from normal function sequences, it is necessary to generate a method invocation chain. This chain does not only contain the current sequence of method invocations within the analysis, it does also provide information about the exact location of the invocations in the bytecode. The locations are again stored as 4-tuple consisting of class, method, node, and instruction identifier.

The invocation chain is a ordered list of location tuples, whereas the first element is the root method invocation and the last element is the current invocation. With each new element in the list, the algorithm enters a new recursion level. When a method is processed completely, the last element is deleted and the recursion level is decreased by one. The element at the current recursion level is updated once a new invocation is reached. This way, it is possible to keep track of the current invocation history.

Multiple executions of the same invocation sequence within an application does not necessarily create a loop. A sequence $method_A \rightarrow method_B \rightarrow method_C$ might be executed in different parts of the application. It is only considered as a loop if after $method_C$, $method_A$ is invoked at the same location as before. Thus, it is crucial to store the exact invocation locations rather than the method names.

Each time a new invocation is about to be processed, it is checked whether the current invocation chain includes the same method invocation at the same location. If a loop is detected, the algorithm stops and marks this path a dead-end.

### 6.3.2 Incorporating Expert Knowledge

The previous chapter introduced the concepts behind backwards symbolic execution and the corresponding approach of collecting and storing data tokens. Some of these approaches can be optimized by using expert knowledge. The concept of resolving closed-source method invocations and the generation of a call history (see *Section 5.4.3*) are candidates for optimization.

Closed-source method invocations are resolved by dumping the full method name and replacing the argument types by the resolved argument values. This forms a comprehensive representation as it includes any resolvable information. However, there is a more compact way of presenting the data in certain situations. By using expert knowledge, unnecessary method information may be omitted to improve readability.

In dex bytecode, strings are internally created and modified via the *String-Builder* class. Dumping the method and class name for a StringBuilder method invocation does not provide useful information. It is even unfavorable because it blows up the result without given additional value. Taking up the example of *Section 5.6.3*, the URL string *www.foo.com/?key=12345* is assembled via the StringBuilder function calls listed in *Table 6.2*.

The default algorithm generates a call history of these invocations and outputs

| Method invocation header | Resolved Argument |
|---|---|
| Ljava/lang/StringBuilder;→<init>(Ljava/lang/String;) | "http://www.foo.com/" |
| Ljava/lang/StringBuilder;→append(Ljava/lang/String;) | "?key=" |
| Ljava/lang/StringBuilder;→append(I) | 12345 |
| Ljava/lang/StringBuilder;→toString() | ∅ |

Table 6.2: StringBuilder invocation sequence

the following result (the line-wrap is due to the limited horizontal space):

Ljava/lang/StringBuilder; → <init>("http://www.foo.com")
→ append("?key=") → append(12345) → toString()

The result contains all resolved information and provides a comprehensive, but too complex representation of the string. In these cases, expert knowledge can be used to optimize string representations. It is known that the java.lang.StringBuilder class handles string values. Furthermore, it is known that class methods like the constructor or the *append* method are used to assemble a string value. The invocation order within the call history already represents the correct order of substrings. Thus, we can simply omit the class and method information, resolve the arguments and assemble the fragments in the correct order.

This approach can also be referred to as *resolve arguments only* approach. In the example above, this results in the raw string *http://www.foo.com/?key=12345*. Thus, using expert knowledge provides a more compact and readable output for strings created via the StringBuilder class.

The approach can also be used for certain other well-known methods. The set of methods for which expert knowledge can be applied to generate an improved output includes *toString* and *valueOf* methods of known Java classes. The optimization is restricted to widely used classes within the common *java.lang* package, like the Integer or Float class.

In general, *toString* methods take no arguments and return a string representation of the object on which they are invoked. Thus, the method itself does not provide any useful information. Resolving the object register will result in a string which can be directly incorporated into the associated set of tokens.

In Java, *valueOf* methods in the java.lang package usually take a primitive type or a string representation thereof and return the corresponding object type. The Integer.valueOf function, for example, takes an int argument and returns an Integer object. The analyzer does not make any difference between the primitive type and its corresponding object type. The resolved values are stored equally and are finally output in a string representation to the user. Therefore, the *resolve arguments only* approach can be applied in these cases as well.

### 6.3.3 Handling special Method Invocations

Functions often include invocations of other methods. This does not only complicate the backwards symbolic execution, it does also require a sophisticated and robust mechanism to find the associated method definition to proceed with the analysis. Contrary, if the resolver reaches a method header, a routine must find all locations in the application at which the current method is invoked.

By use of pre-generated lookup maps, these searches are quite simple and efficient. However, there are various situations at which no matching definition or method invocation is found. The following list includes a number of situations which require special handling in case the default search hasn't been successful:

- Super class method invocations
- Interface method invocations
- Resolution of Android event handler
- Class initializer invocations (constructors)

Any Java class inherits from the root base class *Object*. In order to implement specific concepts, it is often necessary to create a class hierarchy in which certain classes inherit from a base class. Thereby, the child classes may override methods of the base class in order to implement a custom behavior. If no custom version of a method is implemented in the child class, the corresponding function of the base class is executed.

As the bytecode dictates, the algorithm searches in the child class for the matching definition. If no implemented method is found, the resolver must check if the class inherits from a super class and must use this information to find the corresponding method definition. In case that the super class is a closed-source method, it is processed like any other API call.

In the Java programming language there is also a concept called interfaces. Interfaces are a kind of class skeletons. They provide class information and method header definitions without an actual implementation. Implementations can use interfaces like real classes without knowing implementation details.

If the resolver algorithm processes an invocation instruction that targets an interface method, it won't find the implementation in the associated interface. Instead, it must search for the implementation in classes that declare to implement the interface. In order to speed up this search, the parser module extracts interface information and generates a convenient lookup table during the parsing step.

Android event handler methods are considered entry points of the application. In contrast to default Java or C++ programs, there is no single entry point, like a main function, in Android applications. Event handlers are used to directly react to user input, for example if a button is clicked. The Android event handler method names are prefixed with *on* followed by the action that triggers the event, e.g. *onClick* or *onKeyDown*.

If the algorithm resolves an event handler function and reaches the function header, the resolver stops. Event handlers are usually invoked by the framework automatically. Further resolving, especially of function arguments, does not add useful information to the current result.

When class methods are reached, the algorithm does not only resolve the arguments. In case of API method invocations, a call history is created to provide information about method invocations on the instance that preceded the current invocation. The first invocation on an object instance is always the constructor call. In dex bytecode, this is the method initializer, denoted by <*init*>. As there is no preceding invocation, the algorithm stops here as well.

## 6.4 The Decoder Module

After the static analysis has been completed, there is a distinct target graph for each sink holding multiple sets of encoded tokens. The target graph is a directed, acyclic graph which is used for multi-path encoding. It represents a compact and space-efficient data structure to encode multiple paths with shared sub-paths. Each node in the graph holds a list of tokens that represents a fragment of the final output.

In order to decode the target graph and to dump all URLs associated with a certain sink, two steps have to be performed. In the first essential step, the graph is dissected into single results. This means that all paths through the graph, beginning at the root node and ending at a leaf node, have to be found. Furthermore, the path token lists have to be assembled by merging all token lists that belong to a node within the path. This merging step dismantles the shared representation of sub-paths. Each resulting path list holds a distinct copy of node tokens. Once the token lists are complete, they have to be decoded as described in *Section 5.6*. Decoded tokens of a path are subsequently assembled and finally output to the user.

In order to make the decoding efficient, the implementation must combine these steps rather than performing them consecutively. Dumping all paths first causes a waste of memory, especially on large graphs. Thus, it is meaningful to only store as much data as necessary. In the actual implementation, the decoding routine is triggered immediately, once a new path has been assembled completely.

Unfortunately, there is no better way for retrieving all paths, from a start node to a certain end node, than trying all possible combinations. A standard technique for performing this task is the backtracking approach. A backtracking algorithm traverses the graph recursively in depth-first order. It assembles the path by collecting the node indices that are encountered during traversal.

Default backtracking algorithms usually include various abort criteria such that partial solutions, which cannot lead to a valid end result, can be discarded early. This way, backtracking algorithms are usually faster than brute-force approaches. Since all paths through the DAG must be retrieved, reaching an end node is the only abort criterion. In fact, this represents a brute-force approach.

*Algorithm 6.4* shows the graph traversal via backtracking. It requires a current node, the recursion depth and a list that represents the current path. It performs a depth-first search and creates a path by collecting a list of node identifiers. Once an end node is reached, the token list associated with the path is assembled. Before the graph traversal proceeds, the current path is decoded to minimize the memory footprint. The initial invocation of the backtracking method includes the root node with index 0, a recursion depth of -1, and an empty list representing the path.

