Faculty of Natural Science and Technology I
Department of Computer Science

Master's Program in Computer Science

Master's Thesis

Static Analysis
of Android Applications

submitted by

Ilya Grishchenko

on February, 12th, 2014

Supervisor
Prof. Dr. Matteo Maffei

Reviewers
Prof. Dr. Matteo Maffei
Prof. Dr. Christian Hammer
Eidesstattliche Erklärung
Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, .................................................. ...................................................
(Datum / Date) (Unterschrift / Signature)
Acknowledgements

This thesis would be impossible without the help and the advice of a number of people, some of whom I would truly like to thank.

First of all, I am thankful to Matteo Maffei, for the possibility to investigate the interesting and challenging research topic and for his supervision.

I would especially like to thank Manuel Reinert for the review of my thesis, his patience and insightful comments.

Also I thank Alexander Andreychenko for the proofreading of my work.

I am deeply grateful to my parents and friends for their support.
Abstract

Mobile and portable devices are machines that users carry with them everywhere, they can be seen as constant personal assistants of modern human life. Today the Android operating system for mobile devices is the most popular one and the number of users still grows: as of September 2013, 1 billion devices have been activated [Goob]. This makes the Android market attractive for developers willing to provide new functionality. As a consequence, 48 billion applications ("apps") have been installed from the Google Play store [BBC].

Apps often require user data in order to perform the intended activity. At the same time parts of this data can be treated as sensitive private information, for instance, authentication credentials for accessing the bank account. The most significant built-in security measure in Android, the permission system, provides only little control on how the app is using the supplied data.

In order to mitigate the threat mentioned above, the hidden unintended app activity, the recent research goes in three main directions: inline-reference monitoring modifies the app to make it safe according to user defined restrictions, dynamic analysis monitors the app execution in order to prevent undesired activity, and static analysis verifies the app properties from the app code prior to execution.

As we want to have provable security guarantees before we execute the app, we focus on static analysis. This thesis presents a novel static analysis technique based on Horn clause resolution. In particular, we propose the small-step concrete semantics for Android apps, we develop a new form of abstraction which is supported by general theorem provers. Additionally, we have proved the soundness of our analysis technique.

We have developed a tool that takes the bytecode of the Android app and makes it accessible to the theorem prover. This enables the automated verification of a variety of security properties, for instance, whether a certain functionality is preceded by a particular one, for instance, whether the output of a bank transaction is secured before sending it to the bank, or on which values it operates, for instance, whether the IP-address of the bank is the only possible transaction destination.

A case study as well as a performance evaluation of our tool conclude this thesis.
Chapter 1

Introduction

Android platform [Andd] applications have become targets for numerous attacks within the last years: it was shown that the user’s private data protection is a main concern, whereas applications can leak these data without any notification [EGC+10]. Android devices have at their disposal various sorts of private information that usually include user typed input (e.g., passwords or credit card numbers), GPS location, local storage, contacts, etc. All these data can be accessed by apps in order to provide their functionality. Although this is mostly done with generous ideas in mind, the executable code of the app might be inconsistent and therefore increase the potential of private information leakage. This potential can be controlled only by one standard technique in Android, namely, the permission system which allows a developer to specify all possible sorts of data that the application can access. Once granted at install time these permissions cannot be modified and some of them are imprecise. At the same time, this approach does not control propagation of private data: one cannot distinguish whether the data is available only inside the app or whether it is sent out [AZH+11]. Additionally, developers frequently use more permissions than necessary [FCH+11] to have the full functionality, while users do not devote a lot of attention to the permission requests and tend to confirm them all if they like the app. Hence, this realm alone does not suffice to enforce reasonable data protection. The most recent publication [Lea] even shows that hidden vulnerabilities in the apps (especially the most popular kind - apps for entertainment) were used by governments in order to obtain private information of users.

This manyfold problem can be tackled in different ways. One possible technique is so-called inline-reference monitoring (IRM) that helps to overcome the limitation of Android’s permission system via rewriting the app code [Erl04]. IRM provides the user with the possibility to still use the app but with user refined permissions, for instance, one can allow for accessing a certain website, while before the only option was to fully disable Internet activity. This powerful and successful method, however, has limitations: the performance of the application depends on the adjustments adopted to it by the rewriting tool which is a serious problem for the developer and, at the same time, the user refines permissions, but one cannot show that this prevents data leakage (permissions that allow leakage may remain intact if the user does not modify them).
Two other techniques are dynamic and static app analysis. Dynamic analysis creates a monitor that observes the app’s behavior and prevents undesired activity. On the one hand, this method has strong properties: fine-grained control over the execution, the absence of false positives (e.g., if the app is detected to leak data, it really does) and immediate reporting about found attacks. On the other hand, if an app has not been caught sending out user data, it can still be potentially dangerous. Moreover, users have to modify devices’ firmware to use dynamic analysis.

In contrast to the two techniques mentioned above, static analysis does not require any modifications on the device or/and to the app and provides guarantees for its results even without execution. However, it has one strong limitation: static analysis assumes an over-approximation of the app’s behavior. All potential states of a real program execution are in the subset of the static analysis states for the same app. Therefore, found misbehavior may exist only in the analysis and does not need to map to a real attack.

In this thesis, we take a basic static analysis approach to provide guarantees for satisfying the predefined properties without an actual execution. In other words, we intend to establish the trust of the user to the application before the execution. We extend the applicability of static analysis to Android apps allowing for handling even more complicated cases of the app’s behavior. For example, when a user would like to send his credit card information to the bank to perform a transaction with an Android app, the private information source is the user input (e.g., credit card data) that our app eventually will sends out to the Internet, as it should reach the bank side. In this situation all existing solutions for app analysis report that there is a data leakage and those based on static analysis even prove that. However, there is no "leakage", if the data was secured and, at the same time, sent to the bank only. Our approach is able to solve this problem and others involving precise analysis of the app execution.

The verification of primitives that are used in order to secure applications (e.g., cryptographic protocols and their implementations) are in the focus of the scientific community. Hence, we assume their correctness and devote our attention to the way these primitives are used by the particular app.

**Our Contributions.** In this thesis, we firstly provide the formalization of the entire Android app execution environment in form of the concrete small-step semantics. This formalism is of independent interest, for instance, for software verification.

Using the concrete semantics as a reference we provide the definition of a novel static analysis technique, based on abstract interpretation [CC77]. It can be applied to any possible kind of Android app and is mostly automated.

Given this analysis technique and the full semantics of the app’s bytecode we show their correspondence, namely, the soundness of our analysis technique. This guarantees that all shown properties for the abstraction of the app in the analysis carry over to the real app.

In order to demonstrate the applicability of our approach, we have implemented a prototype. The analysis is performed by a state-of-the-art theorem prover, which makes the results truly reliable.
We also have designed a special case study: an app for an ad-hoc remote e-voting system. It is of independent interest by itself since it is the first known app of this kind [Andc].

1.1 Related Work

We provide a short overview of the research within the area in the same order as before.

IRM. AppGuard is a powerful approach making use of inline-reference monitoring, introduced by Backes et al. [BGH+13]. AppGuard overcomes the limitations of the Android permission system allowing for user-driven permission refinements treated as security policies. These policies are enforced by rewriting the source code of the app, in other words the "monitor" is embedded inside the executable code as a wrapper for the functionality. AppGuard has extremely little run-time overhead and has shown the elegant way of handling such complicated structure as a compiled code of the Android app.

Dynamic analysis. Dynamic analysis was implemented in TaintDroid [EGC+10]. It introduces a monitor for real-time tracking of tainted user private data and checks whether it can reach the information sink. While TaintDroid tracks only explicit flow (leakage happens explicitly), a follow-up approach [GCBCC12] can also tackle implicit flow (leakage happens through the control flow). Dynamic approaches involve run-time overhead (e.g., for TaintDroid it may be up to 14%) and involve the modification of the device firmware. Our approach does not have this limitation.

