Master’s Degree programme
in Computer Science
“Software Dependability and Cyber Security”

Final Thesis

Flow logic based information flow analysis of Android applications

**Supervisor**
Dott. Stefano Calzavara

**Graduand**
Fabio Salvini
Matriculation Number 840824

**Academic Year**
2017 / 2018
Abstract

Android is the world’s most popular mobile OS, with more than 2 billion monthly active devices. Static analysis is an essential tool to protect the sensitive data stored in the devices from malicious applications. In this thesis, we present the first flow logic for the information flow analysis of Dalvik bytecode that is specifically tailored to the peculiar lifecycle of Android applications. A prototype implementation based on a state-of-the-art SMT solver demonstrates the practicality of our approach.
## Contents

1 Introduction ........................................................................................................ 5
   1.1 Contributions .................................................................................................. 7
   1.2 Thesis structure .............................................................................................. 7

2 Background ........................................................................................................... 9
   2.1 Android virtual machine ................................................................................. 9
   2.2 Smali ............................................................................................................... 10
   2.3 Information flow control ................................................................................ 11
      2.3.1 Taint analysis .......................................................................................... 11
      2.3.2 Direct/Indirect flows .............................................................................. 13

3 Concrete semantics ................................................................................................ 15
   3.1 Syntax .............................................................................................................. 15
      3.1.1 Instructions ............................................................................................ 17
   3.2 Semantic domains .......................................................................................... 19
   3.3 Activity semantics .......................................................................................... 20
4 Abstract semantics

4.1 Flow logic specification ........................................... 23
4.2 Flow logic judgements .............................................. 25
4.3 Activity rules ......................................................... 29
4.4 Taint analysis ......................................................... 31

5 Implementation

5.1 SMT solver ............................................................ 34
5.2 Abstract domains ...................................................... 34
5.3 Prototype architecture ................................................. 38
   5.3.1 Parser ............................................................. 38
   5.3.2 Pre-analyzer ...................................................... 40
   5.3.3 Constraints generator ........................................... 41
   5.3.4 Z3 ................................................................. 42

6 Experiments

6.1 Direct leak ............................................................ 43
6.2 Life cycle leak ......................................................... 46
6.3 Implicit flow leak ....................................................... 50

7 Conclusions

53
Chapter 1

Introduction

Smartphones are extremely popular nowadays, being used not only as cellphones but also to keep track of activities, as navigation systems or to access the internet. A key feature of smartphones is the extensibility. All popular mobile OS have stores where the user can find and download apps with ease. For their usage, smartphone devices end up storing private and sensitive information that must be treated carefully. Applications can access this information so mechanisms must be developed to ensure the correctness and well-behaviour of the applications. A common approach implemented by all major OS is access control. Authorizations must be granted by the user before a certain app can access sensitive data (like sms, contacts) or use systems features such as the camera or the internet [11].

The problem of access control is that once the program has been granted access, there are no guarantees on how the data is handled and the user is trusting the application to behave correctly.

A more precise control is given by information flow control analysis [14]. It tracks how the information propagates through the program, to make sure its confidentiality and integrity is preserved. We focus on Android, the world’s most popular mobile OS developed by Google. Both dynamic [9], [5] and static [7], [6], [13], [24], [17] analyses have been studied to enhance the security of applications running in the Android framework. Dynamic tools like TaintDroid [9], a taint
tracking system implemented as an extension of the Android execution environment, can track sensitive information leakages in running applications, but they require changes to the underlying OS and the analysis is limited to the running code, it does not consider all possible computational paths. Static tools instead have complete code coverage (at the cost of possible false positives) and can be used to analyze the applications beforehand, blocking malware applications from being published to the market. HornDroid [7] abstracts the semantics of Android applications through Horn clauses, that are discharged by an SMT solver. FlowDroid [6] leverages the Soot framework [21] to perform a control flow analysis on the callgraph. Both tools achieve a notable level of precision, but the analysis is expensive and each time an application component changes, the whole application must be analyzed again. Wognsen at al. [24] formalize a control flow analysis using a compositional flow logic approach. Being compositional, the application components are analyzed at their definition point instead of at their invocation, limiting the analysis computation to the components that have been changed. The resulting analysis is less precise but faster. The flow logic of [24] models advanced features like exceptions and reflection, but lacks a formalization of Android unique characteristics like the activity life cycle and inter-component communication.

In this thesis we formalize the missing features of [24], developing a flow logic that is specially suited for analyses of Android applications. We also apply a taint information model to our flow logic for sensitive information leaks detection. Due to the fact that the existing prototype, based on the flow logic approach, does not work anymore with recent Android applications and uses outdated tools, we implement a new prototype that generates constraints from the flow logic judgments resolved by a state-of-the-art SMT solver.
1.1 Contributions

- We extend an existing flow logic for Dalvik bytecode to support the Android activity lifecycle and inter-component communication.

- We develop a prototype which implements the most significant core of the analysis to detect leaks of sensitive data in Android applications by generating constraints solved by the state-of-the-art SMT solver Z3.

- We present some experiments of sensitive information leaks and how the leak detection is possible using our prototype.

1.2 Thesis structure

In Chapter 2 we present an overview of the applications bytecode running in the virtual machine. We explain how information control flow works and how it can be implemented. Chapter 3 reviews the $\mu$-Dalvik$_A$ formal model, the most complete concrete semantics available for Dalvik bytecode. Our approach to static analysis of Android applications is described in Chapter 4, where we present the abstract domains and rules that will create the foundations of our tool implementation, presented in Chapter 5. We then show some experiments (Chapter 6) that highlight the capabilities of our tool, before drawing our conclusions (Chapter 7).
Chapter 2

Background

2.1 Android virtual machine

Android applications are first compiled into Dalvik bytecode, and then run in the Android runtime (ART) [2]. ART was introduced in Android 5.0 Lollipop as a replacement of the Dalvik virtual machine, the previous runtime system. The new runtime was developed to introduce ahead-of-time compilation, among other improvements in the memory allocation, garbage collection and debugging. ART and Dalvik VM are however compatible, an application compiled into Dalvik bytecode can run in both of them.

ART/Dalvik are based on a register architecture, while regular Java virtual machines are stack-based. In a stack-based virtual machine, operands are stored in a stack data structure. An operation consists of removing (pop) data from the stack and pushing the result back into the stack (in a LIFO order). In a register-based virtual machine instead, data is directly stored into the CPU registers. The advantage is less overhead (since there are no POP/PUSH instructions) that leads to faster execution. The cost however is a significant increase of the instructions size, that impact both the overall size of the program, but also the time required to fetch the instructions to execute (important on unpredictable branches).
Listing 1 shows how the same fragment of Java code is compiled into Java and Dalvik bytecode. Java bytecode is longer but the instructions are very short, instead each Dalvik bytecode instruction carries more information but is bigger in size.

<table>
<thead>
<tr>
<th>Java</th>
<th>Java bytecode</th>
<th>Dalvik bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int a = 1;</code></td>
<td><code>iconst_1</code></td>
<td><code>const/4 v0, 0x1</code></td>
</tr>
<tr>
<td><code>int b = 2;</code></td>
<td><code>istore_1</code></td>
<td><code>const/4 v1, 0x2</code></td>
</tr>
<tr>
<td><code>int c = a + b;</code></td>
<td><code>iconst_2</code></td>
<td><code>add-int v2, v0, v1</code></td>
</tr>
<tr>
<td></td>
<td><code>istore_2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>iload_1</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>iload_2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>iadd</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>istore_3</code></td>
<td></td>
</tr>
</tbody>
</table>

Listing 1: Java/Dalvik bytecode comparison

For a complete comparison analysis between stack and register-based virtual machine we refer to [22] and [10].

2.2 Smali

Dalvik bytecode is stored in binary format, so in order to be analyzed it must be first decompiled in an easier to read format. The most common syntax used today is Smali [4]. It supports the complete Dalvik bytecode specification and it provides quality of life annotations and registers naming without changing the semantic and the number of instructions. In the following chapters we will use the smali syntax to show Dalvik bytecode. Baksmali [4] is a library, written in Java, that accepts dex and apk files and converts them into smali code.