**Algorithm 6.4**: Graph traversal via backtracking

```
Require: int curNode, recursionDepth; list curPath
  recursionDepth++
  curPath[recursionDepth] ← curNode {update current path}

  if curNode is end node then {path is complete}
    pathTokenList ← [ ]
    for all nodes n on curPath do {assemble tokenlist for current path}
      append tokenlist of n to pathTokenList
    end for
    decodePath(pathTokenList)
    recursionDepth--
    return
  end if

  {backtrack all successor nodes}
  for all successor nodes s of curNode do
    backtrack(s, recursionDepth, curPath)
  end for

  curPath[recursionDepth--] ← ∅
  return
```

Intensive testing on large graphs showed that the recursive approach causes a high memory consumption. Recursive methods are often easier to implement because the stack handles the book-keeping of data during the recursive descent. This convenience is bought with an increased memory consumption, because of larger stack sizes. In order to optimize the graph decoding, an iterative graph traversal routine has been implemented additionally. The pseudo-code is depicted in *Algorithm 6.5*.

The iterative algorithm has the same abort criterion and path assembly than the backtracking approach. The crucial difference is that the algorithm explicitly manages a data structure to keep track of the nodes that still have to be processed. It mimics the task of the stack, but is more memory efficient, because no return data or local method data needs to be stored.

The data structure is called *worklist* and is a 2-dimensional list. It stores lists of successor nodes for all nodes on the current path. The *depth* variable holds the same information than the *recursionDepth* variable in the backtracking algorithm. The current node variable is always assigned the first element of the current successor set. The poll function does not only retrieve the element, but it additionally removes it from the set.

If a complete path is found, the algorithm assembles the token list and passes it to the *decodePath* function. The list is then decoded and evaluated and the final result is stored. Afterwards, the algorithm mimics the backtracking behaviour by moving backwards in the *worklist* as long as it finds empty successor sets.

**Algorithm 6.5**: Iterative graph traversal

```
depth ← 0
worklist ← [[0]]

while worklist $\neq \emptyset$ do
   curNode ← pollFirst(worklist[depth])
   curPath[depth] ← curNode {update current path}

   if curNode is end node then
      pathTokenList ← [ ]
      for all nodes n on curPath do {assemble tokenlist for current path}
         append tokenlist of n to pathTokenList
      end for
      decodePath(pathTokenList)

      {move back to the last path node that has successors to process}
      while worklist[depth] = $\emptyset$ do
         remove worklist[depth]
         if depth = 0 then
            break
         end if
         remove curPath[depth]
         depth--
      end while

      if depth = 0 then
         return {root node reached, done}
      end if
   else
      depth++
      worklist[depth] ← list of successor nodes of curNode
   end if
end while
```

Once a non-empty set is found, the algorithm continues the search for new paths. Finally, the algorithm stops when the root node is reached.

# Chapter 7

# Evaluation

The last chapter described implementation specific details of the analysis tool. This chapter is about the evaluation of the approach. It starts by showing the results of the analyzer on common programming paradigms, like conditionals and loops. With the help of small Java code snippets, concrete examples are presented. Certain parts of the code are highlighted to describe areas that are difficult or special to analyze.

Additional examples show more complex code including class field and array accesses. They require advanced analysis techniques, like the concepts of array reconstruction or the path-sensitive resolution. In order to show the different types of method invocations, dedicated examples are presented for custom methods, their special subtype, recursive invocations, and closed-source method invocations. Concepts like the generation of a call history or the special dumping of recursive invocations are explained with the help of descriptive examples. Finally, the analysis of a more complex Android application is presented. It covers the resolution of data sinks across classes and includes various data structures that need to be analysed in order to output a correct result.

The following sections contain several Java code examples that are used to show the effectiveness of the static analysis approach. Each example includes programming paradigms that are common in real applications. The code parts are kept at a minimum size such that important aspects can be illustrated without requiring a larger context. For the sake of simplicity, all examples use the URL constructor data sink. Furthermore, the examples are presented in Java code for readability and understanding reasons. However, it should be noted, that the actual analysis is performed on the dex bytecode, which is generated when the application is translated with the dx compiler.

## 7.1 Resolving Basic Constructs

This section shows how basic constructs like expressions or conditionals are resolved by the algorithm. The examples are presented in form of small Java functions that return a URL instance. The associated bytecode instruction is the target data sink to be resolved.

Listing 7.1 shows a URL object that is assembled by means of strings and expressions. The expressions include an subtraction of two float values and an integer addition of a numeric value with a return value of a method. In this example, *TestClass* is a class of some official API for which the source code is not available. A new *TestClass* instance is created and initialized. The *getValue()* method simply returns the integer value which has been used in the constructor.

```
private URL calc() {
    float x = 1.43f;
    float y = 1.174f;
    int   z = 5 + new TestClass(17).getValue();
    String dom = "www.foo.com/?val=";
    return new URL(dom + (x-y) + "&val2=" + z);
}
```

Listing 7.1: Simple calculations

The result of the analysis is shown in *Output 1.* In order to produce this result several tasks have to be performed. The URL argument string is assembled by processing the fragments in left to right order. Thereby, the variables are resolved until constant values are found. In order to evaluate the expressions, the numeric values have to be converted first. In dex bytecode all numbers are stored in a hexadecimal representation, e.g. the hexadecimal value of 1.43f is 0x3fb70a3d. They are converted according to their data type, before the operation is performed.

```
www.foo.com/?val=0.25599992&val2=$(EXPR1)
  $(EXPR1): $$((int) !Lorg/test/TestClass;-><init>(17);->getValue() + 0x5)
```

Output 1: Assembled URL string

As both operands of the float subtraction are real values, the expression is evaluated and the result is output. The small loss of significance is a general problem that occurs when two close floating point numbers are subtracted.

The second expression is not resolvable for the static analyzer, as the second operand is not a numeric value. The return value of the function can also not be inferred since no context or source code is available. In this case, the algorithm uses a default template for expressions to provide the most comprehensive set of information possible. The template also includes the data type of the operation result. Furthermore, the full class name along with the constructor call and the final method invocation are output.

For improved readability, unresolvable expressions are replaced by a simple placeholder in the final result. The complete expression is then printed in an extra line.

### Conditionals

*Listing 7.2* shows a nested conditional that needs to be resolved in order to output the URL strings. The phi instruction that is placed right behind the outermost if-conditional indicates three distinct paths. Two of these paths assign a new value to the *dom* variable, which contains the core part of the domain. The third path only includes an expression that is of no importance for the final URL. In this path, the depth-first search will therefore find the initial assignment at the very beginning of the method.

```java
private URL nestedConditional() {
    String dom = "foo.com/init";
    int i = 5;

    if (i < 10)
        dom = "foo.com/then";
    else {
        if (i < 5)
            i = i*i;
        else
            dom = "foo.com/else_else";
    }

    String domain = "www." + dom;
    return new URL("http://" + domain + "?noArgs");
}
```

Listing 7.2: Nested conditional

This initial assignment is also part of the other two paths. However, in backwards symbolic execution only the first match will reach the data sink and thus the algorithm will stop there. The three paths through the nested conditional also produce three URL values, which are depicted in *Output 2*. Note, that in this concrete example only the third output reaches the sink if the method is invoked. In general, evaluating conditions is complex and infeasible in many situations. The current algorithm does not perform reachability analysis by evaluating conditions and therefore outputs one result for each path.

```
1. http://www.foo.com/then?noArgs
2. http://www.foo.com/init?noArgs
3. http://www.foo.com/else_else?noArgs
```

Output 2: Result for the nested conditional

## 7.2 Evaluating Class Fields

Resolving a class field access is a more complex task. Various factors have to be considered for the analysis. The algorithm must check whether the target class field is a static or an instance field. In addition, field accesses often need to be resolved for multiple paths, i.e. the resolution has to be path-sensitive. If the field access can not be resolved within the current method, the algorithm extends its search to other parts of the program, depending on the field access type.

*Listing 7.3* shows a slightly more complex code example to describe the different types of class field accesses. The code introduces a class *ClassFieldRecovery* with a static field of type string and an instance field of type integer. The class constructor invokes the class method *target* which contains a URL sink. Furthermore, the class contains a nested class with a single public class field.

```
public class ClassFieldRecovery {
    static String staticVal = "C";
    public int instanceVal  = 555;

    private class nestedClass {
        public int item;
        nestedClass() {
            item = 5;
        }
    }

    ClassFieldRecovery(){
        target("www.foo.com");
    }

    private URL target(String domain) {
        nestedClass nc = new nestedClass();
        nc.item = 10;
        nestedClass nc2 = new nestedClass();

        int x = 24;
        if (x < 35) {
            staticVal = "A";
        } else {
            staticVal = "B1";
            this.instanceVal = 30;
            nc.item = 50;
            staticVal = "B2";
        }

        String args = "?staticVal=" + staticVal +
                      "&instanceVal=" + this.instanceVal +
                      "&pair=" + nc.item + "-" + nc2.item;
        return new URL("http://" + domain + args);
    }
}
```

Listing 7.3: Class field resolution

Resolution of the class starts at the URL class instantiation. The *domain* variable is a method argument that is not modified within the function. Therefore, the algorithm searches for invocations of the *target* method. The only invocation is found in the class constructor. The passed value for the domain argument is stored and the analysis proceeds.