Static analysis. Static analysis does not involve the app execution to proceed. The authors of [PS12] present an approach that can check various properties: class cast, nullness, termination, etc., however, it is not security-oriented.

A special category of static analysis approaches tackles the privileges escalation problem [DDSW11] when one app can exploit privileges delegated to another app. In order to prevent this, ComDroid [CFGW11] checks messages sent between apps. The authors of the tool Epic [OMJ+13] present a technique that reduces the problem of Inter-Component Communication to the Interprocedural Distributive Environment (IDE) [SRH96] and achieves higher precision than ComDroid. Another approach CHEX introduces a sound static analysis tool [LLW+12] for component hijacking. A similar approach is found in [CHY12]. The paper by Bugliesi et al. [BCS13] presents Lintent, a sound type system for privilege escalation detection. Lintent uses the source code of the app as the input, hence, it is developer-oriented. We are not targeting the problem of privilege escalation but our technique can be extended with ideas evolved in the approaches in this section. Especially, we can implement a privilege escalation detection mechanism that is close in spirit to the Lintent but uses bytecode, and adopt the sound method for entry points detection [CHEX] in order to avoid using only domain knowledge.

The tool ScanDroid [FCF09] is not limited to the privilege escalation detection. This data-flow analysis approach can also be used to find data leakage in a program, however,
it requires the source code of the application (or at least its compiled Java bytecode) to be available in order to use/extend the techniques that are available for Java bytecode. A newer approach FlowDroid [FAR+13] demonstrates the high precision, but also works with Java bytecode only. We do not introduce such a requirement, while our approach is as well context-, object- and field- sensitive.

SCANDAL [KYYS12] presents a sound technique for user data leaks: the program counter associated with private data is determined at creation time and is then tracked across the states. A leak is detected if a sink contains this program counter. The authors also instrumented the 220 most popular Android API methods such that they can be used in their analysis. Another approach [GCEC12] detects leaks via tainting data sources from sinks and checking whether their forward slice contains the API method that can output the data. Our approach can be used to perform similar analysis to the two mentioned above, but is no limited to the above properties.

We have already mentioned that we do not focus on the primitives for securing applications, but rather on their usage in the app. We consider these modules to be trusted. However, we mention techniques that introduce realms for verification of such primitives. Dupressoir et al. [DGJN11] develop the notion of log (we use something close in spirit as nothing can be "forgotten" in the logic). Two other approaches [AGJ11, BFGT08] show that it is possible to have implementations of cryptographic functionality verified to some extent. Android allows for execution of native code, hence this functionality can be used in apps as a native binary.

1.2 Outline

This thesis proceeds as follows: Chapter 2 presents the key ideas underlying our analysis, Chapter 3 describes the Dalvik Bytecode small-step concrete semantics of Android apps, Chapter 4 demonstrates our analysis technique, Chapter 5 relates the concrete semantics to our analysis by the proof of soundness, Chapter 6 presents our tool prototype description, and Chapter 7 shows a performance evaluation of our tool. Finally, Chapter 8 concludes this thesis.
Chapter 2

Key Ideas

In this Chapter we provide the general description of the Android operating system and its apps as well as the main ideas of our approach: the static analysis target, the static analysis technique, and Horn Clauses [Hor51].

2.1 Apps in Android OS

We outline the general concepts of the Android Operating System. An overall simplified structure is provided in Figure 2.1 [EOMC11].

![Android OS simplified structure](image)

All hardware of the device (e.g., Display, Bluetooth) is accessed by a modified Linux system. Both the Binder component (acting as a middleware) and APIs run on top of the Linux system. An app accesses the device’s hardware only through these APIs. To have access to a specific API every app must declare it as a permission request in a special file called manifest [?]. The manifest file is parsed at install time and users are asked whether they allow an app to use permissions. Apps are separated: each of them is running in a separate copy of the Dalvik Virtual Machine with a unique UID but they can communicate with each other through Inter Process Communications (IPC). Binder
mediates every IPC. Applications can be separated into two big classes: system apps (pre-installed) and the apps that the user might install additionally.

An Android app is written in a Java language environment using a special Android API mentioned in the previous paragraph. In order to get executed, apps are compiled first to Java bytecode .class files compatible with a Java Virtual Machine [JVMa] and then to one .dex bytecode file which is Dalvik Virtual Machine (DVM) compatible [Dalb]. So, the code that is executed eventually is Dalvik bytecode.

2.2 Static Analysis Target

As we have described in Section 2.1, the app we would like to analyze exists in three versions: Java source, JVM bytecode, and DVM bytecode. Generally, static analysis can be performed on any of those, as it can be applied both to application source code or compiled (object) code - bytecode. Java source code analysis is well developed [Ham09], however, this technique has two requirements. The first one is the source code of the app at our disposal which we, in general, do not have (e.g., when we are not the developers of the app). The second is the verified compiler for Android applications, because the properties that have been shown for the source code should carry over to the compiled code. Compiler verification is a hard task on its own [Loc10]. In our case it is especially complex as the app first gets compiled and then even transformed additionally by the special tool dx. Therefore, we focus on performing object code analysis, so we end up with two options: JVM bytecode [JVMb] and DVM bytecode [Dala]. Again, as for source code, we do not have JVM bytecode, only Dalvik bytecode. JVM bytecode can be recovered from Dalvik bytecode [OJM12]. However, this step depends on the effectiveness of the tool that performs re-targeting. In addition, there are classes that cannot be re-targeted and, thus, their functionality is out of the analysis scope. Hence, we work with Dalvik bytecode directly, targeting the code which is executed on an Android device.

Although Java bytecode analysis is well developed [BDFL+08, ABDF03], we cannot reuse its achievements directly and require a special analysis technique for Dalvik bytecode because of the following reasons:

- the Dalvik virtual machine is register-based, as opposed to the JVM which is stack-based; the Dalvik bytecode opcodes directly work with registers, instead of accessing program stack elements;
- there are multiple entry points for the app, while in case of Java there is only one, the main method;
- Java bytecode is converted into a substantially different instruction set, not syntactically, but by nature:
  - (a) in Java bytecode there are special operational codes that move variables between the stack and the local variable table which are absent in Dalvik as expected since it is register-based;
(b) Dalvik instructions are longer: they include source and destination in a single operation;

(c) opcodes for control flow (loops, switches, exception handlers) have a different structure in Java and Dalvik Virtual Machines;

(d) Dalvik bytecode has no Null type, the constant 0 is used instead, hence, the 0 constant value is equivocal, it can be both a constant integer with this value or the Null reference. As a consequence, Dalvik does not have special comparison opcodes for objects and null references. Instead, general zero equality checks are applied.

2.3 Static Analysis Technique

We have chosen an abstract interpretation as the analysis technique. Abstract interpretations introduced in [CC77] can be seen as the most general framework for any kind of static analysis of programs (apps). This shows the power of the abstract interpretation: because of its generality, our concept can be extended in many ways providing a solid base for future research.

In order to define an abstract interpretation, we should first of all derive concrete semantics of the program in the form of a mathematical object. This object is infinite as it represents all possible executions of the app in all possible execution environments. As we cannot operate on infinite objects, we further define an abstract semantics which is a superset of the concrete semantics. In other words, abstract interpretation forms are sound approximations.

We express the abstract interpretation in terms of Horn clauses [Hor51]. They are defined as logical formulas. In the following, we specify the categorization of Horn clauses explaining how the real program can be reflected in both concrete semantics and as an instance of our analysis technique:

(a) Fact - encoding of a logical assumption that a certain logic formula holds. In our setting facts are sets of logical predicates of different arity that describe the analyzed app. Example 2.1 shows how one of the possible facts reflects the Dalvik bytecode.