Listing 2 shows the Smali equivalent of the classic Java hello world, where a constant string is printed to the console.
2.3 Information flow control

Information flow control, in the general term, models how the data flows between the application components. We first explain the taint tracking analysis, designed to prove security properties as confidentiality and integrity. Then we show the difference between direct and indirect flows, that determines how a secret information is propagated [18].

2.3.1 Taint analysis

Taint analysis is a form of information flow control where each variable is decorated with a tag, or label, that marks its level of secrecy (usually secret or public). The label, called “taint”, is then propagated when other variables are derived from the tainted one. Taint analysis can either be implemented statically [16] or at runtime [9].

Providers of secret information, called sources, include API methods that return
the device ID, the phone number, the user contacts. Sinks are operations that leak information to the public, like sending an sms message, making an http request or writing to a file.

All taint analysis tools need to identify the sources and sinks prior to the analysis. The Android library provides a huge number of API methods, different approaches have been taken to detect sources and sinks, using machine learning [20], android permissions [12] or even by hand [13].

Taint abstraction on Java objects can be more or less precise depending on the sensitivity of the analysis on fields. Listing 3 shows an example of taint analysis where field-sensitivity determines the precision of the result. A field-insensitive analysis keeps the taint label on the whole object, marking all the contained information as tainted when a secret is assigned to a field. Lines 4, 6 would be detected as leaks. A field-based analysis instead, differentiates between fields but without considering the different instances. Line 4 would not raise a false positive but line 5 would be detected as leak. The more precise analysis is field-sensitive, where each instance field can be marked as tainted without affecting other fields/instances. Only line 6 would be detected as leak.

```
1 C a = new C();
2 C b = new C();
3 a.v = source();
4 sink(a.z); // no leak
5 sink(b.v); // no leak
6 sink(a.v); // leak
```

Listing 3: Taint analysis example
2.3.2 Direct/Indirect flows

When a secret value is directly assigned to a variable or an object field, we have a direct flow (Listing 4).

```
1 C a = new C();
2 a.v = source();
3 sink(a.v); // leak
4 String s = source();
5 String s1 = s;
6 sink(s1); // leak
```

Listing 4: Direct flow example

There are however leakages that are more difficult to detect. Conditionals can be exploited to take different paths depending on the secret information (Listing 5), so the secret can be reconstructed and leaked to the public without direct assignments. As most of the tools available today, our analysis focuses on direct flows for leaks detection.

```
1 String s = source();
2 String s1 = null;
3 if (s.startsWith("a")) {
4    s1 = "a";
5 }
6 sink(s1); // leak
```

Listing 5: Indirect flow example
Our work makes use of the $\mu$-Dalvik$_A$ formal model [7], an extension of the $\mu$-Dalvik calculus [15] that introduces the activity life cycle. In this chapter we explore the syntax and semantics of $\mu$-Dalvik$_A$.

### 3.1 Syntax

The syntax of $\mu$-Dalvik$_A$ programs is defined in Table 3.1. $\mu$-Dalvik$_A$ defines a program $P$ as a sequence of classes $cls^*$. Each class has a direct superclass $c'$, some implemented interfaces $c^*$ and contains a sequence of fields $fld^*$ and methods $mtd^*$. Fields are defined as the name $f$ and its type $\tau$. In Dalvik the field name also contains the type information (a Java field $a$ of type int in Dalvik is encoded as $a:I$), so fields names of the same class are pairwise distinct. Methods have the name $m$, the argument types $\tau^*$, the type of the returned value $\tau$ and the body $st^*$. The annotation $n$ of the method signature is the number of local registers.
### Table 3.1 $\mu$-Dalvik Syntax

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>::= $cls^*$</td>
</tr>
<tr>
<td>$cls$</td>
<td>::= $cls ; c \leq c' ; \text{imp} ; c^* {fld^* ; mtd^*}$</td>
</tr>
<tr>
<td>$\tau_{prim}$</td>
<td>::= $\text{bool}</td>
</tr>
<tr>
<td>$\tau$</td>
<td>::= $c</td>
</tr>
<tr>
<td>$fld$</td>
<td>::= $f : \tau$</td>
</tr>
<tr>
<td>$mtd$</td>
<td>::= $m : \tau^* \xrightarrow{n} \tau {st^*}$</td>
</tr>
<tr>
<td>$st$</td>
<td>::= $\text{goto} ; pc$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
</tbody>
</table>

$r \in \mathit{Registers}$

$pc \in \mathbb{N}$

$\oplus ::= + | - | \ldots$

$\odot ::= - | - | \ldots$

$\ominus ::= < | > | \ldots$

$\text{prim} ::= \text{true} | \text{false} | \ldots$

$lhs ::= r$

$| \; r[r]$  

$| \; r.f$    

$| \; c.f$    

$rhs ::= lhs$

$| \; \text{prim}$
The Dalvik bytecode has many instructions, there are instructions with the same semantic that differs for the number of bits reserved for the operands or the type of the operands. There are also complex instructions that can be rewritten using simpler instructions in sequence, without changing the semantic of the program. \( \mu \)-Dalvik\(_A\) hides the complexity that is not useful for the static analysis (while it is useful to reduce bytecode size and optimize performance) defining a concise set of statements. An example is the \texttt{move lhs rhs} statement (Table 3.2), whose semantic is to move the value of the right-hand side to the left-hand side. The right-hand side can either be a primitive value or an element of the left-hand side. The left-hand side may be a register, an array cell, an object field or a static field.

### Table 3.2 Subset of Dalvik instructions abstracted by \( \mu \)-Dalvik\(_A\) \texttt{move lhs rhs}

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CONST A, B}</td>
<td>Move the literal value B into the register A.</td>
</tr>
<tr>
<td>\texttt{CONST_STRING A, B}</td>
<td>Move a reference to the string specified by the index B into the register A.</td>
</tr>
<tr>
<td>\texttt{MOVE A, B}</td>
<td>Move the contents of the non-object register B to A.</td>
</tr>
<tr>
<td>\texttt{IPUT A, B}</td>
<td>Load or store the object instance field B into the value register A.</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.1.1 Instructions

The unconditional branch \texttt{goto pc} sets the program counter (index of the currently executed instruction) to the \texttt{pc} value. A conditional branch \texttt{if} \( r_1 \ r_2 \) \texttt{then} \texttt{pc} compares the content of the two registers and, if the check is successful, changes the program counter to \texttt{pc} (example in Listing 6). \texttt{unop} \( r_d \ r_s \) and \texttt{binop} \( r_d \ r_1 \ r_2 \) store the result of the unary and binary operations respectively into the \( r_d \) register. Objects are created using the instruction \texttt{new} \( r_d \ c \), which creates an instance of the \( c \) class and stores the reference in the \( r_d \) register. \texttt{newarray} \( r_d \ r_l \ \tau \) stores the reference of a new array of length \( r_l \) and type \( \tau \) into \( r_d \). The instruction \texttt{checkcast} \( r_s \ \tau \) does nothing if the register \( r_s \) contains a reference to an instance of


<table>
<thead>
<tr>
<th>Java</th>
<th>Smali</th>
<th>(\mu)-Dalvik(_A)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (a == 1) {</code></td>
<td><code>const/4 v1, 0x1</code></td>
<td>move r(_1) 1</td>
</tr>
<tr>
<td><code>b = true;</code></td>
<td><code>if-ne v0, v1, :cond_0</code></td>
<td>(\text{if } r_0 \neq r_1 \text{ then } :\text{cond}_0)</td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>const/4 v2, 0x1</code></td>
<td>move r(_2) 1</td>
</tr>
<tr>
<td></td>
<td><code>:cond_0</code></td>
<td><code>:cond_0</code></td>
</tr>
</tbody>
</table>

Listing 6: If statement in Java, Smali and \(\mu\)-Dalvik\(_A\) instructions

type \(\tau\), otherwise stops the execution (exceptions are not formalized by \(\mu\)-Dalvik\(_A\)).