The interesting part is the *args* string that is assembled by using various class field values. First, the value of the static field is accessed. It turns out that the assignments to this field can be resolved within the conditional. The else-branch contains two assignments, whereas only the latter one will reach the data sink.

Next, the value of the instance class field is accessed. The current class instance is referenced by the *this* variable. The class field *instanceVal* is assigned the value *30* in the else-branch. Resolving the access in the then-branch isn't successful. The field is also not set in the remaining path towards the method header. If this is the case, all class field initializations have to be parsed. These initializations are stored in a special class field annotation that is not parsed in the current implementation. Therefore, the algorithm doesn't output a valid value for this path.

The last fragment of the *args* string is a pair of values. The variables reference the public class field *item* of the nested class instances. These objects are created at the very beginning of the method. The field of the first instance is only set in the else-branch. However, tracing the access through the then-path will lead to the assignment right after the instantiation. The concrete class field value of the second instance cannot be determined within the *target* method. The field value is initialized in the constructor of the nested class. By analyzing this initializer method, the value *5* can be recovered for the second class instance.

Combining all resolved data chunks for the URL sink generates the results shown in *Output 3*. Obviously, the analysis overestimated the real results. Although there are only two distinct paths through the CFG of the *target* method, the output shows four URLs. This overestimation originates from the target graph generation algorithm (see also *Section 5.5.4*). The resolution of the *staticVal* and the *nc.item* variable results in two distinct target graphs. Each graph contains the same two paths through the if-conditional. But since the graphs are merged, the resulting target graph contains four distinct paths. This leads to the overestimation that produces the first and fourth URL of the output. The incorrect *instanceVal* values for the first two URLs originate from the missing parsing support for class field annotations.

```
1. http://www.foo.com?staticVal=A&instanceVal=30&pair=50-5
2. http://www.foo.com?staticVal=A&instanceVal=30&pair=10-5
3. http://www.foo.com?staticVal=B2&instanceVal=30&pair=50-5
4. http://www.foo.com?staticVal=B2&instanceVal=30&pair=10-5
```

Output 3: Result for the class-field example

## 7.3 Evaluating Arrays

The analysis of arrays includes the evaluation of array accesses as well as the reconstruction of array elements at a certain point in time. In general, resolving an array access also requires the full reconstruction of the array content. It is unknown in advance whether the index is reducible to a numeric value such that the access can be fully resolved.

In order to show the effectiveness of the approach, *Listing 7.4* includes various array accesses with constant and dynamic indices. Again, a single URL data sink is included. It is instantiated with a domain and an argument string is appended with values from an integer array.

```
private URL arrayAccess() {
    int i = 4;
    int[] intArray = { 12, 13, 14, 15, 16 };

    if (i < 10)
        intArray[1] = 35;
    else
        intArray[1] = 10;

    intArray[0] = 1;
    intArray[2] = i * 5;

    String arg = "?key=" + intArray[i-2] +
                 "-" + (intArray[1] + intArray[3]);
    return new URL("http://www.foo.com/" + arg);
}
```

Listing 7.4: Array access and reconstruction

The first value of the key pair is an array element that is indexed by an expression. The expression is resolved normally and the result is encoded and stored. Afterwards, the elements of the array are reconstructed. Therefore, all array modifying instructions are evaluated within the code from the actual array access to the array definition.

Since the conditional modifies certain elements differently, the array reconstruction will produce two results. The fully resolved array contents, at the time of the access, are [1, 35, 20, 15, 16] and [1, 10, 20, 15, 16]. Only the second element differs. The actual value depends on the variable used in the if-condition.

During decoding, the array access is evaluated. The index expression targets the array element at index two which is again an expression. The multiplication is evaluated and the result is returned.

The second value of the key pair is an addition of two array elements. The indices are constant values and thus no further resolution is necessary. The array elements are reconstructed as described before. Since different values for the second element are possible at the time of the access, there are two distinct results for the addition. If the then-branch of the conditional has been executed, the operands of the addition are 35 and 15. Otherwise, the value 35 is replaced by 10. Finally, the fully resolved and assembled URLs are depicted in *Output 4*.

```
1. http://www.foo.com/?key=20-25
2. http://www.foo.com/?key=20-50
```

Output 4: Results of the array accesses

## 7.4 Evaluating Method Invocations

Resolving and evaluating method invocations is a complex area. There are different kinds of invocations and the level of analysis applicable strongly depends on the availability of the method source code. If the functionality is known to the analyzer, much more compact and precise results are possible. On the other side, if closed-source functions are invoked, special concepts like the call history are used to give the user comprehensive information about the context of the invocation.

In the following, code examples for different kinds of invocations are shown to highlight both the general analysis approach and special concepts that are applied if only limited information about a method is known.

### 7.4.1 Call Histories

*Listing 7.5* shows an example with closed-source method invocations. Thereby, a call history is generated to provide more context to the user than normal resolution would provide. The method also includes a URL sink where the actual URL string is assembled via several fragments.

```
private URL callHistory() {
    File f = File.createTempFile("test", ".txt");
    Float size = 1.235f;
    String fargs = "?file=" + f.getAbsolutePath() +
                   "&size=" + size;
    String domain = "www.foo.com/upload";

    BigInteger bi = new BigInteger("123456");
    bi.add(new BigInteger("77777"));
    return new URL("http://" + domain + fargs
                          + "&id=" + bi.toString());
}
```

Listing 7.5: Call history example

A part of the argument string uses a file instance that is created by invoking the static method *createTempFile* of the java.io.File class. This is similar to creating a new object and initializing it with some values. The file argument string *fargs* is built up of the absolute path of the instance, which is retrieved by invoking the corresponding method. Furthermore, the file size is appended as a floating point value.

The last parameter *id* uses a string representation of a BigInteger class instance that has been created before. In a subsequent class method invocation its value is modified. Finally, the current value is converted to a string representation by invoking the *toString* method.

The results of the analysis are shown in *Output 5*. They clearly show the benefit of generating a call history. With normal backwards symbolic execution, only the method invocations that directly reach the URL sink would have been dumped. In this example, only the final object invocations would have been output; *Ljava/io/File;→getAbsolutePath()* for the File object and *Ljava/math/BigInteger;→toString()* for the BigInteger object. Certainly, this gives some information, but the crucial part necessary to understand the big picture is missing.

```
http://www.foo.com/upload?file=$(FNC1)&size=1.235&id=$(FNC2)
   $(FNC1): Ljava/io/File;->createTempFile("test", ".txt");
                              ->getAbsolutePath()
   $(FNC2): Ljava/math/BigInteger;-><init>("123456");
                                      ->add("77777");->toString()
```

Output 5: The resulting call histories

By generating a call history, all invocations on the target object are recorded until the object definition is reached. Then, this information is assembled in chronological order. This provides the most comprehensive information possible without having the actual source code of the function. In final results, call histories are usually replaced by a marker and printed in a dedicated line for readability reasons. In *Output 5* lines are wrapped due to space limitations.

### 7.4.2 Fully Resolvable Methods

Methods that are defined and implemented within the application can be fully resolved by the analyzer. The functionality can be reconstructed by parsing the bytecode. Expressions and instructions within the function can be invoked symbolically. Evaluation can be performed by replacing variables with either concrete or symbolic values. The resulting output shows the full power of the symbolic execution approach.

*Listing 7.6* shows two custom methods that implement some calculations of input values that reach a data sink. The main function is the *target* method that includes a URL sink. The argument string, appended to the domain, contains a single argument whose actual value is the return value of the function *calc*.

```
private URL target() {
    return new URL("http://www.foo.com/?val=" + calc(7));
}

private int calc(int input) {
    int i = 5;
    if (input < i) {
        return i+5;
    } else {
        if (i*input > 60)
            return input-i;
        else {
            i *= input;
        }
    }
    return calc2(i-input, 5);
}

private int calc2(int input1, int input2) {
    int result = input1 + (input1 * input2);
    return (result / 3);
}
```

Listing 7.6: Full resolving of custom methods

In total, there are three distinct paths through the *calc* function. The first two end by returning a simple expression. The third path through both else-branches is slightly different. It does not only include expressions, but also another method invocation that changes the value to be returned. Evaluating this path does shows the effectiveness of the expression resolver.

The first two paths directly end within the conditional of the *calc* method. The first return value does not depend on input arguments and is thus easily resolvable. The second path returns an integer value which is the result of an expression that includes a method parameter. Resolving this expression requires the concrete argument value that is passed to the method invocation.