Example 2.1. Consider the following bytecode from the real application:

```java
.class LMainActivity$1;
...
.method public onClick(LView;)V
...
  5 new-instance v0, LSingleChoiceBallot;
  ...
```

This piece of code defines the creation of a new object LSingleChoiceBallot; (c') in class MainActivity$1 (c), method onClick(LView;)V (m), and program counter
5 \((pc)\) and it places the reference to the new object instance in the register \(v0\) \((i)\). Based on that we create a logical formula of the form \textbf{new-instance} \(c\) \(m\) \(pc\) \(i\) \(c'\):

\[
\text{new-instance} \text{MainActivity$1$} \text{onClick(Landroid/view/View;)}\text{V} \text{5}
\]

\[
0 \text{Lcivitas/common/SingleChoiceBallot;}
\]

We observe that the logical fact in this case is almost a one-to-one correspondence to the bytecode.

(b) \textit{Definite clause} - encoding of a logical implication, i.e., the assumption that a formula holds if a given set of logical formulas holds. This is the crucial part of the analysis, since it allows us to logically formalize the app’s behavior, the evolution of the different states of the app execution. This formalization should be derived from the full semantics of the Dalvik bytecode in order to be valid. Therefore, we exemplify how a concrete semantics rule is transformed to its logical equivalent.

\textbf{Example 2.2.} In the \textbf{Example 2.1} we explain in detail what instruction \textbf{new-instance} does during the execution. We execute the app, hence, we have a local state \(c.m.pc :: C\) that tells that we execute an instruction on line \(pc\) \((e.g., 5)\) of method \(m\) \((\text{onClick(LView;)}\text{V})\) of class \(c\) \((\text{MainActivity$1$})\). This local state is mapped to the registers. It operates on \(R\) \((\((c.m.pc) :: C \mapsto R\))\). We test \(P(c.m.pc) = \textbf{new-instance} \ i \ c', \ i.e. \ whether \ we \ indeed \ reach \ the \ instruction \ \textbf{new-instance}, \ where \ \(i\) \ is \ a \ register \ number \ \(0\) \ and \ \(c'\) \ is \ the \ name \ of \ the \ class \ to \ be \ created \ (LSingleChoiceBallot;). Then \ we \ declare \ \(o = f \mapsto \emptyset\) \ a \ new \ object \ instance \ of \ a \ particular \ class \ \(c'\) \ with \ all \ its \ fields \ \(f\) \ \(i.e., c', f \in P(c'.f)\) \ mapped \ to \ the \ undefined \ values \ \(0\). \ Afterwards, \ we \ place \ \(H' = H[\{(\ell, (c.m.pc) :: C, c', 0) \mapsto o\}\] \(the\ \new{}\ object\ \with\ version\ \emptyset\ \on\ \heap\ \in\ \ell, \specifying\ its\ creation\ point,\ class\ name,\ and\ version: \ (c.m.pc) :: C, c', 0\). \ This \ reference \ is \ unique \ \(because\ \it{it\ is\ impossible\ in}\ \Dalvik\ \bytecode\ \to\ \create\ \multiple\ \instances\ \of\ \the\ \same\ \type\ \in\ \one\ \step\), \ hence, \ there \ is \ no \ check \ whether \ \(\text{dom}(H)\) \ \already\ \contains\ \the\ \same\ \element. \ \Then\ \we\ require\ the\ \last\ \precondition\ \(R' = R[\ell \mapsto (\ell, (c.m.pc) :: C, c', 0)\]\) \that\ \describes\ \the\ \register\ \modification:\ \register\ \(i\) \ \specified\ \in\ \the\ \code\ \is\ \mapped\ \to\ \the\ \object\ \reference. \ As\ \a\ \consequence\ \of\ \the\ \instruction\ \textbf{new-instance} \ \we\ \will\ \have\ \a\ \new\ \mapping\ \starting\ \from\ \the\ \next\ \program\ \state\ \(c.m.pc + 1\) \ \to\ \the\ \registers\ \(R' ((c.m.pc + 1) :: C \mapsto R')\) \ and\ \the\ \new\ \heap\ \(H'\). \ We\ \compactly\ \represent\ \pre-conditions\ \and\ \post-conditions\ \in\ \the\ \following\ \rule:

\[
\begin{align*}
\text{NEW-INSTANCE} & / P(c.m.pc) = \textbf{new-instance} i c' \\
c'.f & \in P(c'.f) \\
o & = f \mapsto \emptyset \\
H' & = H[(\ell, (c.m.pc) :: C, c', 0) \mapsto o] \\
R' & = R[\ell \mapsto (\ell, (c.m.pc) :: C, c', 0)] \\
& \langle ((c.m.pc) :: C \mapsto R), H, S \rangle \mapsto p \langle ((c.m.pc + 1) :: C \mapsto R'), H', S \rangle
\end{align*}
\]
We transform the above rule into the following Horn Clause, where all variables are implicitly universally quantified:

\[
\begin{align*}
\text{NEW} - \text{INSTANCE} & \quad \text{new-instance } c \ m \ pc \ i \ c' \\
\land \text{field } c' f & \quad \Rightarrow \land \ h ((c \ m \ pc) :: C \ c' 0) \\
\land r (c \ m \ pc) :: C i \ val_r & \quad \land \hat{r}_{\neq i} (c \ m \ pc+1) :: C
\end{align*}
\]

Every part of the concrete semantics rule is reflected in the abstract rule. This is needed to further show the soundness of our approach. The latter is of certain need, otherwise our approach does not fulfill the abstract interpretation definition.

While in the concrete semantics we have \(P(c.m.pc) = \text{new-instance } i \ c'\), in the abstract semantics we have \(\text{new-instance } c \ m \ pc \ i \ c'\) which creates the logical encoding of the information in the concrete semantics that checks the fact of the instruction presence in the app. The check for all fields \(f\) that are declared to be the non-static \(c'.\overline{f} \in P(c'.\overline{f})\) is transformed into \(\text{field } c' f\). The logical statement can be true for multiple field names \(f\) because of the universal quantification. The three next concrete statements:

\[
o = f \mapsto 0 \\
H' = H[(\ell, (c.m.pc) :: C, c', 0) \mapsto o] \\
R' = R[i \mapsto (\ell, (c.m.pc) :: C, c', 0)]
\]

are defined in the analysis as the logical conjunction of \(h ((c \ m \ pc) :: C \ c' 0) f 0\) and \(\hat{r}_{\neq i} (c \ m \ pc+1) :: C\) and \(r (c \ m \ pc + 1) :: C i ((c \ m \ pc) :: C c' 0)\), respectively. As we have \(R'\) and \(H'\) as the result of the step, their logical encodings are brought to the right side of the implication to reflect the reasoning model (i.e., they are the consequence of the given knowledge).

Both rules look similar, however there are certain differences: there is no memory location \(\ell\) in the logic, as we do not model the memory.

(c) **Goal clause** - logical statement which should be proven. We can see this category as the statement about the app that we would like to check. This last type of Horn clauses being accepted by our analysis technique demonstrates its versatility.

**Example 2.3.** For instance, we may ask our analyser to prove

\[
r \ \text{MainActivity}\$1 \ \text{onClick}(LView;)V \ 6 :: C \ 0 \\
(MainActivity\$1 \ \text{onClick}(LView;)V \ 5 :: C \\
LSingleChoiceBallot; 0)
\]

In other words, we check whether the register 0 has a reference to the class \(LSingleChoiceBallot\) on program counter 6. This proof succeeds as the negation of this statement is unsatisfiable given the fact from Example 2.1 and the definite clause from Example 2.2.
2.4 Termination

The abstract semantics operates on the sets of all possible values for each of the program states while the concrete semantics has exact values. Because of the aforementioned properties one should carefully choose abstraction domains and handlers for operations with loops in order to achieve termination. Dalvik bytecode analysis is not trivial regarding loops. There are no specific operational codes for loops, hence, one cannot distinguish easily between a conditional branch or a cycle, while both are stated by using the same primitives, namely conditionals (e.g., \texttt{if}, \texttt{ifz}). Cycles have an additional unconditional (e.g., \texttt{goto}) jump at the end, so as to check the new value of the guard but it is not immediately seen in the beginning of the iterations. However, as we can see in the examples from Section 2.3, our analysis technique description is expressive but not complicated. The first technique that we use to handle loops is called \textit{fixed-points}. \(X\) is a fixed point of function \(F\) if and only if the following equation holds:

\[ F(X) = X \]

In our case \(F\) is the analysis itself and \(X\) is the analysis’ input (e.g., Horn Clauses): facts about the concrete program and general rules of Android’s app execution together with the queries. When new Horn Clauses cannot be derived we have reached a fixed-point.