\textit{instof} \(r_d r_s \tau\) stores true into \(r_d\) if \(r_s\) is a reference to an instance of \(\tau\), otherwise it stores false. Method invocation is modelled by \texttt{invoke} \(r_o m r^*\), that calls the method \(m\) on the receiver object referenced by \(r_o\), using \(r^*\) as parameters (example in Listing 7). Invocation of static methods is modelled by \texttt{sinvoke} \(c m r^*\), similar

<table>
<thead>
<tr>
<th>Java</th>
<th>Smali</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A a = new A();</code></td>
<td><code>new-instance v0, Lcom/example/A;</code></td>
</tr>
<tr>
<td><code>a.sum(1, 2);</code></td>
<td><code>invoke-direct \{v0\}, Lcom/example/A;-&gt;&lt;init&gt;()V</code></td>
</tr>
<tr>
<td></td>
<td><code>const/4 v1, 0x1</code></td>
</tr>
<tr>
<td></td>
<td><code>const/4 v2, 0x2</code></td>
</tr>
<tr>
<td></td>
<td><code>invoke-virtual \{v0, v1, v2\}, Lcom/example/A;-&gt;sum(II)I</code></td>
</tr>
<tr>
<td></td>
<td><code>\mu\)-Dalvik\(_A\)</code></td>
</tr>
<tr>
<td></td>
<td><code>new r_0 Lcom/example/A;</code></td>
</tr>
<tr>
<td></td>
<td><code>invoke r_0 Lcom/example/A;-&gt;&lt;init&gt;()V</code></td>
</tr>
<tr>
<td></td>
<td><code>move r_1 1</code></td>
</tr>
<tr>
<td></td>
<td><code>move r_2 2</code></td>
</tr>
<tr>
<td></td>
<td><code>invoke r_0 Lcom/example/A;-&gt;sum(II)I \{r_1,r_2\}</code></td>
</tr>
</tbody>
</table>

Listing 7: Method invocation in Java, Smali and \(\mu\)-Dalvik\(_A\) instructions

to the \texttt{invoke} instruction but instead of the receiver object reference there is the class name. The \texttt{return} instruction operates on a special register \texttt{ret}, where the result of the method invocation must have been previously stored. The last
instructions are dedicated to the unique Android lifecycle. newintent \( r_i \) \( c \) is similar to the \texttt{new} instruction but the newly created instance is of type \texttt{Intent}, and \( c \) is the activity class that will be started on the \texttt{start-activity} \( r_i \) instruction (\( r_i \) is the register that contains the pointer to the intent). Intents have a dictionary-like data structure to pass information between activities. put-extra \( r_i \) \( r_k \) \( r_v \) adds an entry \( k \mapsto v \) to the intent referenced by \( r_i \), where \( k \) and \( v \) are stored respectively in \( r_k \) and \( r_v \). The value is retrieved with get-extra \( r_i \) \( r_k \) \( \tau \), with the requirement that it must have type \( \tau \).

### 3.2 Semantic domains

\( \mu \)-Dalvik\(_A \) operational semantics is based on the semantic domains of Table 3.3.

Android activities are independent screens with a user interface. Their state is represented by \( \Sigma = \alpha \cdot \pi \cdot H \cdot S \), called a local configuration. The callstack \( \alpha \) is a list of local states \( L \), tuples with the program point \( pp \), the list of the method statements \( st^* \) and a function \( R \) that maps each local register to its value. Each time a method is called, a new local state is created and added to the head of the list. When the method returns, the local state is removed. \( \pi \) is a list of intents that represents the pending activity stacks, activities that have been started by the activity in the local configuration. Intents specify which activity to start (marked as \texttt{@c}) and contain the parameters in a dictionary structure. The heap \( H \) maps each location \( \ell \), an annotated pointer, to the corresponding memory block, that can be an object, an array or an intent. Objects are represented similarly to intents, with the difference that the fields are typed. The static heap \( S \) maps each class static field \( c.f \) to its value.
### Table 3.3 $\mu$-Dalvik$_A$ Semantic Domains

<table>
<thead>
<tr>
<th>Domain</th>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointers</td>
<td>$p$</td>
<td>$\in \text{Pointers}$</td>
</tr>
<tr>
<td>Program points</td>
<td>$pp$</td>
<td>$::= c, m, pc$</td>
</tr>
<tr>
<td>Annotations</td>
<td>$\lambda$</td>
<td>$::= pp</td>
</tr>
<tr>
<td>Locations</td>
<td>$\ell$</td>
<td>$::= p_\lambda$</td>
</tr>
<tr>
<td>Values</td>
<td>$u, v$</td>
<td>$::= \text{prim}</td>
</tr>
<tr>
<td>Registers</td>
<td>$R$</td>
<td>$::= (r \mapsto v)^*$</td>
</tr>
<tr>
<td>Local states</td>
<td>$L$</td>
<td>$::= \langle pp \cdot st^* \cdot R \rangle$</td>
</tr>
<tr>
<td>Call stacks</td>
<td>$\alpha$</td>
<td>$::= \varepsilon</td>
</tr>
<tr>
<td>Pending activity stacks</td>
<td>$\pi$</td>
<td>$::= \varepsilon</td>
</tr>
<tr>
<td>Objects</td>
<td>$o$</td>
<td>$::= {c; (f_\tau \mapsto v)^*}$</td>
</tr>
<tr>
<td>Arrays</td>
<td>$a$</td>
<td>$::= \tau[v^*]$</td>
</tr>
<tr>
<td>Intents</td>
<td>$i$</td>
<td>$::= {@c; (k \mapsto v)^*}$</td>
</tr>
<tr>
<td>Memory blocks</td>
<td>$b$</td>
<td>$::= o</td>
</tr>
<tr>
<td>Heaps</td>
<td>$H$</td>
<td>$::= (\ell \mapsto b)^*$</td>
</tr>
<tr>
<td>Static heaps</td>
<td>$S$</td>
<td>$::= (c.f \mapsto v)^*$</td>
</tr>
<tr>
<td>Local configurations</td>
<td>$\Sigma$</td>
<td>$::= \alpha \cdot \pi \cdot H \cdot S$</td>
</tr>
</tbody>
</table>

### 3.3 Activity semantics

A distinctive characteristic of Android, that makes the analysis different from the usual static analysis of Java applications, is its unique lifecycle [1].

Android applications are structured in components, like activities, services and content providers. The behaviour of the application is affected by the callbacks executed by the Android framework (Figure 3.1), caused by system events (low memory, arrival of a phone call), user inputs or inter-component communication (intents). System events are non-deterministic, so they introduce a new challenge in the static analysis of the application behaviour. [7] provides an extension of the $\mu$-Dalvik$_A$, reported in Table 3.4.
Figure 3.1: The Android activity lifecycle
Table 3.4 Extensions to the Syntax of $\mu$-Dalvik$_A$

<table>
<thead>
<tr>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity states $s \in ActStates$</td>
<td>$s$</td>
</tr>
<tr>
<td>Frames $\varphi ::= \langle \ell, s, \pi, \alpha \rangle</td>
<td>\langle \ell, s, \pi, \alpha \rangle$</td>
</tr>
<tr>
<td>Activity stacks $\Omega ::= \varphi</td>
<td>\varphi :: \Omega$</td>
</tr>
<tr>
<td>Configurations $\Psi ::= \Omega \cdot H \cdot S$</td>
<td></td>
</tr>
</tbody>
</table>

**Convention:** each activity stack $\Omega$ contains at most one active (underlined) frame.

The call stack and pending activity stack of the previously described local configuration are decorated with a location $\ell$ pointing to an activity and an activity state $s$. This information forms a frame $\varphi$. An activity stack $\phi$ models the activities currently running in the application. The configuration $\Psi$ pairs the activity stack with the (shared) heap and static heap. Activities are identified as instances of classes that extend the Android Activity class. To track their state and the inter-component communication, $\mu$-Dalvik$_A$ adds the following fields to all the activity classes:

- **finished**: when the activity is first started this field is set to false. When it finishes or it is destroyed by the system, it is set to true.

- **intent**: reference to the intent that has started the activity.

- **parent**: name of the class that has started the activity.

- **result**: reference to the intent that stores the result of the activity computation.
Chapter 4

Abstract semantics

In this chapter we present the information flow analysis that we have specifically designed for the analysis of Android applications. The analysis is based on the concrete semantics of Chapter 3 and it is also greatly influenced by the control flow analysis of [24], that we expand with the formalization of the activities life cycle and inter-component communication. We follow the flow logic approach, a formalism for static analysis that is based on logical systems.