The third path is more complex than the previous ones. All tokens collected during the analysis are stored and encoded properly. This includes the encoding of the custom methods, expressions, and constant values. Once the methods are processed, the decoding routine starts and the data evaluation is performed.

For convenience, the single steps through the path are highlighted and described in more detail in *Table 7.1*.

| Instruction | Calculation |
|---|---|
| calc(7) | |
| i = 5 | |
| i = i * input | 5 * 7 = 35 |
| return calc2(i-input, 5) | calc2(35-7, 5) → calc2(28, 5) |
| calc2(28, 5) | |
| result = input1 + (input1 * input2) | 28 + (28 * 5) = 168 |
| return (result / 3) | 168 / 3 = 56 |

Table 7.1: Evaluation of custom method invocations

Finally, the results for the analysis of the URL data sink in function *target* are shown in *Output 6*.

```
1. http://www.foo.com/?val=10
2. http://www.foo.com/?val=2
3. http://www.foo.com/?val=56
```

Output 6: The results after evaluating the expressions and custom method calls

### 7.4.3 Recursive Method Invocations

Recursive method invocations are a special subset of custom method invocations. A method is considered recursive, if it is invoked within its implementation. The analysis of recursive methods is pretty difficult and the invocations have to be handled similar to loops within a method. The arguments that are passed to the invocation are modified within the function. Furthermore, the function includes some abort criterion which stops the recursion. In contrast to a loop, in which all loop relevant data is stored within one block, information that targets the recursion might be distributed all over the function.

In general, it is not possible to fully resolve and evaluate a recursive function with static analysis. Information about the modifications of method arguments and abort criteria might be very complex and not resolvable. Therefore, special handling is necessary to approximate the concrete behavior of the method. The goal is to output the most comprehensive information possible, as it is done for closed-source method invocations.

Recursive methods are resolved like normal custom methods. Their recursive invocation however is treated like an API call. This means, that the whole method functionality is analyzed and evaluated by using the arguments passed to the initial invocation. The recursive call within the function is not followed, as this causes a loop which cannot be handled effectively. In general, information about the abort criteria is unknown and might only be fully resolvable for a limited number of cases.

*Listing 7.7* shows a simple function with a URL sink. A fragment of the URL argument string is assembled with the return value of a recursive function. Actually, the value results of a multiplication of a constant value with the outcome of the recursive invocation. The function *recSum* recursively calculates the sum of the numbers within the interval [*input*, 2], whereas *input* denotes the value passed to the initial invocation.

```
private URL target() {
    int x = 17;
    return new URL("http://www.foo.com/?val=" + (x * recSum(7)));
}

private int recSum(int input) {
    return input > 2? input + recSum(--input) : 2;
}
```

Listing 7.7: A recursive method invocation

The backwards symbolic execution finds two paths in the function *recSum*. The first path is executed if the recursion proceeds, that is if the input value is larger than two. The else-branch is taken if the recursion stops. In this case, the value two is returned. Because of the two paths, the analysis outputs two different results. The second URL in *Output 7* appears if the initial argument value is smaller than the abort threshold. This is also the only case in which the method is not invoked recursively.

```
1. http://www.foo.com/?val=$(EXPR1)
     $(EXPR1): $$((int) 17 * $$((int) 7 +
                          TestClass;->recSum($$((int) $ARG1 + -0x1))))
2. http://www.foo.com/?val=34
```

Output 7: The results for the recursive method invocation

The situation becomes more complex if the other path is executed. In this case the *then*-branch is analyzed normally, with the exception of the recursive invocation. In order to prevent a loop, this special invocation is handled like a closed-source method call. The method argument passed to this invocation cannot be resolved, and is thus replaced by a marker *$ARG1* that indicates its number within the argument list.

The expression of the first resulting URL shows the multiplication of two integer values of the root expression. The second operand is the return value of the invocation. Since it is not fully resolved, the template provides information about the functionality of the method. The integer addition starts with the initial value seven. In each subsequent invocation the first argument is decremented by one. The abort criterion is only shown indirectly by the value in the second result. Due to space limitations the full package name of the *TestClass* is omitted and a line break is inserted into the expression line.

## 7.5 The Translation App

In order to show the effectiveness of the approach on real Android applications, a simple translation app was implemented. The application translates an English text entered by a user into a selected destination language. The actual translation is done by use of the Google Translate API[1]. The result is then presented to the user.

The application comprises two classes. The main class *Translate* is an Activity that shows up when the application is started. It defines all UI widgets, like the EditText for the text to be translated and a Spinner/ComboBox to select the destination language. Furthermore, a TextView shows the results of the translation.

The translation is triggered implicitly, if the text input does not change within a small time frame that starts right after the last modification. The actual translation is performed asynchronously in a distinct thread, such that the GUI is not blocked while the application waits for the result of the Google servers. The translation thread is implemented in a second class *TranslateJob*. Its constructor is invoked with the user input extracted from the UI widgets. Using this data, the actual request is assembled. Then, a connection to the translate service is established and the data is transmitted. The translated data is then received in JSON[2] format. The JSON object is parsed and the translated text is returned to the GUI thread which subsequently updates the screen.

In the following, the analysis relevant parts of the application are presented in code listings. Thereby, the code is described in the order in which the backwards symbolic execution processes the code parts. It starts with the main translation method *doTranslation*, goes on to the *run* method of the translation thread, and finally ends with a method in the GUI thread that collects all necessary information and starts a translation thread.

*Listing 7.8* shows the main translation method. It gets the original text as well as the source and destination language to be used for the translation. The source language has been hard-coded to English for all translations, whereas the destination language is specified by the user. The input arguments are used to

[1]The Google Translate API v1 was officially deprecated on May 26, 2011. In a recent statement it was announced that the API will be shut off completely on December 1, 2011. The Translate API v2 is unfortunately only available as paid service. (see also `http://code.google.com/apis/language/translate/v1/getting_started.html`)

[2]The JavaScript Object Notation is a lightweight data-interchange format.

create the actual request. To ensure correct transmission and processing, the text has to be URL encoded. Thereby, the default *UTF-8* character set is used. Source and destination language are separated by the pipe operator (converted to *7C* in URL-encoding). The translation API is finally accessed by using the domain *http://ajax.googleapis.com/ajax/services/language/translate* .

```
private String doTranslation(String origText, String src, String
    dst) {
     String result;
     HttpURLConnection con = null;

     try {
         // Build query for Translation API
         String query = URLEncoder.encode(origText, "UTF-8");

         URL url = new URL(
             "http://ajax.googleapis.com/ajax/services/language/
                 translate"
                 + "?v=1.0"
                 + "&q=" + query
                 + "&langpair=" + src + "%7C" + dst);

         con = (HttpURLConnection) url.openConnection();
         con.setRequestMethod("GET");
         con.setConnectTimeout(10000); // 10 seconds
         con.setDoInput(true);
         con.addRequestProperty("Referer", "http://foo.com");
         con.connect(); // Start the request

         // Read and parse results from the query
         // Set result variable to be returned
         [...]
    } catch(Exception e) {
         result = "An error ocurred during translation.";
    } finally {
         if (con != null) con.disconnect();
    }

    return result;
}
```

Listing 7.8: Main translation method

The URL object is then used to create a *HttpURLConncection*. Before the connection is established, some connection parameters have to be set, as it is not possible to change them after the connection is active. The connection properties include the request method, the connection timeout, and a property to allow input for the connection. Furthermore, the API requires a valid *referer* header. Once the parameters are set, the connection can be established to transmit the data. The remaining part of the listing includes parsing and returning the result of the translation.

During the parsing of the application bytecode, the URL sink gets stored and marked as starting point for the backwards symbolic execution. The constant strings within the method can be retrieved directly, whereas the resolution of method arguments requires additional analysis effort. The variables for source and destination language can be traced back to the method header directly.

The query parameter contains the original text that is passed to a static encode function of the URLEncoder class.

The *doTranslation* method is invoked in the *run* method of the TranslateJob class. The run function usually contains any code that is to be executed in a thread. In this case, it contains a call to the translation routine. All arguments passed to the function have been set previously in the constructor of the TranslateJob class. The result is then used to invoke a method of the Activity, which performs a screen update to display the translation to the user. *Listing 7.9* depicts the code for the run method.

```
public void run() {
    // Perform translation and call screen update
    String translation = doTranslation(this.original, this.src,
        this.dst);
    translate.setTranslated(translation);
}
```

Listing 7.9: Run method of the translation thread

In order to proceed with the analysis, the algorithm searches the application for any instantiation of the TranslateJob class. A translate job is assembled and executed in a dedicated thread within the *Translate* class. *Listing 7.10* shows the corresponding part of the source code. First, the user input is retrieved from the UI widgets. Any pending translation task is cancelled, before a new one is started. For performance reasons, a new translation is only invoked, if there is any text. A new instance of the TranslateJob class is created and the thread is executed. The analysis proceeds with the constructor invocation within the try-block. The original text and the destination language are traced back to their corresponding widgets *origText* and *dstLanguage*. Before the actual widget definitions are reached, multiple method invocations are executed to convert and modify the respective values.