We use another concept called \textit{widening} [CC76] which allows us to find \(X\) as a fixed point even in presence of infinite loops. It uses the widening operator to infer the properties of the abstract values to hold in order for the loop to terminate and then checks the rest of the program with the inferred properties for the abstract values added to the fixed-point.

2.5 Outline of an Approach

Our approach involves the following components:

1. an app to analyze;
2. a formal model of the app in the concrete semantics, the model is universally valid;
3. a sound abstraction of the formal model based on Horn clauses;
4. an implementation that accepts Horn clauses as a base construct, hence, it directly corresponds to the abstraction.

These components and the relations between them are depicted in Figure 2.2.
2.5 Outline of an Approach

Figure 2.2 Outline of an approach
Chapter 3

Dalvik Bytecode Concrete Semantics

This section describes the Dalvik bytecode and the, to the best of our knowledge, first known formalization of the full Dalvik bytecode semantics. The most crucial part of this chapter describes the state in a program execution as well as the rules that define transitions between states. These rules are based on the formalization of the generalized version of the full Dalvik bytecode opcodes set [Dala].

3.1 App Definition

Applications in Android are sets of classes compiled to be executed by the Dalvik Virtual Machine. Class definitions are placed inside one special .dex file that contains all relative fields and methods. In order to formalize apps we use the following concepts:

- **mappings**, e.g., a class name maps to the class definition $c \mapsto \tau_c$, in other words, knowing the class name we can access its definition;
- **vectors**, e.g., an app usually consists of a number of class definitions which we express as a vector of mappings between classes and their definitions $\vec{c} \mapsto \vec{\tau}_c$.

In the following we provide the formal app’s definition in the concrete semantics.

**Definition 3.1.** We define an app $P$ as a vector of mappings from class names $c$ to their definitions $\tau_c$, from pairs of class names $c$ and field names $f$ to field definitions $\textit{field}$, from pairs of class names $c$ and method names $m$ to method definitions $\textit{method}$ and from triples of class names $c$, method names $m$, and program counter $pc$ to the instructions $\textit{inst}$ in bytecode. This can be represented as follows:

$$P := (\vec{c} \mapsto \vec{\tau}_c, \vec{c}.f \mapsto \vec{\textit{field}}, \vec{c}.{m} \mapsto \vec{\textit{method}}, \vec{c}.{m}.pc \mapsto \vec{\textit{inst}})$$

We do not declare a specific type "interface" as interfaces are essentially classes having no mapping from the method name $m$ to the actual $\textit{method}$ definition. We do not have such a mapping in $P$. 
We continue the formalization with definitions of the app’s components: classes, methods and instructions.

**Definition 3.2.** We define a class definition $\tau_c$, as a tuple consisting of a super-class $c_{\text{sup}}$, a vector of interfaces $c_{\text{imp}}$, vectors of fields $c_{\vec{f}}$, and vector of methods $c_{\vec{m}}$. We define field to be its type $\tau$. We define method as a tuple containing the number numloc of local registers it operates on, a vector of parameter types $\tau_{\text{arg}}$, a return type $\tau_{\text{ret}} \mid \text{void}$, and the vector of instructions $c_{\vec{m}, \vec{p}}$. The short description is provided below:

$$
\tau_c ::= (c_{\text{sup}}, c_{\text{imp}}, c_{\vec{f}}, c_{\vec{m}}) \\
\text{field} ::= \tau \\
\text{method} ::= (\text{numloc}, \tau_{\text{arg}}, \tau_{\text{ret}} \mid \text{void}, c_{\vec{m}, \vec{p}})
$$

The possible bytecode instructions inst include arrayData, packedSwitch, sparseSwitch, and opcode. The concrete semantics represents arrayData, packedSwitch, and sparseSwitch in the same manner as they appear in the bytecode: arrayData is a vector of values $\overrightarrow{\text{val}}$, which is encoded in the bytecode, sparseSwitch is a vector of mappings between integers $\overrightarrow{\text{key}}$ and program counters $\overrightarrow{\text{pc}}$, packedSwitch is a vector of program counters $\overrightarrow{\text{pc}}$ and the additional integer $\text{fst}$ which determines the value of the first case (the rest of the cases are computed accordingly). The generalized set of all possible opcode instructions that is used in an approach is given in Table C.1.

### 3.2 Types Description

While defining the app we have already mentioned types $\tau$. The type structure of Dalvik bytecode is fully reflected in our semantics.

**Definition 3.3.** We define the type $\tau$ to consist of base type $b$ (e.g., int, float, etc.) and a special type ref, which is recursively defined to be a class name $c$, an array $\tau_a$ of $b$ type elements, an array of class names $c$, etc. We call it ref because this type is assigned to the elements which are stored on the heap and hence need a reference to be stored in registers.

$$
b ::= \text{byte} \mid \text{int} \mid \text{short} \mid \text{long} \mid \text{float} \mid \text{double} \mid \text{char} \mid \text{bool} \\
\tau ::= b \mid \text{ref} \\
\tau_a ::= \text{array of } \tau \\
\text{ref} ::= \tau_a \mid c
$$

Notice that strings are not arrays of chars: they are immutable, like in Java. There is no type Null for the reference: Dalvik Bytecode uses a constant $\text{null}$. We refer the interested reader to Table A.1 where we collect all the definitions from Sections 3.1 and Section 3.2. The definition of sub-typing that we introduce in Table A.3 is general.
3.3 Modeling the Execution State

This section provides the formal definition of the app’s execution. It is context-sensitive: inter-procedural calls jump back to the saved call site.

Context. The local state determines which instruction we execute next in the local method. Our program $P$ contains the mapping between the instructions and the triple $c.m.pc$. We denote by $c.m.pc$ the local state which consists of the program counter $pc$ in the method $m$ of class $c$.

When a method calls another method we store the point of invocation in the call stack, which is a recursive structure. Hence, the call stack can be seen as the list of triples $c.m.pc$.

$$L ::= c.m.pc$$

Local state

$$C ::= L :: C \mid L$$

Call stack

Register. Every method in Dalvik bytecode stores its local values in the registers. We denote the array of registers $R$ to be the vector of mappings of the registers number $i \in \mathbb{N}$ to the values $val_\tau$ of type $\tau$:

$$R ::= (i \mid ret) \mapsto val_\tau$$

The special register $ret$ is used to store the last method’s return value if there is one.

Dynamic heap. The execution of every method can create objects and arrays. We define an object instance $o$ as the vector of mappings from fields $f$ to values $val_\tau$ of type $\tau$ and the array $alpha$ as a vector of values $val_\tau$:

$$o ::= f \mapsto val_\tau$$

$$alpha ::= val_\tau$$

Objects $o$ and arrays $alpha$ are stored on the heap.

We denote the heap $H$ to be the vector of mappings between memory objects and arrays to their addresses $\ell$. We annotate the address of each element on the heap with the full call stack $C$ which determines the point of creation of an element, its type $c$ or $\tau_a$, and its version $ver$. It is needed further in the abstract semantics and without it, the proof of soundness fails.

$$H ::= (\ell, C, c \mid \tau_a, ver) \mapsto (o \mid alpha)$$

Static heap. Static class fields do not require any specific instance to exist in order to have a value (e.g., if they have the default values, they are created in the app right
after the start even if no instance of such a class is present on the heap). Hence, they are stored on the static heap. We denote the static heap $S$ to be the vector of mappings between fields $f$ of classes $c$ to values $val_r$:

$$S ::= c.f \mapsto val_r$$

**State.** In the following we define the state $\Sigma$.

**Definition 3.4.** We define the state $\Sigma_P$ to be the state in the execution pg the app $P$ that compounds all elements declared above as follows:

$$\Sigma_P ::= \langle (C \mapsto R), H, S \rangle$$

In order to operate on the value that belongs exactly to the current method we store the mapping $C \mapsto R$ between the call stack and the registers in $\Sigma_P$. So, the program state is a mapping between the call stack $C$ and registers $R$ together with the static heap $S$ and the dynamic heap $H$. The summary of this section is provided in Table A.2.