4.1 Flow logic specification

We define a compositional verbose flow logic specification [19]. The analysis is based on on the abstract domains in Table 4.1.

Locations \( \ell = p_\lambda \) keep their concrete interpretation. We do not place restrictions on the abstract domain that approximates the primitive values. Their abstraction is indicated as \( \mathcal{prim} \) and we require the existence of a partial order \( \preceq_{prim} \) such that \((\mathcal{prim}, \preceq_{prim})\) is a lattice. Abstract values are sets of elements that can be either primitive values or locations. We write \((\hat{v}, \preceq)\) as the lattice on the abstract values, where the semantic of \( \preceq \) depends by the type of the value. The statement \( \hat{v} \preceq \hat{u} \) holds if \( \hat{u} \) is not less precise than \( \hat{v} \). Abstract operators \( \hat{\land}, \hat{\lor}, \hat{\circ} \) on abstract values
Table 4.1 Abstract Domains

<table>
<thead>
<tr>
<th>Locations</th>
<th>( \ell ) ::= ( p_\lambda )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs. values</td>
<td>( \hat{v}, \hat{u} ::= \emptyset \mid { \text{prim} } \mid { \ell } \mid \hat{v} \cup \hat{v} )</td>
</tr>
<tr>
<td>Abs. mem. blocks</td>
<td>( \hat{b} ::= \hat{\ell} \mid \hat{i} )</td>
</tr>
<tr>
<td>Abs. objects</td>
<td>( \hat{o} ::= { c; (f_\tau \mapsto \hat{v})^\ast } )</td>
</tr>
<tr>
<td>Abs. intents</td>
<td>( \hat{i} ::= { \cup @c; \hat{v} } )</td>
</tr>
<tr>
<td>Abs. registers</td>
<td>( R_{pp} ::= N \to \hat{v} )</td>
</tr>
<tr>
<td>Abs. heap</td>
<td>( H ::= \ell \to \hat{b} )</td>
</tr>
<tr>
<td>Abs. static heap</td>
<td>( S ::= c \times f \to \hat{v} )</td>
</tr>
<tr>
<td>Abs. pending activities</td>
<td>( I ::= c \to { \hat{i} } )</td>
</tr>
<tr>
<td>Abs. method evaluation</td>
<td>( M ::= c \times m \to \hat{v} )</td>
</tr>
</tbody>
</table>

are defined over the elements of the appropriate abstract domain. Objects keep their concrete interpretation structure, but the fields values are abstract values. The analysis on objects is field-sensitive but flow-insensitive [23], we track the values of the distinct fields but the values are the same at all program points. An object instance is referenced by the program point where it is created, except for activity instances that are referenced by the class name since their creation is managed by the Android framework (so their abstraction is object-insensitive). We write \( \hat{o}.f \) to refer to the field \( f \) of the object \( \hat{o} \). The field type \( \tau \) can be omitted when not needed, since the field name is unique in the object (see Chapter 3).

Intents are field-insensitive. While objects fields are statically known, the intents values are stored in a dictionary-like structure that is hard to track during the analysis. Registers are flow-sensitive, we track their value at every program point. \( R_{pp} \) maps each register index to its abstract value, at the program point \( pp \). We define the operator \( \subseteq \) as follows:

\[
R_{c,m,pc} \subseteq R_{c,m,pc'} \iff \forall n \in \text{dom}(R_{c,m,pc}) : R_{c,m,pc}(n) \preceq R_{c,m,pc'}(n)
\]

\[
R_{c,m,pc} \subseteq_{(i)} R_{c,m,pc'} \iff \forall n \in \text{dom}(R_{c,m,pc}), n \neq i : R_{c,m,pc}(n) \preceq R_{c,m,pc'}(n)
\]

A reference \( l \) is mapped into an abstract memory block by the abstract heap.
Chapter 4. Abstract semantics

H. An abstract memory block \( \hat{b} \) can either be an abstract object or an abstract intent. The static heap \( S \) maps each class field into its abstract value. The compositionality of our flow logic specification requires the use of a cache to hold the results of the methods, analyzed at their definition point. \( M \) is a function that returns the value \( \hat{v} \) that abstracts the computation of the method \( m \) of class \( c \). Activity classes can send intents to start new activities. The intents \( \{\hat{i}\} \) sent by class \( c \) are abstracted by \( \text{I}(c) \). Our analysis does not include arrays and exceptions, since they have already been formalized by [24] and a more precise analysis is out of scope.

For better readability of the flow judgements we introduce the function \( \mathcal{E}_{pp}^{R,H,S}(rhs) \) that evaluates the right hand side \( rhs \) at the the program point \( pp \). If \( rhs \) is an abstract value, it returns the value itself:

\[
\mathcal{E}_{pp}^{R,H,S}(\hat{v}) = \hat{v}
\]

If \( rhs \) is a register \( r_i \), it returns the value of the \( i \) register at the program point \( pp \):

\[
\mathcal{E}_{pp}^{R,H,S}(r_i) = R_{pp}(i)
\]

If \( rhs \) is a static field, it returns its value in the static heap.

\[
\mathcal{E}_{pp}^{R,H,S}(c.f) = S(c, f)
\]

If \( rhs \) is an instance field \( r_i.f \), it returns the join of the fields \( f \) values of the instances referenced by \( r_i \).

\[
\mathcal{E}_{pp}^{R,H,S}(r_i.f) = \bigsqcup_{\ell \in R_{pp}(i)} H(\ell).f
\]

4.2 Flow logic judgements

We now present the judgements for the most significant instructions. The first simple instruction is \texttt{goto pc'}. All registers at the current program point \((c, m, pc)\)
are abstracted by the registers at the program point \((c, m, pc')\).

\[
(R, H, S, M, I) \models_{c, m, pc} \text{goto } pc' \\
\text{iff } R_{c, m, pc} \sqsubseteq R_{c, m, pc'}
\]

For the conditional branch \(\text{if } r_1 r_2 \text{ then } pc'\) we keep the over-approximation that the abstraction consider as if both branches are taken. The registers at both program points \((c, m, pc + 1)\) and \((c, m, pc')\) abstract the registers at the current program point. To increase precision, we would need to introduce the concept of code reachability, that would make harder to express the flow logic judgements since the bytecode is not structured into code blocks that easily identify the instructions inside a conditional.

\[
(R, H, S, M, I) \models_{c, m, pc} \text{if } r_1 r_2 \text{ then } pc' \\
\text{iff } R_{c, m, pc} \sqsubseteq R_{c, m, pc'} \\
R_{c, m, pc} \sqsubseteq R_{c, m, pc + 1}
\]

The move \(lhs rhs\) instruction, with a register as the left hand side, sets the register \(i\) as an approximation of the evaluation of the \(rhs\). The abstraction of all the other registers is carried on as usual.

\[
(R, H, S, M, I) \models_{c, m, pc} \text{move } r_i \text{ rhs} \\
\text{iff } \hat\oplus R_{c, m, pc}(s) \preceq R_{c, m, pc + 1}(i) \\
R_{c, m, pc} \sqsubseteq\{i\} R_{c, m, pc + 1}
\]

When the left hand side is an instance field \(r_i.f\), the object abstraction is retrieved from the heap and a new constraint on the field value is applied. The registers are not changed.

\[
(R, H, S, M, I) \models_{c, m, pc} \text{move } r_i.f \text{ rhs} \\
\text{iff } \forall \ell \in R_{c, m, pc}(i) : \\
\mathcal{E}_{c, m, pc}^{R, H, S}(rhs) \preceq H(\ell).f \\
R_{c, m, pc} \sqsubseteq R_{c, m, pc + 1}
\]

Unary and binary operations change the value of the destination register \(rd\) with the result of the operation on the argument registers.

\[
(R, H, S, M, I) \models_{c, m, pc} \text{unop } r_d \text{ rs} \\
\text{iff } \hat\circ R_{c, m, pc}(s) \preceq R_{c, m, pc + 1}(d) \\
R_{c, m, pc} \sqsubseteq\{d\} R_{c, m, pc + 1}
\]
(R, H, S, M, l) |=_{c, m, pc} \textbf{binop}_{\oplus} r_d r_1 r_2
\iff R_{c, m, pc}(1) \oplus R_{c, m, pc}(2) \leq R_{c, m, pc+1}(d)
R_{c, m, pc} \sqsubseteq \{d\} R_{c, m, pc+1}

The \texttt{return} instruction sets the result of the method evaluation as an approximation of the \textit{ret} register, a special register that for the purpose of the analysis can be seen just as a register with an index that does not overlap with the existing ones (ex. -1).