The results of the analysis are presented in *Output 8*. The URL has been assembled and call histories are generated to provide context information. The histories are output in an extra line to improve readability.

```
http://ajax.googleapis.com/ajax/services/language/translate?
   v=1.0&q=$(FNC1)&langpair=en%7C$(FNC2)

   $(FNC1): Ljava/net/URLEncoder;->encode
                (Landroid/widget/EditText;->getText()
                    =>Landroid/text/Editable;->toString()
                    =>Ljava/lang/String;->trim(),
                 "UTF-8")
   $(FNC2): Landroid/widget/Spinner;->getSelectedItem()
```

Output 8: Resulting URL for the Translation app

Fragments like the source language code are constant strings which can be directly incorporated into the URL string. The destination language, as well as the text to be translated originate from user input. In order to provide comprehensive context information, call histories are generated by the decoder module.

```java
translateTask = new Runnable() {
   public void run() {
      // Get user input
      String inputText = origText.getText().toString().trim();
      String lang = dstLanguage.getSelectedItem().toString();

      // Cancel previous translation if there was one
      if (transPending != null)
          transPending.cancel(true);

      // Translate only if there is any text
      if (inputText.length() > 0) {
         // Inform the user, that the translation just started
         transText.setText("Translating...");

         try {
            TranslateJob translateJob = new TranslateJob(
                  this,             // Translate activity
                  inputText,        // original text
                  "en",             // source language
                  lang              // destination language
            );
            transPending = transThread.submit(translateJob);
         } catch (RejectedExecutionException e) {
            transText.setText("The translation couldn't be
                initiated!");
         }
      }
   }
};
```

Listing 7.10: Translate Task

*FNC1* is the marker for the analysis result of the original text part. As described before, the URL-encoded user input reaches the actual URL that is used for the request. The static *encode* method takes two arguments, the string to be encoded and the charset. The source of the text is the *EditText* widget of the user interface. A series of method invocations is used to retrieve the final string that is encoded.

Here, a verbose version of the call history is used. It includes the full class name for each method invocation. This is done, if the class names within a history do not match. In this example, the *getText* function returns an Editable instance on which the *toString* method is applied. Finally, the *trim* function is applied on the resulting String object.

The second marker, *FNC2*, shows the origin for the destination language part of the URL. This is the currently selected value of the ComboBox[3], which is retrieved by the *getSelectedItem* method.

The final *toString* invocation is stripped, because of the expert knowledge used in the resolver module. The method returns an Object instance. The Object class is the root class and is part of the *java.lang* package. Therefore, the method can be stripped without loosing information (see also *Section 6.3.2*).

[3]In Android the Spinner widget denotes a common ComboBox, which is a selection list.

# Chapter 8

# Related Work

With the growing popularity of the Android platform, there has also been a lot of research on how to improve the system security and, in particular, on how to protect unauthorized leakage of sensitive user data. In the last two years, there has been a rising number of frameworks and tools that improve certain security aspects by either leveraging the Android core system, by implementing some user-space protection, or by providing offline certification tools to analyze applications on a large scale. The Android research can be partitioned into the conceptual improvement of the core operating system and the detection of malware and information leakage.

Since the Android market does not apply application vetting, a core security part relies on the permission model. However, the concept of the permission model has some severe limitations like for example the missing support for runtime permissions. Due to this inflexibility, there has been active research on how to extend and modify Android's permission framework to provide more flexibility and security. Enck *et al.* [28, 29] have built a tool called Kirin to perform a lightweight certification of Android applications. Permissions are extracted from the manifest file and compared to a manually created list of potentially dangerous combinations of permissions. In a different work, they present Saint [47], a tool that allows application developers to define install-time and runtime constraints. Nauman *et al.* extended the permission model [43] to allow users to define fine-grained permission constraints at install-time and to modify/revoke permissions at runtime. MockDroid [7] is an approach to allow the user to mock an application's access to a specific resource. Thereby, it is possible to revoke access to particular resources dynamically at runtime. A current approach called TISSA [64] implements a privacy mode for Android. The user can specify access policies for private data, like the address book, for each application individually.

The remaining part of this chapter puts the focus on analysis approaches to detect malware and information leakage. The concrete goals of these approaches are manifold. The number of malicious apps that leak sensitive user data without the user's explicit consent is rising quickly. Therefore, the detection and prevention of such leaks becomes more and more important. Other approaches target malware like online-banking trojans or applications that silently send premium SMS to foreign countries.

The approaches presented in the following sections are based on static and dynamic analysis. Dynamic analysis usually means runtime analysis that is applied during execution of the application/system. Static analysis approaches are often used for offline certification and are thus suitable to be applied on many applications simultaneously.

## 8.1 Dynamic Analysis

Bos *et al.* present a dynamic Android security system in the cloud [50]. They argue that many security checks that are common on modern computers cannot be applied on resource limited smartphones. Therefore, they devised a solution called Paranoid Android, which outsources resource-intensive security checks like detection of zero-day attacks and anti-virus scanning to servers in the cloud. Their system is composed of a server and a client part. The client part that resides on the smartphone stores an execution trace of the current actions and transmits it securely to remote servers. The server part includes virtual machines that replay the actions read from the execution trace and apply the security checks. Thereby, each smartphone has an exact virtual replica on the remote servers.

In 2010, Enck *et al.* presented TaintDroid, a dynamic information-flow analysis system for Android [25, 26]. It leverages Android's runtime environment to provide realtime data flow analysis. The approach is used to detect privacy leaks in applications. Data originating from privacy-sensitive sources are tainted automatically by the system. Thereby, several kinds of sources are monitored, including private data providers, files, and interprocess-communication. TaintDroid tracks sensitive data as it propagates through the program during execution. Each time labeled data reaches a data sink and leaves the phone, for example via network or Bluetooth, the user is informed. Besides the concrete values that are transmitted, TaintDroid also logs information about the destination.

In order to provide a system-wide taint-tracking system, the Android source had to be instrumented. They state, that TaintDroid incurs an average of 14% performance overhead during runtime. Intensive testing on real applications resulted in a minimal perceived latency. A fundamental limitation of the approach is that TaintDroid does not track control or implicit flows due to performance issues. They claim that the use of implicit flows to avoid taint detection is itself a strong indicator for malicious intent.

Hornyack *et al.* devised a dynamic privacy control system called AppFence [38]. It is composed of two mechanisms to control application usage of sensitive user data. The first mechanism implements data shadowing. It prevents applications from accessing sensitive data that is not required for its core functionality. AppFence provides mock data instead of real data originating from sensitive data provider.

The control system additionally includes an exfiltration blocking feature. Outgoing network communication is blocked, if it includes sensitive user data that is not essential to ensure the application's functionality. This part of the system is based on TaintDroid which provides the tracking of tainted data from predefined sources to network sinks. In order to test which sensitive data can be blocked without loosing functionality, they devised an automated testing methodology. During application runtime, screenshots are taken both with and without enabled privacy control. Any visual difference in the screenshots indicates a change in application behaviour which potentially causes side-effects. With manual tweaking of control settings, these side-effects could often be minimized or avoided.

XManDroid [13] is a security framework that targets the mitigation of privilege escalation attacks. It extends Android's monitoring mechanism to effectively detect and prevent application-level privilege escalation attacks like Soundcomber [58] at runtime. The system dynamically analyzes the transitive permission usage of applications. It monitors the inter-process communication to detect malicious requests. XManDroid thereby maintains information about all applications installed on the system to decide whether certain calls can be exploited for escalation attacks.

## 8.2 Static Analysis

ComDroid [15] is a static analysis tool that targets inter-process communication (IPC). Message exchange in IPC is a potential attack surface in Android applications. Sensitive content that is sent via communication channels may be intercepted, modified and/or forged by malicious applications. ComDroid performs static analysis on dex bytecode to identify security risks in application components. It parses the output of the open-source disassembler dedexer to warn the user of potential component and Intent vulnerabilities.

In a follow-up work, Chin *et al.* investigate whether developers follow the principle of least privilege when specifying the permissions for their applications. In order to detect overprivileged Android applications, they implemented the static analysis tool Stowaway [30]. It uses a previously generated permission map to identify which permissions are required for API calls. Stowaway then extracts any API call that is used within an application and creates a minimal set of permissions that is required to ensure functionality. Any difference to the set of specified permissions from the application's manifest file indicates an overprivilege.