### 3.4 Operational Semantics

Given an app $P$ in a state $\Sigma_P$ we determine the next state $\Sigma'_P$ by a small-step semantics which specifies what changes in the program state while performing a step. A step in the small-step semantics is denoted as $\rightarrow_P$. The transitive closure of $\rightarrow_P$ is referred to as $\rightarrow^*_P$. The number of all possible kinds of steps is determined by the number of Dalvik bytecode opcodes.

We use the generalized instruction set which allows us to specify in a compact form the operational semantics for the entire Dalvik bytecode. The next subsections provide examples of selected rules from the entire set which is depicted in the Tables A.5 - A.12 in Appendix A. The operational semantic rules' description involve definitions of unary and binary operations in Table C.2 and some external functions that are defined in Table A.4.

#### 3.4.1 Operations over Registers

We start our description of the rules with a simple example of the MOVE instruction:

**Example 3.5.**

\[
\text{MOVE} / P(c.m.pc) = \text{move}_r \ i \ j \\
R[j] = val_r \\
R'[i] = val_r \\
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_P \langle ((c.m.pc + 1) :: C \mapsto R'), H, S \rangle
\]

From the rule we conclude that the current configuration is $\Sigma = \langle ((c.m.pc) :: C \mapsto R), H, S \rangle$. At the head of the rule we have a requirement for it to be executed: $P(c.m.pc) = \text{move}_r \ i \ j$. This means that the program $P$ which includes the mapping
between \(c.m.pc\) and instructions as was described in Section 3.1 should return \texttt{move} \(i \ j\) as an instruction placed on program counter \(c.m.pc\) in the app \(P\). Additionally we have the mapping between the call stack \(C\) and the set of registers \(R\), so we know all the values of registers that are used by the local state. The requirement \(R[j] = \text{val}_r\) takes the value \(\text{val}_r\) from register number \(j\). The statement \(R' = R[i \mapsto \text{val}_r]\) updates \(R\) at position \(i\) to \(\text{val}_r\). As a result of the \(\rightarrow_P\) step we have a new mapping in \(\Sigma_P\) such that 
\((c.m.pc + 1) \mapsto C \mapsto R'\), and the rest stays the same. It means that starting from the next program counter \(pc + 1\) in the same method \(m\) of the same class \(c\), register \(i\) has the new value \(\text{val}_r\) taken from register \(j\).

### 3.4.2 Program Counter Jumps

There is a set of operations that do not modify registers, the heap, and the static heap. They perform a jump from the current program counter \(pc\) to the new one specified in the instruction \(pc'\). Let us consider as an example the \(\text{SPARSE-SWITCH-TRUE}\) rule.

**Example 3.6.**

\[
\text{SPARSE-SWITCH-TRUE} / P(c.m.pc) = \text{SPARSE-SWITCH} \ i \ pc' \\
P(c.m.pc') = \text{sparseSwitch} \quad R[i] \in \text{sparseSwitch} \quad \text{sparseSwitch}[R[i]] = pc'' \\
\langle (\langle c.m.pc :: C \mapsto R \rangle, H, S) \rangle \\
\rightarrow_P \langle (\langle c.m.pc'' :: C \mapsto R \rangle, H, S) \rangle
\]

We use the fact that \(\text{sparseSwitch}\) is among the instruction set, so inside the bytecode, hence, it provides a \(pc'\) value. The statement \(P(c.m.pc') = \text{sparseSwitch}\) ensures that we have a sparse switch on the \(pc'\) specified in the instruction. After that we take the value of register \(i\) from the instruction and (knowing that \(\text{sparseSwitch}\) is a vector of mappings between integer values and program counters) check that \(i\) is in \(\text{sparseSwitch}\). If this is true we can take the value \(pc''\) which corresponds to \(R[i]\) in \(\text{sparseSwitch}\) as the next program counter value.

### 3.4.3 Operations with Heap Allocations

We demonstrate the operation over heap allocations on the example instruction \(\text{IPUT}\) that puts a value from the register to the object field on the heap.

**Example 3.7.**

\[
\text{IPUT} / P(c.m.pc) = \text{IPUT} \ i \ j \ c' \ f \\
R[i] = \text{val}_r \quad R[j] = (\ell, C', c', \text{ver}) \\
H[(\ell, C', c', \text{ver})] = o \quad o' = o[f \mapsto \text{val}_r] \quad H' = H[(\ell, C', c', \text{ver} + 1) \mapsto o'] \\
R' = R[j \mapsto (\ell, C', c', \text{ver} + 1)] \\
\langle (\langle c.m.pc :: C \mapsto R \rangle, H, S) \rangle \\
\rightarrow_P \langle (\langle c.m.pc + 1 :: C \mapsto R' \rangle, H', S) \rangle
\]
In the rule we specify that the register with number \( i \) contains the value \( \text{val}_\tau \), i.e., \( R[i] = \text{val}_\tau \), and the register with number \( j \) specified in the instruction must contain a reference to the instance of the class \( c' \), i.e., \( R[j] = (\ell, C', c', \text{ver}) \). Then we take the instance \( o \) from the heap with the specified address: \( H[(\ell, C', c', \text{ver})] = o \). After that we define the new object \( o' \) which is the same as \( o \) with the difference that the field \( f \) specified in the instruction is assigned a value \( \text{val}_\tau \), i.e., \( o' = o[f \mapsto \text{val}_\tau] \). We then perform two updates: the first \( H' = H[(\ell, C', c', \text{ver} + 1) \mapsto o] \) updates the version of the object instance and the second \( R' = R[j \mapsto (\ell, C', c', \text{ver} + 1)] \) updates its reference in the corresponding register \( j \), such that in the following execution it will point to the object with the changed value.

### 3.4.4 Function Calls

In order to speak about the execution of the app, we first of all specify what the initial state is. The initial program state requires all the static fields to be initialized on the static heap with default values. Android applications do not have any specific exit state, therefore, there is no formalization of program termination (e.g., \textit{stop}).

We continue the description of function calls with demonstrating how a method is resolved. We perform this resolution based on the information that is included in the \( \tau_c \) definition, in other words, for each method invocation we statically know which class implements it. Hence, we define our concrete semantics with resolved classes for each method call, following the standard for Java \([\text{Java}]\) resolution which Android agrees with.

It also includes simple cases of reflection when we know all the information statically: we tackle the bytecode operation \texttt{const-class} that creates a class from the user encoded string. If this class is present in the bytecode of the app (i.e., it is not a dynamically attached library) we can resolve any method of such a class. When defining the concrete semantics we assume to have all the necessary functionality at our disposal, the apps’ bytecode is provided together with the application or defined in standard libraries for Java and Android and, hence, known.

In other words, in the concrete semantics we assume that we know all the methods called and proceed as specified when method \( m \) of class \( c \) is called:

1. check whether the class \( c \) contains the implementation of \( m \); if yes - perform the invocation, else go to 2;

2. go recursively down in the inheritance chain following the class definitions \( \tau_c \); if the method’s implementation is found in the class \( c_{\text{sup}} \) - perform the invocation of the method \( m \) in class \( c_{\text{sup}} \).