(R, H, S, M, l) |=_{c, m, pc} \texttt{return}
\iff R_{c, m, pc}(\textit{ret}) \preceq M(c, m)

On the \texttt{new} \( r_d c' \) instruction, an instance of \( c' \) must be referenced by the current program point. The instance fields abstract their default values. It’s important to note that the same \texttt{new} instruction can be called multiple times (inside a loop), so there might already be an object referenced by the current program point. The destination register is updated with the reference.

(R, H, S, M, l) |=_{c, m, pc} \texttt{new} \ r_d \ c'
\iff H(c, m, pc) = \{c'; f_\tau_1 \mapsto \hat{v}_i, \ldots, f_\tau_n \mapsto \hat{v}_n\}
\forall i \in [1, n] : \hat{0}_\tau_i \preceq \hat{v}_i
(c, m, pc) \preceq R_{c, m, pc+1}(d)
R_{c, m, pc} \sqsubseteq \{d\} R_{c, m, pc+1}

The rule of the \texttt{invoke} \( r_0 \ m' \ r'_1, \ldots, r'_n \) instruction is the most complicated one, since it has to deal with dynamic dispatching. The name of the method is statically known, but the actual method that will be called depends by the object referenced by \( r_0 \). First the register \( r_0 \) is dereferenced, than the resulting object’s class and the method signature are used to obtain the method implementation that will be called. The method signature is obtained using the function \texttt{sign}(c', m'), that returns the signature of the method \( m' \) invoked on an instance of \( c' \) (if \( c' \) does not implement \( m' \), it returns the parent implementation signature). \( \tau_1, \ldots, \tau_n \) are the method parameters types, and \( loc \) is the number of method local registers. The method parameters abstract the values \( r'_1, \ldots, r'_n \), the register that represent the
this object abstracts $\ell$ and all the other registers are initialized to their default values. The return register is updated with the value provided by $M$.

$$(R, H, S, M, I) \models_{c, m, pc} \text{invoke } r_0 \ m' \ r'_1, \ldots, r'_n$$

iff $\forall \ell \in R_{c, m, pc}(0) :$

$$H(\ell) = \{[c'; (f \to \_)]^*\} \Rightarrow$$

$$\text{sign}(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{\text{loc}} \tau$$

$$\forall j \in [1, \text{loc}] : 0_j \preceq R_{c', m', 0}(j)$$

$$\ell \preceq R_{c', m', 0}(\text{loc} + 1)$$

$$\forall k \in [1, n] : R_{c, m, pc}(k) \preceq R_{c', m', 0}(\text{loc} + k)$$

$$M(c', m') \preceq R_{c, m, pc + 1}(\text{ret})$$

$$R_{c, m, pc} \sqsubseteq \{\text{ret}\} R_{c, m, pc + 1}$$

The rule of static invocation $\text{sinvoke } c' \ m' \ r'_1, \ldots, r'_n$ is a simplified version of the previous one. The class $c'$ and method $m'$ are known, so the signature can be retrieved easily. The invoked method local registers abstract their default values, there is no this parameter since the method is static and the method parameters abstracts the values $r'_1, \ldots, r'_n$. The return register is updated with the method result.

$$(R, H, S, M, I) \models_{c, m, pc} \text{sinvoke } c' \ m' \ r'_1, \ldots, r'_n$$

iff $\forall \ell \in R_{c, m, pc}(0) :$

$$H(\ell) = \{[c'; (f \to \_)]^*\} \Rightarrow$$

$$\text{sign}(c', m') = \tau_1, \ldots, \tau_n \xrightarrow{\text{loc}} \tau$$

$$\forall j \in [1, \text{loc}] : 0_j \preceq R_{c', m', 0}(j)$$

$$\forall k \in [1, n] : R_{c, m, pc}(k) \preceq R_{c', m', 0}(\text{loc} + k)$$

$$M(c', m') \preceq R_{c, m, pc + 1}(\text{ret})$$

$$R_{c, m, pc} \sqsubseteq \{\text{ret}\} R_{c, m, pc + 1}$$

Instruction $\text{start-activity } r_i$ creates a new activity that will be displayed when the current method finishes. All intents referenced by $r_i$ must be contained in $I(c)$.

$$(R, H, S, M, I) \models_{c, m, pc} \text{start-activity } r_i$$

iff $\forall \ell \in R_{c, m, pc}(i) :$

$$H(\ell) = \hat{i} \Rightarrow \hat{i} \in I(c)$$

$$R_{c, m, pc} \sqsubseteq R_{c, m, pc + 1}$$

Intents are created with the $\text{newintent } r_d \ c'$ instruction. An intent to class $c'$ is referenced by the current program point $(c, m, pc)$. There are no constraints on
Chapter 4. Abstract semantics

the intent value because it is initialized to the empty set. The intent reference is stored in the \( r_d \) register.

\[(R, H, S, M, I) \models_{c, m, pc} \text{newintent } r_d \ c'
\]
\[\text{iff } H(c, m, pc) = \{[@c'; _]\}
\[(c, m, pc) \preceq R_{c, m, pc+1}(d)
\]
\[R_{c, m, pc} \sqsubseteq \{d\} R_{c, m, pc+1}
\]

After an intent has been created, it is possible to add an entry to its dictionary structure with the \text{put-extra} \( r_i \ r_k \ r_j \) instruction. All the values of the intents referenced by \( r_i \) must abstract the value of \( r_j \). We track the intent content as a single abstract value and not a map so the key in the \( r_k \) register is unused.

\[(R, H, S, M, I) \models_{c, m, pc} \text{put-extra } r_i \ r_k \ r_j
\]
\[\text{iff } \forall \ell \in R_{c, m, pc}(i) :
\]
\[H(\ell) = \{[@c'; \hat{v}]\} \Rightarrow R_{c, m, pc}(j) \preceq \hat{v}
\]
\[R_{c, m, pc} \sqsubseteq R_{c, m, pc+1}
\]

In the \text{get-extra} \( r_i \ r_k \ \tau \) instruction, the value of the \( ret \) register must approximate the values of the intents referenced by \( r_i \). Again, the key \( r_k \) is unused and so the type \( \tau \) since we do not model exceptions.

\[(R, H, S, M, I) \models_{c, m, pc} \text{get-extra } r_i \ r_k \ \tau
\]
\[\text{iff } \forall \ell \in R_{c, m, pc}(i) :
\]
\[H(\ell) = \{[@c'; \hat{v}]\} \Rightarrow \hat{v} \preceq R_{c, m, pc}(ret)
\]
\[R_{c, m, pc} \sqsubseteq \{ret\} R_{c, m, pc+1}
\]

### 4.3 Activity rules

As explained in Chapter 3, Android classes that extend the \texttt{Activity} class have callbacks that are called by the Android framework, outside the normal application functions. To model this unpredictable behaviour, we introduce a set of rules that must be satisfied by all analyses.