By evaluating 940 applications, Chin *et al.* found out that about one third of the analyzed apps are overprivileged. Furthermore, they criticize that the official Android API documentation is incomplete and does not provide sufficient information about which classes/methods require which permissions. They showed that, in general, overprivileged applications only contain a few dispensable permissions. Furthermore, the main reasons for incorrect permission settings are the incomplete API documentation and the lack of developer understanding.

Christin *et al.* took on the problem of overprivileged applications from a developer view [62]. Permissions in Android are granted at install time and are then silently enforced at execution time. For developers, it is usually very difficult to set the appropriate permissions that are necessary for an application. The Android API provides almost no information about which methods require a certain permission. An insufficient permission set does only become apparent during runtime when the application is executed and a certain functionality fails. Therefore, many developers overestimate permissions to ensure that the application directly works as intended.

To help developers specifying the correct set of permissions before deploying, Christin *et al.* developed an Eclipse plugin which inspects the Java source files and subsequently generates a list of required permissions. This list is adjusted with the permissions specified in the manifest file to inform the developer in case of missing permissions or if the application is overprivileged.

The work of Grace *et al.* [35] targets the detection of capability leaks in Android smartphones. Capability leak means that an untrusted application can obtain unauthorized access to sensitive data and/or privileged actions without having granted the respective permissions. Often, unsafely exposed interfaces and functionality of already installed apps are exploited to gain access to privileged functionality.

In order to detect leaked capabilities, they implemented a system called Woodpecker. It employs inter-procedural data-flow analysis on applications to detect unauthorized access to private data. Woodpecker distinguishes between two types of capability leaks, explicit and implicit leaks. Explicit leaks allow an application to gain certain permissions by exploiting public interfaces without requesting the permission itself. Recently, Backes *et al.* [4] were able to mount a local XSS attack by exploiting an explicit leak in the default Android web browser. Implicit leaks, however, exploit interfaces and services of another application that has the same signing key. Such apps share the same user ID and thus an application inherits the permissions of other applications from the same developer.

Chaudhuri *et al.* present SCanDroid [31], a static analyzer for Android applications. It extracts specifications about components and permissions from the manifest file. This information is then used to check whether data flow through the application is consistent with the specification. The actual program syntax is formalized in a core calculus to model the information flow. The implementation of the analyzer is based on the WALA framework, a widely used open-source library collection for code analysis. The main limitation of their approach is that SCanDroid requires the Java source code or the JVML bytecode of the application. It has never been tested on real Android applications.

AndroidLeaks [34] is a static analysis framework designed to detect privacy leaks in Android applications. Similar to SCanDroid, they use the WALA libraries to perform data-flow analysis. However, they apply their analysis on real applications. Therefore, they use the dex2jar [55] tool to convert the dex bytecode into the Java class format. Using the JVM bytecode, they create an application callgraph to perform a reachability analysis. The tool checks whether the CFG contains paths from sensitive data sources to network sinks. If such paths are found, a data-flow analysis is applied to verify that private data reaches a method that subsequently sends the data via network.

On a large-scale evaluation on 18,000 applications, they found 9,631 potential privacy leaks in about 3,000 different Android applications. Private data leaked via network included phone information, location and WiFi data, as well as audio recorded with the microphone.

In another work Enck *et al.* present ded [27, 46], a Dalvik decompiler. Their system decompiles dex application bytecode back into JVM bytecode. This is a challenging task, since dex is a compact, size-optimized bytecode format (see *Section 2.2.1*). In order to achieve a comprehensive decompilation, many information lost during translation into the dex bytecode format has to be restored. After the reconstruction of Java .class files, they used existing tools for decompilation to finally recover the application's Java code.

They chose to decompile the application bytecode rather than operating directly on the dex bytecode for several reasons. With Java source code they were able to use existing tools to perform static analysis. In their work, a commercial analysis suite was used to search for dangerous functionality and vulnerabilities in Android applications. Another reason is that they required access to the application's Java code to manually verify their results from the automated code analysis.

PiOS [22] is a static analysis system to detect privacy leaks in Apple's iOS operating system. PiOS performs a data-flow analysis on Mach-0 binaries that are compiled from Objective-C code. By using a commercial disassembler, they generate a control flow graph of the application. Then, a reachability analysis is performed to check whether there exist paths from sensitive data sources to sinks that transmit data via the network.

In order to enhance the precision of the results, they applied an additional flow-analysis. Forward propagation is used for every path found in the first step to verify that sensitive information accessed at a source node propagates to method calls in a sink node.

Among all related work, PiOS comes closest to the approach I devised. They created a CFG to perform data flow analysis on the disassembled application binary and used forward propagation to improve their results. However, there are major differences between the approaches. They use the flow analysis only to check if there are paths through the program that connect sources of sensitive data with network data sinks. They cannot provide information about the actual values that are transmitted. Furthermore, my approach is not limited to a certain type of data source. Starting from predefined data sinks, Bati finds any piece of information that reaches this sink, both sensitive and non-sensitive data fragments. This gives a more comprehensive and detailed insight into the application's behaviour.

# Chapter 9

# Conclusion & Future Work

This chapter finally concludes the thesis. It starts by describing the limitations of the work that has been developed and implemented during this master's thesis. This includes both limitations inherited from the chosen approach as well as limitations of the current analysis tool. The chapter continues by summarizing the accomplishments of this thesis. It describes the contribution to the current state of Android application analysis. The concrete implementation of these concepts, the analysis framework Bati, is highlighted in particular. Finally, the last section gives some proposals for the continuation of this work.

## 9.1 Limitations

Although the thesis objectives have been accomplished, there are some limitations and shortcomings. Static analysis approaches always have some limitations as compared to dynamic approaches and vice versa. With respect to the implementation, the respective limitations primarily derive from the limited time frame available during the thesis. In the following, the concrete approach and implementation specific limitations are explained.

### 9.1.1 Approach Limitations

Bati provides offline certification for Android applications. It's approach is based on backwards symbolic execution. In contrast to dynamic runtime analysis, this static approach inherently suffers from the fact that only available code can be completely analyzed. Closed-source API calls can only be resolved symbolically. The full class and method name is output along with the resolved arguments used for the invocation. This set of information represents the most detailed view possible in this context. However, this information is almost never as accurate as the real values that can be observed during runtime analysis. By adding more expert knowledge, mitigation of this drawback is possible.

Loops constitute another common problem in static analysis approaches. *Section 6.3.1* explained the different kind of loops that can occur during analysis. By applying multiple techniques for loop detection, these cases can be handled efficiently. However, it is pretty difficult to extract semantic information from loops, in particular information about the loop variable and its modification. In contrast to the different ways a loop can be created in a Java program, dex bytecode only knows a single kind of loop consisting of a conditional with a jump instruction. *Listing 6.1* showed that even the resolution of a small, compact for-loop is difficult or even infeasible in most cases.

The current approach is limited to the analysis of dex application bytecode. This usually includes any Java source that has been converted to dex bytecode by using the dx compiler. As described in *Section 2.2*, the Dalvik virtual machine is not limited to executing dex bytecode. Developers can use the NDK to generate native code for their applications. This is usually done for outsourcing performance-intensive tasks. Native code is thereby compiled from C++ code and thus differs substantially from dex bytecode. In a large-scale evaluation of Android applications, Chen *et al.* [34] found out that about 6% of Android applications contained at least one native code file. This is an important fact, as making network connections is not limited to Java code. Since the Internet permission is enforced through the underlying Linux kernel, any network communication could be moved to native code as well. In this case no sink would be found by the parsing module. However, if an application requires the Internet permission and no sink is detected, it's a strong indicator for malicious intent.

The static analysis framework targets single Android applications. Bati processes an application and reassembles data that is subsequently transmitted via the network. Based on the data sources found during the analysis, data fragments reaching a network sink can be classified as sensitive or non-private data. This works as long as no further applications/components are involved. In privilege escalation attacks, the interaction of multiple applications might lead to a leakage of sensitive information. For example, an application without Internet permission might retrieve sensitive user data and store it in a file or database. Another application then accesses this data and transmits it to some remote servers. If the application with Internet permission is analyzed, the results will indicate that certain data portions come from some external source. But the results will give no information about the confidentiality of the data. In order to provide this kind of information, the analysis would have to be extended to also process information stored in the manifest file. Parsing this file reveals the application interfaces as well as information about the Intents that are received by the application. This information is then used to detect various kinds of privilege escalation.

### 9.1.2 Implementation Limitations

The current implementation of multi-path encoding in form of target graphs (see also *Section 5.5.4*) suffers from overestimation in certain situations. This is due to the way the graphs are constructed during analysis. Multiple registers in a bytecode instruction are resolved sequentially in a depth-first search. For each register a distinct graph is created, whereas the collected data is encoded in a token list per node. In the current implementation, these separate graphs are merged to a single graph. This allows an easy graph construction on a per sink level. However, it results in overestimations if multiple target graphs encode the same CFG paths. In these cases the decoder produces the cross product of subresults.