We describe the concrete semantics of the function calls at the example of the \texttt{INVOKE-REF} rule. It has \texttt{-REF} in its name as it describes the semantics of the group of method calls where the first argument supplied should be the reference to the instance of the class to which the method belongs. In the bytecode there are four operations that are mapped to a single one in the operational semantics: \texttt{invoke-super, invoke-virtual,}
invoke-interface, and invoke-direct. The type of the call reflects the way the resolution is implemented: invoke-super invokes a method from the super class of the class mentioned in the instruction, invoke-virtual performs a call to the method implemented in this class or inherited from the super-class, invoke-interface invokes a method from the class which implements the interface mentioned in the instruction, and, finally, invoke-direct performs a call to a method, e.g. method constructor, that cannot be overridden.

Example 3.8.

\[
\text{INVOKEREF} / P(c.m.pc) = \text{INVOKEREF} \ c' \ m' \ i_j \ldots i_n \\
R[i_j] = (\ell, C, c'', \text{ver}) \\
c' \leq P \ c'' \ P(c'.m').\text{numloc} = \text{val} \text{int} \\
R[i_j] = \text{val} \ldots R[i_n] = \text{val} \text{'r} \\
R' = [0 \mapsto 0 \ldots \text{val} \text{int} - 1 \mapsto 0, \text{val} \text{int} \mapsto \text{val} \text{r} \ldots \text{val} \text{int} + n - 1 \mapsto \text{val} \text{'r}] \\
C' = (c.m.pc) : C \\
\rightarrow_p (((c.m.pc) :: C \mapsto R'), H, S)
\]

In order to perform INVOKEREF, first of all the register \( i_i \) supplied as a parameter to the function call should be a reference to the instance of the class which is in subtyping relation with the resolved class \( c' \leq P c'' \), the method of which we are executing. We have to know the number of local registers \( \text{numloc} \) because all the argument values are placed right after the local registers. We take this information from the method definition in the app \( P : P(c'.m').\text{numloc} = \text{val} \text{int} \). Then we take all the values from registers supplied as parameters of the method call: \( R[i_j] = \text{val} \ldots R[i_n] = \text{val} \text{'r} \). Using this information we create the initial register \( R' \) assigning all registers to \( 0 \) followed by parameter values \( \text{val} \ldots \text{val} \text{'r} : R' = [0 \mapsto \text{NULL} \ldots \text{val} \text{int} - 1 \mapsto \text{NULL}, \text{val} \text{int} \mapsto \text{val} \text{r} \ldots \text{val} \text{int} + n - 1 \mapsto \text{val} \text{'r}] \). In the last step we modify the call stack. We save the location from where the method was called \( C' = (c.m.pc) : C \) and place the new program counter \( c'.m'.0 \) on the top of it creating the mapping between this new call stack and the register \( R' : (c'.m'.0) : C' \rightarrow R' \).

Let us show that the called method can be finished with the rule for the method that returns a value.

Example 3.9.

\[
\text{RETURN} / P(c.m.pc) = \text{RETURN} i \\
\Sigma_P((c'.m'.pc') :: C) = R' \\
\rightarrow_p (((c.m.pc) :: (c'.m'.pc') :: C \mapsto R), H, S) \\
\rightarrow_p (((c'.m'.pc' + 1) :: C \mapsto R'[\text{ret} \mapsto R[i]]), H, S)
\]

When the method returns it proceeds with the tail of the call stack supplied as parameter to \( \Sigma \), which stores the mappings between the call stack and the register: \( \Sigma((c'.m'.pc') :: C) = R' \). As a result, we have the exact register \( R' \) that was present when the method was called (i.e., we know the register state of the callee). Afterwards we
proceed with assigning a special register $ret$ of the callee a value $val_r$ which is accessible from the next (to the call) program counter: $(c'.m'.pc' + 1) :: C \rightarrow R'[ret \mapsto R[i]]$. This is done due to the fact that the very next instruction after the method call should be MOVE-RESULT which takes the result of the last invocation and places it to the standard register specified in the instruction.
Chapter 4

Abstract Analysis Based on Horn Clauses

This Chapter defines an analysis technique that is based on Horn clauses. The analysis is precise for values that the app operates on. Our analysis technique is context-sensitive and models the call chain even in presence of recursion and exceptions. We define a novel representation of the class instance that introduces a notion of state for a (usually stateless) object. To our knowledge this is the first abstract interpretation technique for the Dalvik bytecode that allows a detailed description of the analyzed app together with a soundness property. We demonstrate the power of the analysis by a selection of rules.

We include in this section the description of the app in the analysis. Moreover, we show how we encode subtyping into the logic. The most crucial part of this chapter is a transformation of the operational semantics into Horn clauses. The transformation is done in such a way such that we are further able to show the soundness property of our analysis. Then we discuss the modeling of exceptions in the analysis that demonstrates precision on traversing the call chain in the case when an exception is thrown.

4.1 Abstract App Definition

In Section 3.1 we formally define the app using vectors and mappings. In our analysis technique we transform vectors into a set of logical formulas and encode mappings inside logical formulas. While the latter is intuitive, we motivate the absence of the vectors in the analysis technique description. We have performed a number of experiments how the state-of-art-theorem prover works with vectors represented as arrays. Unfortunately, we have discovered that the theorem prover cannot succeed when arrays are involved in the analysis. At the same, time tasks without arrays create no complications. Hence, we decide to omit arrays completely in order to achieve better correspondence between the theoretical approach description and its implementation.

Every app has its own structure. We have created all general Horn clause formulas which are expressive enough to represent any possible app, as our logical encoding includes all features introduced in the concrete semantics for the app description.
Definition 4.1. We denote an app \( \hat{p} \) to be the set of Horn clauses of the fact kind:

\[
\begin{align*}
\hat{p} & ::= \text{field } c' f \mid \text{move}_c, c m pc i j \\
& \mid \text{sparse-switch } c m pc i pc' \mid \text{sswitch-data } c m pc' \\
& \mid \text{invoke-ref } c m pc c' m' \mid \text{par } c m pc i ind \mid \cdots
\end{align*}
\]

There is a correspondence between the app definition in the concrete semantics and in the analysis, e.g., if in the concrete semantics we have \( P(c.m.pc) = \text{move}_c i j \), our analysis reflects the same in the fact \( \text{move}_c c m pc i j \), i.e., we encode the mapping inside the fact.

We use the same names for elements in both concrete and abstract semantics in order to avoid confusion, we just slightly change the representation of the program counter which is not \( c.m.pc \), but \( c m pc \). The encoding of \text{arrayData}, \text{packedSwitch}, and \text{sparseSwitch} involves two fact kinds: one determines the position (i.e., program counter) of each \text{arrayData}, \text{packedSwitch}, and \text{sparseSwitch} in the method (e.g., \text{sswitch-data } c m pc'). Another predicate (e.g., \text{sswitch-data-item } c m pc' ind val \text{int } pc'') we create for each element of \text{arrayData}, \text{packedSwitch}, \text{sparseSwitch}, i.e., we transform the vector in a set of logical predicates.

The fact \( \text{par } c m pc i ind \) is used to formulate the function call instruction: we cannot have general facts for every function call since they can have different number of parameters. The fact \( \text{par } \) connects the register \( i \) to its position \( ind \) in the arguments supplied to the function call. Two more special fact kinds are: \text{num-loc} that we create for each function in order to store the number of local registers for each function in the logic and \text{size} that indicates the size of the given array. The definitions from this section are summarized in Table B.1 in Appendix B.

4.2 Types in Horn Clauses

In the analysis we have the same types as in the concrete semantics, the definition of \( \tau \) does not change. However, subtyping changes: the sequence of typing rules in Table A.3 is encoded into the set of logical facts of the following form:

\[
\text{super } \tau \tau'
\]

This fact determines whether the type \( \tau \) is super-type of the type \( \tau' \). We include this kind of facts into the set of facts for app’s description \( \hat{p} \). Types of values in the abstract semantics are inferred from the types of the bytecode instructions they are operated by.

4.3 Analysis State

Given the set of facts representing the app, we use another kind of logical primitive also based on Horn clauses in order to be able to show the state of the app execution in
4.3 Analysis State

logic. In this section, we represent the register, the dynamic heap, and the static heap element by element (e.g., we have the value val\textsubscript{\texttau} that corresponds to the single register element) because of the already mentioned impossibility of effective work with arrays in the analysis implementation. We proceed in the same order as in Section 3.3.