The status of an activity (running or finished) is abstracted by the class field \texttt{finished}. It is set to \( \top_{\text{bool}} \) since it is difficult to track during static analysis.
Chapter 4. Abstract semantics

Finish rule:

\[\forall c \in \text{dom}(H) : \]
\[H(c) = \{c; f_1 \mapsto \hat{v}_1, \ldots, f_n \mapsto \hat{v}_n, \text{finished} \mapsto \hat{v}', \text{finished} \mapsto \hat{v}'\} \land c \preceq \text{Activity} \Rightarrow \]
\[\top_{\text{bool}} \preceq \hat{v}'\]

When the screen orientation changes, the activity fields may be reset to their default values. Replace rule:

\[\forall c \in \text{dom}(H) : H(c) = \{c; f_{\tau_1} \mapsto \hat{v}_1, \ldots, f_{\tau_n} \mapsto \hat{v}_n\} \land c \preceq \text{Activity} \Rightarrow \]
\[\forall i \in [1, n] : \hat{0}_{\tau_i} \preceq \hat{v}_i\]

Intents are used by activities to start child activities. The parent field of an activity class \(c'\) that receives an intent from class \(c\) must approximate \(c\), and the value \(\hat{u}'\) of the intent referenced by the intent field must approximate the value \(\hat{u}\) of the intent sent by \(c\). Activity rule:

\[\forall c \in \text{dom}(l) : \{\@c'; \hat{u}\} \in l(c) \Rightarrow \]
\[H(c') = \{c'; f_1 \mapsto \hat{v}_1, \ldots, f_n \mapsto \hat{v}_n, \text{parent} \mapsto \hat{v}', \text{intent} \mapsto \ell\}\]
\[\land H(\ell) = \{\@c'; \hat{u}'\} \land c \preceq \hat{v}' \land \hat{u} \preceq \hat{u}'\]

If an activity starts a child activity expecting a result, it receives the result in the respective class field. The following rule states that the result \(\hat{u}\) of an activity class with \(c'\) as parent, must be approximated by the result \(\hat{u}'\) of \(c\). Result rule:

\[\forall c \in \text{dom}(H) : \]
\[H(c) = \{c; f_1 \mapsto \hat{v}_1, \ldots, f_n \mapsto \hat{v}_n, \text{parent} \mapsto c', \text{result} \mapsto \hat{u}\}\]
\[\land c \preceq \text{Activity} \land \{\@c; _\} \in l(c') \Rightarrow \]
\[H(c') = \{c'; f_1 \mapsto \hat{v}_1', \ldots, f_n' \mapsto \hat{v}_n', \text{result} \mapsto \hat{u}'\}\]
\[\land \hat{u} \preceq \hat{u}'\]
The last rule is for the activities entry points. The function \( \text{callbacks}(c) \) returns all the methods of \( c \) that may be invoked by the Android framework at runtime. The method registers abstraction is similar to the one presented for the \texttt{invoke} instruction, but the parameters are set to \( \top \) since they are not known. Callback rule:

\[
\forall c \in \text{dom}(H), m \in \text{callbacks}(c) : \\
\quad c \preceq \text{Activity} \Rightarrow \\
\quad \forall j \in [1, \text{loc}] : \hat{0}_{\tau_j} \preceq R(c, m, 0)(j) \\
\quad \land c \preceq R(c, m, 0)(\text{loc} + 1) \\
\quad \land \forall k \in [1, n] : \top_{\tau_k} \preceq R(c, m, 0)(\text{loc} + 1 + k)
\]

where \( \text{sign}(c, m) = \tau_1, \ldots, \tau_n \rightarrow \tau \)

### 4.4 Taint analysis

We formalize the information flow control using a taint analysis. As [7] we define a label \( h \) that identifies an information as \text{secret} or \text{public}. Our goal is to check confidentiality, so we define the taint information as a lattice where \text{public} is bottom and \text{secret} top. The taint is applied to the primitive values \( \hat{\text{prim}} \), forming a two-valued lattice indicated as \( \hat{\text{prim}}^h \).

\[
\text{Taints } h ::= \text{secret} \mid \text{public}
\]

\[
\text{Abs. values } \hat{\nu}, \hat{\mu} ::= \emptyset \mid \{\hat{\text{prim}}^h\} \mid \{\ell\} \mid \hat{\nu} \sqcup \hat{\nu}
\]

The abstract taint function \( \hat{h} \) returns the taint information of the abstract value.

\[
\hat{h}(\hat{\nu}) = \begin{cases} 
\hat{\nu} & \text{if } \hat{\nu} = \{\hat{\text{prim}}^h\} \\
\hat{h}(\hat{\nu}_i) & \text{if } \hat{\nu} = \{\ell\} \land H(\ell) = \{\langle @c; \hat{\nu}_i \rangle\} \\
\bigsqcup_i \hat{h}(\hat{\nu}_i) & \text{if } \hat{\nu} = \{\ell\} \land H(\ell) = \{\langle c; (f \mapsto \hat{\nu}_i)^\ast \rangle\} \\
\hat{h}(\hat{\nu}_i) \sqcup \hat{h}(\hat{\nu}_j) & \text{if } \hat{\nu} = \hat{\nu}_i \sqcup \hat{\nu}_j \\
\text{public} & \text{if } \hat{\nu} = \emptyset
\end{cases}
\]
If the abstract value is a primitive value, it returns the taint annotation. If it is a reference to an intent, returns the taint information of the intent value. In case of object references it returns the join of all the taint information of the object fields. The join of taints returns \textit{secret} if at least one is \textit{secret}, otherwise returns \textit{public}.

For the taint propagation logic it is only required to change the semantics of the unary and binary operations as follows:

\[
\begin{align*}
v_d &= \odot v_s & \hat{h}(\hat{v}_d) &= \hat{h}(\hat{v}_s) \\
v_d &= v_1 \oplus v_2 & \hat{h}(\hat{v}_d) &= \hat{h}(\hat{v}_1) \sqcup \hat{h}(\hat{v}_2)
\end{align*}
\]

The resulting value of a unary operation has the same taint as the operand. The taint of a binary operation result is the join of the operands taints.

We assume the existence of two sets \textit{Sinks} and \textit{Sources} that contain pairs \((c,m)\) if the method \(m\) of class \(c\) is classified as sink/source. A value returned by a sink is always tainted as \textit{secret}, and if a \textit{secret} is given as parameter to a sink, we have a leak.
We developed a static analysis tool based on our flow logic analysis.

The existing prototype by [24] decompiles apk files into Smali code, then a Python parser generates Prolog constraints that are solved by a XSB Prolog Engine. Instead of expanding the tool we decided to replace it with a new architecture for a couple of reasons. The Android apk structure has changed in recent years, with the introduction of the ART runtime and the Python parser does not work with the newly generated apks. A better solution for parsing is to use the Baksmali API to work directly with Smali instructions. To increase efficiency in the constraint resolution we decided to use an SMT solver instead of a Prolog engine. While the Prolog engine can express constructs that an SMT solver cannot easily encode (recursive goals, dynamically allocated data structures of unknown size, etc.), SMT solvers excel at solving problems of software verification.

Our prototype is written in Scala. We had to choose a programming language for the JVM since the Baksmali APIs are only available for Java. Scala has great interoperability with Java, and provides a powerful type system that makes it easy to express the flow logic judgements. The source code of our prototype is available at: https://git.fabiosalvini.com/fabio/DalvikBytecodeAnalysis.
5.1 SMT solver

A Satisfiability Modulo Theories (SMT) problem is a decision problem for logical first order formulas. It can be seen as a boolean Satisfiability problem (SAT) with predicates instead of binary values. Those predicates operate on theories such as arithmetic, bit-vectors, arrays and uninterpreted functions.

Z3 is a state-of-the art theorem prover from Microsoft Research [8]. It is used in many software verification tools, thanks to its supported theories that allow an easy mapping between software constructs and SMT constraints. We have chosen Z3 since it is one of the most powerful SMT solvers, is free for academic research and provides a set of APIs for many programming languages.

5.2 Abstract domains

The representation of the abstract domains presented in Chapter 4 for their usage with the SMT solver is not straightforward. First order logic problems are (in general) undecidable, so their resolution is based on heuristics and the performances can drastically change depending on how the problem is encoded. While maximizing the performances was not the primary goal of our work, we discarded approaches that would result in a simple/elegant implementation with unacceptable performances, based on preliminary experiments.