*Listing 9.1* shows an illustrative example. It contains a method invocation for the URL class constructor that specifies registers for the *file* variable and the *fileSize* variable. The actual content of both variables depends on the conditional branch that is executed. The only valid combinations that can reach the URL sink are ("A.txt", 80) and ("B.txt", 50).

```
private URL target() {
    [...]

    if ([condition]) {
        file = "A.txt";
        filesize = "80";
    } else {
        file = "B.txt";
        filesize = "50";
    }

    return new URL("http://www.foo.com?"
            + "upload=" + file + "&size=" + fileSize);
}
```

Listing 9.1: Overestimation

The current implementation creates the target graph for the URL sink as depicted in *Figure 9.1a*. The light gray nodes represent the part of the graph belonging to the *file* variable. Once the corresponding register is completely processed, a join node merges the resulting paths, such that the resolving of the *fileSize* variable can continue from that node. Finally, the decoder module will find four distinct paths through the graph, which result in four tuples reaching the URL sink; ("A.txt", 50), ("A.txt", 80), ("B.txt", 50), and ("B.txt", 80).

The multi-path encoding could be improved, if each subsequent register resolution continues at the previous start node (as shown in *Figure 9.1b*). Then, also the approach of storing one token list per node would have to be adapted. Furthermore, it has to be noted that the resolution of multiple registers at a certain instruction can also occur arbitrary times during resolving. This complicates the problem and makes it even more challenging. A more sophisticated way to manage the collected token properly does not only enhance the precision of the results, it even improves the evaluation performance and reduces the memory footprint.

Another limitation is the lack of a sink verification. This means, that it is currently not checked whether a certain network sink is really used to create an outgoing connection. Theoretically, a malicious app could obfuscate its real behaviour to complicate static analysis. Multiple sinks could be defined whereas only one of them is actually used to transmit data. Bati would then find and output all specified sink definitions, including the malicious one. However, an

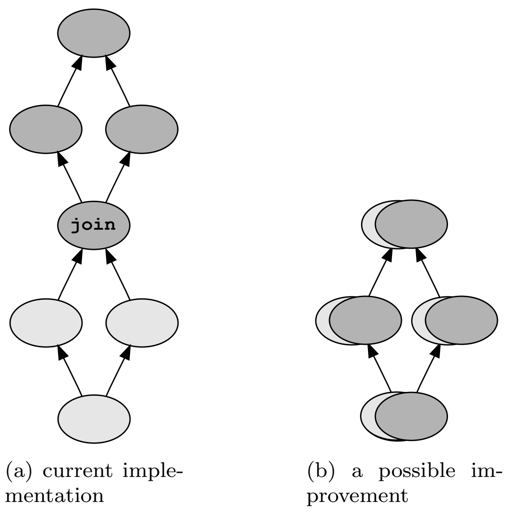


Figure 9.1: Multi-path encoding via target graph

user might think that all URLs are indeed used in this application. Although this is a rare case, such an additional verification step would provide clarity and would validate the set of recovered destinations. Furthermore, connection parameters like request method or referer could be retrieved to provide an even more comprehensive analysis result.

In the Java programming language, developers can use the concept of reflection. With reflection, a program execution can be modified dynamically at runtime. Reflection-oriented components can monitor the execution of a code block and are able to modify themselves according to some conditions related to that block. In general, this is achieved by dynamically assigning application code at runtime. Static analysis of Java reflection is a challenging problem [11, 57] and an area of open research. The current approach implemented in Bati is not capable of handling Java reflection. However, I claim that Bati provides a good basis for a comprehensive and sophisticated resolution of reflective invocations. Class and method information that is assigned dynamically during execution can be resolved with the current approach as long as it does not originate from sources outside the application, like for example from user input.

## 9.2 Conclusion

This thesis presents a novel approach for offline certification of Android applications. In contrast to related work based on static analysis, Bati provides a comprehensive view on the Internet usage of applications. It is capable of outputting the destinations used for network communication as well as the concrete data that is transmitted. Thereby, it is not limited to sensitive data. A comprehensive verification of Internet access is accomplished by making the network communication of Android applications transparent to the user.

The evaluation chapter showed the effectiveness of the extended backwards symbolic execution approach. The output clearly helps users to understand how an application accesses the Internet. Any covert communication that is hidden from the user is revealed. Using the output of Bati, it becomes trivial to detect privacy leaks that are related to Internet usage. Users strongly benefit from this application analysis as almost three out of four apps in the Android market require the Internet permission [61].

In contrast to Apple's App Store, the current Android market has no formal review process [41]. The static analysis framework represents a suitable methodology for a (semi-)automatic application vetting. Although such tools will never be 100% accurate, it would facilitate the application review. Bati automatically analyzes applications and reports the ones that clearly show privacy leaks or some malicious intent. This reduces the number of applications that require manual verification to a manageable size.

The highly modular architecture of Bati allows an easy modification, extension, and/or reuse of existing components. To enhance further research in this area, new components can be simply plugged into the existing infrastructure. The current resolver could for example be replaced by a new component implementing a forward analysis. Existing data structures like CFGs, the SSA form, or lookup tables can be reused to reduce implementation effort. Static single assignment forms are usually generated to perform advanced optimization and validation tasks. Bati's SSA builder generates a SSA form from dex bytecode to facilitate the static analysis and to improve the performance of the algorithm. This form could also be interesting for many other approaches that perform static analysis on Android applications.

The symbolic execution implemented in Bati describes a generic analysis approach. Thus, it is not limited to the verification of the Internet access. By specifying new sinks, the analysis can be applied to nearly arbitrary bytecode instruction. With only little effort, additional sinks can be defined, e.g. for phone calls and SMS transmission. These sinks do not only target the user's privacy, but could also prevent the user from installing applications that silently communicate with premium numbers. This way, a comprehensive security check for Android applications can be accomplished.

## 9.3 Future Work

The work presented in this thesis leaves a wide range of opportunities for future work. Follow-up research may include the port of Bati as an Android application. When integrating this approach as application installer, any new application retrieved from the market can be analyzed before the user finally confirms the installation. This way, an end-user security system is formed that warns users when potentially malicious applications are about to be installed. In order to get this approach working on an Android system, a few challenges have to be overcome. The actual port of the source code is trivial, however the real problem is that an application cannot access the bytecode of other applications.

> *"A central design point of the Android security architecture is that no application, by default, has permission to perform any operations*

> *that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc."*[1]

Since the application's bytecode is a crucial requirement for the analysis, Android's core system would have to be modified to allow Bati read access for application packages.

The precision of the results for method invocations strongly depends on the presence of the corresponding method bytecode. Custom invocations can usually be fully resolved, whereas the lack of bytecode prevents a comprehensive analysis of API invocations. The source of API methods is usually not available within the application package. However, both Java and Android specific APIs are open-source and can be downloaded and analyzed[2]. All public classes and methods could be translated into dex bytecode. In a subsequent pre-processing step, Bati could resolve all class methods like custom application methods. The encoded results are then stored in a database. Each time, the resolver reaches an API invocation during app analysis, the corresponding encoded result is retrieved from the database instead of storing the simple name of the invocation. During evaluation, API invocations can then be resolved completely by using the already generated results. This would strongly enhance the quality of the results and would eliminate some drawbacks in comparison to dynamic runtime analysis approaches.

Another area for optimization targets the improvement of the analysis performance and the reduction of the memory footprint. Devising an improved approach for the multi-path encoding via target graphs would provide great benefit. This would not only effect the performance and memory usage, but also the precision of the final output by eliminating overestimation. Further improvement could be accomplished by introducing multi-threading. Many applications include more than one sink that has to be analyzed. The current implementation is single-threaded, which means that all sinks are processed consecutively. Since the main data structure used for the analysis, the SSA form, is accessed read-only, multi-threading could be introduced without having to implement locking. By analyzing multiple sinks simultaneously, Bati can benefit from modern multi-core CPUs to reduce the processing time dramatically.

The static analysis is currently limited to a single application. Thus, any malicious intent that is concealed by the interaction of multiple applications cannot be revealed. Extending the analysis to processing the information of the application's manifest file could help to cover at least some of these cases.

Finally, another proposal for future works targets the extension and customization of the analysis tool. The generic resolution approach allows an easy extension of verification sinks. By specifying additional sinks, more security and privacy-relevant properties of applications can be checked. Another interesting

[1] `http://developer.android.com/guide/topics/security/security.html`

[2] The Android specific classes can be obtained from the Android Open Source Project [51], the Java API can be downloaded from the OpenJDK project [53].

extension would be to allow users to specify their own sinks. In order to provide maximal customization, users can then also determine the set of sinks to be analyzed.