**Context.** The call stack from the concrete semantics is declared to be the list where each element is a triple of class name \textit{c}, method name \textit{m} and program counter \textit{pc}.

\[
L ::= c\ m\ pc \quad \text{Local state}
\]
\[
C ::= L:: C \mid L \quad \text{Call stack}
\]

This definition fully corresponds to the one of the concrete semantics (the difference is only syntactic as we cancel (.) symbol).

**Register.** We denote the set of facts \(\hat{r}\) to be Horn clauses reproducing the mapping between register and the call stack.

\[
r ::= C\ (i\mid\text{ret})\ \text{val}_{\texttau}
\]

In the analysis we also have the register \(\text{ret}\) that contains the return value of the method call.

**Dynamic heap.** We represent the heap in the analysis as a set of facts \(\hat{h}\) that contain the reference to the object, the name of its fields and the values they are assigned.

\[
h ::= (C\ (c\mid\text{\tau_a})\ \text{ver})\ (f\mid i)\ \text{val}_{\texttau}
\]

We represent even a single object/array as a set of facts: one fact per field/index. Each fact then has the same value of the reference \(C\ (c\mid\text{\tau_a})\ \text{ver}\) as they belong to the same instance. The reference contains the version \textit{ver} which is a counter that we increment after every update of the object. The version allows us to distinguish different states of the same heap element in the abstract semantics, thus, we introduce a notion of the state to the heap elements which are usually stateless.

**Static heap.** In our analysis the static heap representation is the set of facts \(\hat{s}\), where \textit{s} is defined as follows:

\[
s ::= c\ f\ \text{val}_{\texttau}
\]

Our definition of the static heap is stateless. Hence, our analysis is imprecise, i.e., at any point in time we can prove any of the possible value assignments to the field name \textit{f}. However, the stateless static heap preserves soundness during the concurrent execution where we do not know statically which thread updates the static heap first.

**State.** In this paragraph we provide the definition of the analysis state. We intentionally omitted the exception handling for the moment, as we will use this definition further to show the soundness results for our analysis technique.
Definition 4.2. The overall abstract state is defined as the triple of three fact sets (a set of Horn Clauses that characterize the behavior of the app as defined above):

\[ \Delta ::= (\tilde{r}, \tilde{h}, \tilde{s}) \]

The definitions from this section are depicted in Table B.2 of Appendix B.

4.4 Abstract Rules for Operations

In the previous section we define the abstract state \( \Delta \) for the abstract program \( \tilde{p} \). In this section we show the evolution of this state on a selection of abstract rules which determine the relation between different states of the app abstraction. These rules correspond to the concrete semantic rules from Section 3.4.

4.4.1 Operations over Registers

We start the abstract rules description with the rule for move:

Example 4.3.

MOVE

\[
\text{move}\_\tau\; c \; m \; pc \; i \; j \land r\; (c \; m \; pc) :: C \; j \; val_\tau \implies r\; (c \; m \; pc + 1) :: C \; i \; val_\tau \land \tilde{r} \_\neq_i (c \; m \; pc+1) :: C
\]

The first statement move\_\tau\; c \; m \; pc \; i tells us that there is a move\_\tau instruction at the program counter \( c \; m \; pc \). From the next observation we state that we reach this instruction in the method \( m \) of the class \( c \) with the call stack \( C \) and the register \( j \) has a value \( val_\tau \): \( r\; (c \; m \; pc) :: C \; j \; val_\tau \). The type of the value in the abstract interpretation is inferred from the type of the instruction. Similar to the operational semantics we update the mapping between the register \( i \) specified in the fact (i.e., the destination register) and the call stack with the next program counter (which is the current one increased by one) on the top of it: \( r\; (c \; m \; pc + 1) :: C \; i \; val_\tau \). We require an additional element:

\[
\tilde{r}_{\neq i} (c \; m \; pc+1) :: C
\]

It expresses that at the next state all other registers except for \( i \) have the same values as at the current state. We require element of that format in our analysis as we operate over single elements, not vectors as before in the concrete small-step semantics description.

4.4.2 Program Counter Jumps

We explain the program counter jumps rules on the example of the sparse-switch-true rule.
**Example 4.4.**

**SPARSE – SWITCH – TRUE**

\[
\text{sparse-switch } c \ m \ pc \ i \ pc'
\land \text{sswitch-data } c \ m \ pc'
\land \text{r} (c \ m \ pc) :: C \ i \ \text{val}_{\text{int}}
\implies \text{r} (c \ m \ pc'') :: C \ w \ \text{val}_r
\land \text{sswitch-data-item } c \ m \ pc' \ \text{ind} \ \text{val}_{\text{int}} \ pc''
\land \text{r} (c \ m \ pc) :: C \ w \ \text{val}_r
\]

The statement **sparse-switch** \(c m pc i pc'\) specifies that the app has the instruction **sparse-switch** at \(c m pc\). The check whether there is a switch table on the \(pc'\) is specified in the fact **sswitch-data** \(c m pc'\). We take the register \(i\)'s (specified in **sparse-switch**) value \(\text{val}_{\text{int}}\) and \(\text{r} (c m pc) :: C \ i \ \text{val}_{\text{int}}\). It should be definitely integer as the specification requires. The fact **sswitch-data-item** \(c m pc' \ \text{ind} \ \text{val}_{\text{int}} \ pc''\) checks that the value \(\text{val}_{\text{int}}\) is among the values of the sparse-switch table, so we know the jump target corresponding to \(i\)'s value. As a result of the rule application we have a new mapping between call stack and registers encoded as follows: if there is a register predicate \(\text{r} (c m pc) :: C \ w \ \text{val}_r\) we have \(\text{r} (c m pc'') :: C \ w \ \text{val}_r\) (i.e., we transmit the values for predicates \(\text{r}\) of the same method from \(pc\) to \(pc''\)).

**4.4.3 Operations with Heap Allocations**

We exemplify how the operations over the heap are performed in the analysis with two rules. The first one is **iput**.

**Example 4.5.**

**IPUT**

\[
\text{iput}_r \ c \ m \ pc \ i \ j \ c' \ f
\land \text{r} (c \ m \ pc) :: C \ j (C' c' \ \text{ver})
\land \text{r} (c \ m \ pc) :: C' i \ \text{val}_r
\implies \text{h} (C' c' \ \text{ver} + 1) f \ \text{val}_r
\land \text{r} (c \ m \ pc + 1) :: C' j (C' c' \ \text{ver} + 1)
\land \text{h} \neq f (c \ m \ pc + 1) :: C
\land \text{h} \neq f (C' c' \ \text{ver} + 1)
\]

The fact **iput** \(c m pc i j c' f\) shows that the field name \(f\) of the class \(c'\) is updated with the value from the register \(i\). The fact \(\text{r} (c m pc) :: C' j (C' c' \ \text{ver})\) states that the register with number \(j\) contains a reference to the object of the \(c'\). Then we take the value of register \(i\): \(\text{r} (c m pc) :: C' i \ \text{val}_r\) and update the heap \(\text{h} (C' c' \ \text{ver} + 1) f \ \text{val}_r\). In addition, we require the update of the reference in \(i\): \(\text{r} (c m pc + 1) :: C' j (C' c' \ \text{ver} + 1)\). The following element determines that all fields (except \(f\)) of the object instance also obtain the new reference, but their values remain intact.

\[
\text{h} \neq f (C' c' \ \text{ver} + 1)
\]
The next element carries over all registers’ values (except for the updated $j$) to the next program counter without modification.

$$\hat{r} \neq j \ (c \ m \ pc+1):C$$

The next example rule creates the reference for the object of the class, whose name is specified as a string in the `const-class` opcode.