One of the major challenges in the implementation was how to encode abstract values. In our flow logic, abstract values are unions of primitive types and references. While it is possible to encode complex datatypes in Z3, we found that it had a great impact on performances. To overcome this problem we decided to distinguish between different types of registers, object fields, etc. For example, instead of having a single register at the index 0 of a given program point, we have two registers, one that stores the primitive values and another that stores the references. This solution allows Z3 to be efficient when solving constraints.
We define the values abstract domain as in Listing 8. An abstract domain is a lattice with an abstract type `Element` that must be a subtype of a Z3 expression. The sort is the Z3 datatype. Top, bottom, meet and join are the expected operations on a lattice, and approx is used to generate an SMT boolean constraint where the second element abstract the first.

```scala
trait AbstractDomain {
  type Element <: Expr
  def sort: Sort
  def top: Element
  def bottom: Element
  def meet(e1: Element, e2: Element): Element
  def join(e1: Element, e2: Element): Element
  def approx(e1: Element, e2: Element): BoolExpr
}
```

**Listing 8: Values abstract domain**

The abstract domain for primitive values (Listing 9) is an abstract domain with the addition of operations to convert a dex type (ex. an integer) to an `Element` and the unary/binary operations on the elements.

```scala
trait PrimitiveDomain extends AbstractDomain {
  def toElement(n: Long): Element
  def toElement(n: Double): Element
  def neg(e: Element): Element
  def sum(e1: Element, e2: Element): Element
  def div(e1: Element, e2: Element): Element
  ...
}
```

**Listing 9: Primitive values abstract domain**
The abstract registers $R_{pp}$ is represented as multiple functions, one for each program point and register type, that map an integer (the index) to a Z3 expression. This is necessary due to the lack of higher-order functions in Z3.

The abstract object $\hat{o}$ is represented as an algebraic datatype defined in Listing 10. The datatype constructor (lines 2-17) has a name (line 4), a string used as recognizer (line 5), and the fields names/sorts. The name field is a string with the name of the dex class. The other fields are associative arrays that represent the object fields, one for each type. Object fields are accessed using the functions in lines 20-23.
```
class Z3Object(ctx: Context, domains: AbstractDomainsCollection) {
    private val constructor: Constructor =
        ctx.mkConstructor(
            "mk-object",
            "is-object",
            Array("name", "prim", "ref", "taint"),
            Array[Sort](
                ctx.mkStringSort(),
                ctx.mkArraySort(ctx.mkStringSort(),
                    domains.primitiveDomain.sort),
                ctx.mkArraySort(ctx.mkStringSort(),
                    domains.referenceDomain.sort),
                ctx.mkArraySort(ctx.mkStringSort(),
                    domains.taintDomain.sort)
            ),
            Array(0, 0, 0, 0)
        )

    private val accessors = constructor.getAccessorDecls
    val nameAccessor: FuncDecl = accessors(0)
    val primAccessor: FuncDecl = accessors(1)
    val refAccessor: FuncDecl = accessors(2)
    val taintAccessor: FuncDecl = accessors(3)
}
```

Listing 10: Z3 representation of an abstract object
5.3 Prototype architecture

The prototype architecture is shown in Figure 5.1. The Android application apk is given as input to the parser, along with the list of entry points and sources/sinks (information needed by the analysis that cannot be extracted from the apk). The output is the Scala representation of a dex application which is given to the constraints generator, along with a pre-analysis result (explained later). Each instruction is mapped to a collection of Z3 constraints, that together form a goal. The external process Z3 solves the goal and the output (a model or unsatisfiable) determines the presence of sensitive information leaks.

5.3.1 Parser

The apk is parsed into a DexApp (Listing 11). From the manifest file we need to extract the activity classes and the UI callbacks (methods called by UI events like the press of a button). There are no tools (of our knowledge) that provides this kind of information given an apk, so we first decompile the apk using Apktool [3] and then parse the resulting manifest and resource files. The classes are parsed using the Baksmali library [4], that operates directly on the given apk. Entry points and sources/sinks are provided via external files. Entry points are callbacks that are called by the Android framework, like the Activity onCreate/onStart/onStop methods. Sources and sinks are provided by a file made publicly available by [20] (extract in Listing 12).
Figure 5.1: Prototype architecture. Rectangles are processes and ellipses data structures
5.3.2 Pre-analyzer

In our flow logic, abstract values are represented as sets and flow logic judgements like the one for the `invoke` instruction are defined using universal quantifiers. Z3 has limited support for universal quantifiers, since there cannot be a decision procedure for formulas involving them.

We introduce a pre-analysis step where we extract the creation point of all objects (looking at the `new-instance` instructions). Then, we change the flow logic judgements that use universal quantifiers with an equivalent rule where the quantifier operates on a known set. For example, the flow logic judgement for the
move \( r_i.f \) \( rhs \) instruction:

\[
(R, H, S, M, I) \models_{c,m,pc} \text{move } r_i.f \text{ rhs}
\]

iff \( \forall \ell \in R_{c,m,pc}(i) : \)

\[
\mathcal{E}^R_{c,m,pc}(rhs) \preceq H(\ell).f
\]

\( R_{c,m,pc} \sqsubseteq R_{c,m,pc}+1 \)

is changed into:

\[
(R, H, S, M, I) \models_{c,m,pc} \text{move } r_i.f \text{ rhs}
\]

iff \( \forall \ell \in \text{aloc}(f) : \)

\[
\ell \preceq R_{c,m,pc}(i) \Rightarrow \mathcal{E}^R_{c,m,pc}(rhs) \preceq H(\ell).f
\]

\( R_{c,m,pc} \sqsubseteq R_{c,m,pc}+1 \)

Where \( \text{aloc}(f) \) returns all the locations that reference objects with a field \( f \). Since these locations are known during the analysis, instead of writing a constraint with a quantifier we generate \( n \) implications (where \( n \) is the size of the \( \text{aloc}(f) \) domain) that add a constraint on the \( H(\ell) \) field \( f \) only if the register \( R_{c,m,pc}(i) \) approximates the reference \( l \). The number of generated constraints is higher but they can be solved quickly by Z3.

### 5.3.3 Constraints generator

After an application has been successfully parsed, the analysis starts from the application entry points to analyze the methods instructions. While generating the constraints, encountered methods that have not been analyzed are added to the analysis. The analysis of each method instructions generates a list of Z3 constraints, decorated with a textual representation of the rule (for better readability). The resulting constraints form a goal, given to Z3 for its resolution.

### Abstract domains

To show the potential of a parametric abstract domain to represent the primitive values we provide two interchangeable abstract domains. The first is the bit-vector
set domain. It is especially suited to represent the dex primitive values since it is
close to the actual implementation. The other domain is the top domain, a domain
that maps each primitive value to top, losing any information about the value but
not imposing any constraint to solve. The top domain is useful when the only goal
of the analysis is to check for leakages of sensitive information, since only the taint
information is needed. To represent references we use the Z3 string array type.
The domain for the taint information is still an abstract domain, where bottom is
public and top is secret (represented as booleans).

API methods

The implementation of classes and methods that are part of the Java standard
library and Android APIs is not known during the analysis of the Android apk. We
set to top the return value of an unknown method. For the analysis to be sound,
we would need to consider the possibility that the unknown method changes the
heap (for example changing a field of an object parameter). This however would
greatly decrease the precision of the analysis, so it’s common practice to consider
API methods as pure. If an unknown method receives a tainted value as parameter
we taint its result value, since it likely been computed using the parameters.

5.3.4 Z3

Z3 is an external process that receives a goal (a collection of constraints that
must be satisfied) and searches for a model where all constraints are satisfied.
If the model is found, it returns Satisfiable (with the model), otherwise it
returns Unsatisfiable. Since we model the leak detection as constraints where
the sink methods must not receive a tainted input, if Z3 finds a model then there
are no leaks in the application (within the reach of our analysis). If the model
cannot be found, then the leak may be a true leak or a false positive due to
over-approximation.
In this chapter we show some examples of applications analyses for leaks detection. The first example shows a direct leak, where the main challenge is the model of unknown API methods. In the second example we introduce the Android activity lifecycle. Finally, we show an example of leak that is not detected by our prototype, due to an indirect flow. The first two examples sources are taken from DroidBench [6], a popular benchmark for taint-analysis tools on Android applications, adapted for better readability. Performance-wise, all example analyses terminate in less than a second.