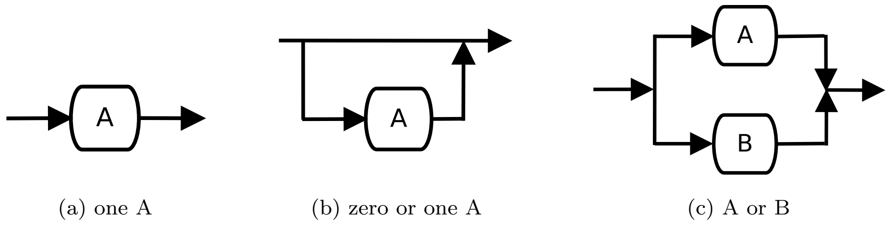
# Appendix A

# Syntax Diagram Notation

In this thesis certain concepts like the control-flow graph in *Section 5.2* and the data encodings in *Section 5.5* are described by context-free grammars which are represented by syntax diagrams. This chapter gives a brief introduction to syntax diagrams and explains all notations that are necessary for understanding.

Each grammar is represented by a set of syntax diagrams. Thereby, each diagram defines a *non-terminal* and has an entry point and an end point. Diagrams describe possible paths between these points by going through other non-terminals and *terminals*. In the context of this thesis, a terminal is usually a constant value or a string. In the diagrams, terminals are represented by round boxes while non-terminals are represented by square boxes. A word belongs to the language defined by the grammar, if it describes a valid path through the root diagram.

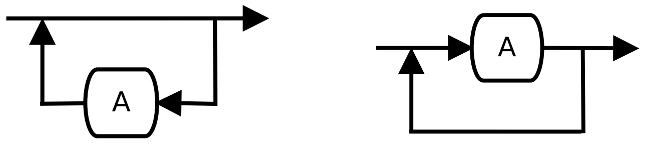
*Figure A.1* shows simple syntax diagrams with terminal symbols *A* and *B*. *Figure A.1a* shows a diagram that describes exactly one terminal A. In the second diagram either zero or one occurrence of A is valid. *Figure A.1c* shows a selection in which either terminal A or B has to be chosen.



(a) one A (b) zero or one A (c) A or B

Figure A.1: Linear syntax diagrams

*Figure A.2* depicts syntax diagrams with a loop. Loops can be used to repeat certain subpaths. In case of *Figure A.2a*, either zero or an arbitrary number of A's is valid. The loop shown in *Figure A.2b* describes at least one occurrence of A.



(a) zero or more A

(b) at least one A

Figure A.2: Loop syntax diagrams

# Bibliography

[1] ABIresearch. Android overtakes apple with 44% worldwide share of mobile app downloads. http://www.abiresearch.com/press/3799-Android+Overtakes+Apple+with+44%25+Worldwide+Share+of+Mobile+App+Downloads, October 2011.

[2] Open Handset Alliance. http://www.openhandsetalliance.com/index.html, November 2007.

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[4] Michael Backes, Sebastian Gerling, and Philipp von Styp-Rekowsky. A local cross-site scripting attack against android phones. http://www.infsec.cs.uni-saarland.de/projects/android-vuln/android_xss.pdf, June 2011.

[5] David Barrera, H. G üne § Kayacik, Paul C. van Oorschot, and Anil Somayaji. 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, pages 73–84, New York, NY, USA, 2010. ACM.

[6] Leonid Batyuk, Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Ahmet Camtepe, and Sahin Albayrak. Developing and benchmarking native linux applications on android. In *MobileWireless Middleware, Operating Systems, and Applications*, volume 7 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 381–392. Springer Berlin Heidelberg, 2009.

[7] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11. ACM, March 2011.

[8] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50:375–425, May 2003.

[9] Android Developers Blog. Exercising our remote application removal feature. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html, June 2010.

[10] The Official Google Blog. 10 billion android market downloads and counting. http://googleblog.blogspot.com/2011/12/10-billion-android-market-downloads-and.html, December 2011.

[11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[12] Dan Bornstein. Dalvik vm internals, May 2008.

[13] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report tr-2011-04, Technische Universität Darmstadt, April 2011.

[14] Ben Cheng and Bill Buzbee. A jit compiler for android's dalvik vm, May 2010.

[15] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[16] The Nielsen Company. U.s. smartphone market: Who's the most wanted? `http://blog.nielsen.com/nielsenwire/online_mobile/u-s-smartphone-market-whos-the-most-wanted/`, April 2011.

[17] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm, 2001.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[20] Dibyendu Das and U. Ramakrishna. A practical and fast iterative algorithm for $\phi$-function computation using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 27(3):426–440, 2005.

[21] Android developer. `http://developer.android.com`.

[22] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Network and Distributed System Security Symposium*, NDSS 2011, San Diego, CA, February 2011.

[23] David Ehringer. The dalvik virtual machine architecture. `http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf`, March 2010.

[24] Gerry Eisenhaur, Michael N. Gagnon, Tufan Demir, and Neil Daswani. Mobile malware madness and how to cap the mad hatters. In *Black Hat Conference*, Las Vegas, NV, August 2011.

[25] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 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, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[26] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. Technical report nas-tr-0120-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, August 2010.

[27] William Enck, Damien Octeau, Patrick McDaniel, and Chaudhuri Swarat. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, August 2011.

[28] William Enck, Machigar Ongtang, and Patrick Mcdaniel. Mitigating android software misuse before it happens. Technical report nas-tr-0094-2008, Pennsylvania State University, September 2008.

[29] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[30] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. Technical report ucb/eecs-2011-48, EECS Department, University of California, Berkeley, May 2011.

[31] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications. `http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf`, 2009.

[32] Inc. Gartner. Competitive landscape: Mobile devices, worldwide, 3q10. `http://www.gartner.com/it/page.jsp?id=1466313`, November 2010.

[33] Inc. Gartner. Market share: Mobile communication devices by region and country, 3q11. `http://www.gartner.com/resId=1847315`, November 2011.

[34] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Detecting privacy leaks in android applications. Technical report, UC Davis, 2011.

[35] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS 2012, Februrary 2012.

[36] Ben Gruver. Smali - an assembler/disassembler for android's dex format. `http://code.google.com/p/smali/`.

[37] D Harel. A linear algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th ACM symposium on Theory of computing*, STOC '85, pages 185–194, New York, NY, USA, 1985. ACM.

[38] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[39] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[40] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1:121–141, January 1979.

[41] Patrick McDaniel and William Enck. Not so great expectations: Why application markets haven't failed security. *IEEE Security and Privacy*, 8:76–78, September 2010.

[42] Inc Millennial Media. Millennial media's mobile mix. `http://www.millennialmedia.com/wp-content/images/mobilemix/MM-MobileMix-March2011.pdf`, March 2011.

[43] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 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, pages 328–332, New York, NY, USA, 2010. ACM.

[44] Juniper Networks. Malicious mobile threats report 2010/2011. www.juniper.net/us/en/local/pdf/whitepapers/2000415-en.pdf, November 2011.

[45] Jon Oberheide. Remote kill and install on google android. http://jon.oberheide.org/blog/2010/06/25/remote-kill-and-install-on-google-android/, June 2010.

[46] Damien Octeau, William Enck, and Patrick McDaniel. The ded Decompiler. Technical report nas-tr-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, September 2010.

[47] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[48] Gabor Paller. Understanding the dalvik bytecode with the dedexer tool. http://pallergabor.uw.hu/common/understandingdalvikbytecode.pdf, December 2009.

[49] Inc Palm Source. Open binder, version 1. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html, 2005.

[50] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[51] The Android Open Source Project. http://source.android.com/.

[52] The Android Open Source Project. Bytecode for the dalvik vm. http://s.android.com/tech/dalvik/dalvik-bytecode.html, 2007.

[53] The OpenJDK Project. http://openjdk.java.net/.

[54] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138, New York, NY, USA, 1959. ACM.

[55] pxb1988. dex2jar: A tool for converting android's .dex format to java's .class format. https://code.google.com/p/dex2jar/.

[56] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[57] Jason Sawin and Atanas Rountev. Improving static resolution of dynamic class loading in java using dynamically gathered environment information. *Automated Software Engg.*, 16:357–381, June 2009.

[58] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS 2011, pages 17–33. The Internet Society, 2011.

[59] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 153–163, New York, NY, USA, 2005. ACM.

[60] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing $\phi$-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 62–73, New York, NY, USA, 1995. ACM.

[61] SMobile Systems. Threat analysis of the android market. `http://globalthreatcenter.com/wp-content/uploads/2010/06/Android-Market-Threat-Analysis-6-22-10-v1.pdf`, June 2010.

[62] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *Proceedings of the WS2P Workshop on Web 2.0 Security and Privacy*, May 2011.

[63] Android x86 Porting Android to x86. `http://www.android-x86.org/`.

[64] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vince Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, Pittsburgh, PA, USA, June 2011.