6.1 Direct leak

The first example is a direct leak (Listing 13). MyActivity is the main activity of the application, on the onCreate entry point the TelephonyManager is used to get the imei of the device (getDeviceId). The imei is then appended to an url that is used to open an http connection, leaking the imei. Listing 14 shows a simplified smali code (no package names and exception handling) of the onCreate method.
public class MyActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my_activity);
        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source
        String query = new String("http://www.google.com/search?q=");
        query = query.concat(imei);
        try {
            URL url = new URL(query);
            url.openConnection(); //sink, leak
        } catch (Exception e) {} 
    }
}

Listing 13: Direct leak
Our prototype correctly detects the leak. We can make it print the constraints descriptions to see how the leak is detected (Listing 15). The invocation of \texttt{LTelephonyManager;}\rightarrow\texttt{getDeviceId()}\texttt{LString;} taints the \textit{ret} register (line 1). It happens because the method is detected as a source of sensitive information, and since the \texttt{String} class code is not known, the taint is applied to the reference. The value of the \textit{ret} register is then moved to the register 0, and so the taint (line 2). It is then carried on between each instruction (lines 3-5) until it is given as parameter to the \texttt{LString;}\rightarrow\texttt{concat(LString;)}\texttt{LString;} method. The method implementation is not known, so we taint the result since it has received a tainted value as
parameter (line 6). The result is moved to the register 1 (line 7) that is then given
as parameter to the URL constructor. Similarly to the previous invocations, the URL
constructor implementation is not known so we taint the reference of the invoked
object (line 9). Then the Ljava/net/URL;->openConnection()LjavaURLConnection;
method is invoked and it is detected as a sink. We require that the invoked object
reference must not be tainted, making it impossible to Z3 to find a model.

Listing 15: Rules that make the direct leak detection possible

6.2 Life cycle leak

In the next example we show a leak which detection requires a model of the
Android activity life cycle (Listing 16). As in the previous example, the device
imei is retrieved from the TelephonyManager on the activity creation, but it is
appended to a class field. On the onStart method (a callback invoked when the
activity becomes visible to the user) the method connect is called, that opens a
connection using the class field URL. Since the imei has been previously appended
to the url, there is a leak. The corresponding (simplified) Smali code is presented
in Listing 17.

```java
public class MyActivity extends Activity {
    private static String URL = "http://www.google.com/search?q=";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my_activity);

        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String imei = telephonyManager.getDeviceId(); //source
        URL = URL.concat(imei);
    }

    @Override
    protected void onStart(){
        super.onStart();
        try{
            connect();
        } catch(Exception e){}
    }

    private void connect() throws IOException{
        URL url = new URL(URL);
        url.openConnection(); //sink, leak
    }
}
```

Listing 16: Life cycle leak
.class public LMyActivity;
.field private static URL:Ljava/lang/String;
.method protected onCreate(LBundle;)V
    invoke-virtual {p0, v2}, LMyActivity;->setContentView(I)V
    const-string/jumbo v2, "phone"
    invoke-virtual {p0, v2},
        LMyActivity;->getSystemService(LString;)LObject;
    move-result-object v1
    check-cast v1, Landroid/telephony/TelephonyManager;
    invoke-virtual {v1}, LTelephonyManager;->getDeviceId()LString;
    move-result-object v0
    sget-object v2, LMyActivity;->URL:LString;
    invoke-virtual {v2, v0}, LString;->concat(LString;)LString;
    move-result-object v2
    sput-object v2, LMyActivity;->URL:LString;
    return-void
.end method
.method protected onStart()V
    invoke-super {p0}, LActivity;->onStart()V
    invoke-direct {p0}, LMyActivity;->connect()V
    return-void
.end method
.method private connect()V
    new-instance v0, LURL;
    sget-object v1, LMyActivity;->URL:LString;
    invoke-direct {v0, v1}, LURL;-><init>(Ljava/lang/String;)V
    invoke-virtual {v0}, LURL;->openConnection()Ljava/net/URLConnection;
    return-void
.end method

Listing 17: Life cycle leak (simplified Smali)
Again, the leak is detected. The rules that make it possible are in Listing 18. The *ret* register is tainted on the invocation of

`LTelephonyManager;->getDeviceId(LString;` (line 1). The value is then moved to the register 0 (line 2). It is propagated between instructions (lines 3-4), then makes the result of `LString;->concat(LString;` `LString;` tainted (line 5). The result is moved into register 2 (line 6). The url is moved into the static class field, making it tainted (line 7). The `onStart` method is analyzed because it’s an activity entry point and it calls the `connect` method. The tainted class field value is moved into the register 1 and so the taint (line 8). The taint is then propagated to the register 0 since the method invocation receives a tainted parameter (line 9) and at the end the invocation of the sink method

`Ljava/net/URL;->openConnection()` `LURLConnection;` generates a leak check (line 10).

```
1 Taint R(MyActivity,onCreate,8)(ret)
2 R(MyActivity,onCreate,8)(ret) <= R(MyActivity,onCreate,9)(0)
3 R(MyActivity,onCreate,8) <={0} R(MyActivity,onCreate,9)
4 R(MyActivity,onCreate,9) <={2} R(MyActivity,onCreate,10)
5 Taint R(MyActivity,onCreate,11)(ret) if
   -> R(MyActivity,onCreate,10)(0) is tainted
6 R(MyActivity,onCreate,11)(ret) <= R(MyActivity,onCreate,12)(2)
7 Taint SH(MyActivity,URL) if R(MyActivity,onCreate,12)(2) is tainted
8 Taint R(MyActivity,connect,1)(1) if SH(MyActivity,URL) is tainted
9 Taint R(MyActivity,connect,3)(0) if R(MyActivity,connect,2)(1) is -> tainted
10 3 Check R(MyActivity,connect,3)(0) LEAK
```

Listing 18: Rules that make the life cycle leak detectable
6.3 Implicit flow leak

The last example (Listing 19) shows a leak that is not detected by our prototype (and similar tools). On the main activity creation, the last device geo position is read from the LocationManager. Then, if the predicate isInTown returns true (for example if the user is in a certain area) an http request is made to an url that register the visit. The location data returned by a source is not leaked to the public, however the application is still able to detect if the user is in a desired location.
public class MyActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my_activity);
        Location l = getLocation();
        if (isInTown(l)) {
            try {
                URL url = new URL("http://www.trackusers.com/register");
                url.openConnection();
            } catch (Exception e) {}
        }
        private Location getLocation() {
            LocationManager locationManager = (LocationManager)
                    this.getSystemService(Context.LOCATION_SERVICE);
            return locationManager
                    .getLastKnownLocation(LocationManager.GPS_PROVIDER);
        }
        private boolean isInTown(Location loc) {
            ...
        }
    }
}

Listing 19: Indirect leak
Chapter 7

Conclusions

In this thesis we have formalized a compositional flow logic model for information flow control of Dalvik bytecode, based on the $\mu$-Dalvik$_A$ formal model, that considers the unique life cycle and inter-component communication system of the applications running in the Android OS. We have then enriched the flow logic with a taint information model, to detect leaks of sensitive data. We have also developed a prototype that applies the flow logic judgments using SMT constraints, highlighting the challenges of an actual implementation, like the encoding of a logical system for the resolution with an SMT solver, the choice of the abstract domains and the analysis of API methods. Although we have not provided a formal proof of the correctness of our flow logic, our work is based on strong theoretical foundations [7] [24] that give assurance on the soundness of our approach.

Future work includes: (i) a formal proof of our flow logic; (ii) an improvement of the objects and intents sensitivity; (iii) a formalization of the Android services, broadcast receivers and content providers; (iv) a stub implementation of the key Android API methods that would make the analysis more precise.
Bibliography


[18] Yin Liu and Ana Milanova. Static information flow analysis with handling of
implicit flows and a study on effects of implicit flows vs explicit flows. *2010
14th European Conference on Software Maintenance and Reengineering*, pages


[20] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning ap-
proach for classifying and categorizing android sources and sinks. In *NDSS*,
2014.

[21] Raja Vallée-Rai Phong Co Etienne Gagnon Laurie Hendren Patrick Lam Vi-

[22] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine

[23] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your

[24] Erik Ramsgaard Wogensen, Henrik Søndberg Karlsen, Mads Chr. Olesen, and
René Rydhof Hansen. Formalisation and analysis of dalvik bytecode. *Science