**Example 4.6.**

```plaintext
CONST − CLASS
const-class c m pc i str
∧ r (c m pc) :: C i val
```

$$\implies h ((c m pc) :: C String \ 0) \ value \ str$$

$$\land h ((c m pc) :: C Object \ 0) \ name \ ((c m pc) :: C String \ 0)$$

$$\land r (c m pc) :: C i ((c m pc) :: C Object \ 0)$$

$$\land \hat{r} \neq i (c m pc + 1):C$$

We create two objects on the heap: the object of the class `String` and the object of the class `Object` and place the reference to the last one in the register $i$ specified in the instruction.

### 4.4.4 Function Calls

We explain the function calls handling on the examples of the `invoke-ref` and the `return` rules. We start with the first one:

**Example 4.7.**

```plaintext
INVOKE − REF
invoke-ref c m pc c' m'
∧ par c m pc i ind
∧ r (c m pc) :: C i val
∧ par c m pc j 0
∧ r (c m pc) :: C j (C' c'' ver)
∧ super c'' c'
∧ numloc c' m' val_int
```

$$\implies r (c' m' 0) ((c m pc) :: C ind + val_{int} val)$$

$$\land \ 0 \leq ind' \leq val_{int}$$

The declaration `invoke-ref c m pc c' m' ∧ par c m pc i ind` encodes the fact that at the program counter `pc` of the method `m` of class `c` there is a function call with certain parameters. We do not have multiple parameter facts because of implicit universal quantification. The expression $r (c m pc) :: C i val$ takes the values from the
4.5 Modeling Exceptions

register predicate numbers that are specified as parameters for the function call. The part of the rule \( \text{par} \ c \ m \ pc \ j \ 0 \land \ r(c \ m \ pc) :: C \ j \ (C' \ c'' \ ver) \land \text{super} \ c'' \ c' \) tells us the following: \( j \) as the first parameter contains a reference to the object of the class \( c'' \) that is in the relation \( \text{super} \) with the class \( c' \) which method \( m' \) we are executing in this instruction. The method \( m' \) which we are calling might not be implemented in the class \( c'' \) that is supplied as the reference, hence, we require two classes \( c' \) and \( c'' \) to be in the relation \( \text{super} \) to reflect the resolution. As a result of the rule application we have the register referring to the arguments of the method assigned with the values from the callee register \( r(c \ m \ pc) :: (c \ m \ pc) :: C \ ind + \text{val}_\text{ret} \text{val}_r \). Local registers are assigned with the values \( \theta \), i.e., they can have any value.

The next rule describes method returns:

**Example 4.8.**

\[
\text{return}_r \ c \ m \ pc \ i \\
\land \ r(c \ m \ pc) :: (c' \ m' \ pc') :: C \ i \ \text{val}_r \\
\land \ r(c' \ m' \ pc') :: C \ j \ \text{val}_r' \\
\implies \ r(c' \ m' \ pc' + 1) :: C \ ret \ \text{val}_r \\
\land \ r(c' \ m' \ pc' + 1) :: C \ j \ \text{val}_r'
\]

The statement \( r(c \ m \ pc) :: (c' \ m' \ pc') :: C \ i \ \text{val}_r \) opens the recursive call stack and displays the point in the program from which the current method was called \((c' \ m' \ pc')\) and also it determines the value \( \text{val}_r \) of the register \( i \) that is specified in the instruction as the register with the return value. The expression \( r(c' \ m' \ pc') :: C \ j \ \text{val}_r' \) displays the values of the registers in the callee exactly at the time when the current method was called. As a result we have an expression \( r(c' \ m' \ pc' + 1) :: C \ ret \ \text{val}_r \): the callee’s special register \( \text{ret} \) is assigned a value from the register \( i \) in the current method.

4.5 Modeling Exceptions

This section presents a feature that has not been shown in the concrete semantics, namely, exceptions. We do not describe exception handling in the concrete semantics as it will dramatically complicate the proof of soundness. However, there are no limitations for our analysis to be proven even with its extended version which captures extensions as well.

In order to be able to reason about the exceptions we introduce a new fact. A fact \( e \) consists of the exception name \( e \), and two saved states of the call stack: the first state saves the location where the exception is casted \( C' \) and the second where the handler is situated \( C \):

\[
\begin{align*}
  e & \in \text{string} & \text{Exception name} \\
  e & ::= (C \ C' \ e) & \text{Exception}
\end{align*}
\]

The information stored in the fact \( e \) allows us to uniquely identify each thrown exception.
Abstract Analysis Based on Horn Clauses

In the following we show the example of the function call that we have already discussed but with exceptions.

Example 4.9.

```
INVOKE − REF
invoke-ref c m pc c' m'
∧ par c m pc i ind
∧ r (c m pc) :: C i val_r
∧ par c m pc j 0
∧ r (c m pc) :: C' j (C' c'' ver)
∧ super c'' c'
∧ numloc c' m' val_int
  r c' m' 0 :: (c m pc) :: C ind + val_int val_r
∧ r c' m' 0 :: (c m pc) :: C ind' 0
∧ 0 ≤ ind' ≤ val_int
∧ (try c m pc_start pc_end e pc_catch)
⇒
  pc_start ≤ pc ≤ pc_end
⇒ handler c' m' (c m pc) :: C e (c m pc_catch) :: C'
∧ (handler c m C e' (c'' m'' pc''_catch) :: C'')
∧ ~(e' = e)
⇒ handler c' m' (c m pc) :: C e' (c'' m'' pc''_catch) :: C''
```

The rule stays the same except for one element that we would like to stress:

```
(try c m pc_start pc_end e pc_catch)
∧ pc_start ≤ pc ≤ pc_end
⇒ handler c' m' (c m pc) :: C e (c m pc_catch) :: C'
∧ (handler c m C e' (c'' m'' pc''_catch) :: C'')
∧ ~(e' = e)
⇒ handler c' m' (c m pc) :: C e' (c'' m'' pc''_catch) :: C''
```

Here we introduce two facts. The first one encodes static information about the try-block in try: c m shows in which method of which class try is declared, pc_start indicates the program counter at which try starts to apply, pc_end indicates the last program counter touched by the try declaration, e shows the name of the exception it handles and the last element pc_catch specifies where in the method the program counter jumps to if the exception is raised. In our rule we specify that if the program counter pc is such that pc_start ≤ pc ≤ pc_end (e.g., method call happens inside the try block), we create a dynamic handler handler which is applied to the entire method. The fact
**handler** allows us to transfer handlers among the executed methods, i.e., if the method does not have its own handler we can find a place on the program stack where we will jump in case of an exception. So, **handler** consists of a class name \( \text{clid}' \) and a method name \( m' \) for which it applies. We also the safe program stack when \( m \) gets executed, \((c m pc) :: C\), the name of the exception \( e \), and the place in the call stack where the handler is situated in, \((c'' m'' pc''_{catch}) :: C''\):

\[
\text{handler } c' m' (c m pc) :: C' e' (c'' m'' pc''_{catch}) :: C''
\]

As we have two ways of exception handling (by **try** and by **handler**) we determine when which is applied. Hence, we distinguish two different kinds of **throw**:

**Example 4.10.**

**THROW**

\[
\text{throw } c m pc i
\]

\[
\land r (c m pc) :: C i ((c' m' pc') :: C' e 0)
\]

\[
\land \neg (\text{try } c m pc_{start} pc_{end} e pc_{catch} \lor pc_{start} \leq pc \leq pc_{end})
\]

\[
\land \text{handler } c m C e (c'' m'' pc_{catch}) :: C''
\]

\[
\land r (c m pc) :: C w val_r
\]

\[
\implies r (c'' m'' pc_{catch}) :: C'' w val_r
\]

\[
\land e ((c'' m'' pc_{catch}) :: C'' (c' m' pc') :: C' e)
\]

**THROW − SELF**

\[
\text{throw } c m pc i
\]

\[
\land r (c m pc) :: C i ((c' m' pc') :: C' e 0)
\]

\[
\land \text{try } c m pc_{start} pc_{end} e pc_{catch}
\]

\[
\land pc_{start} \leq pc \leq pc_{end}
\]

\[
\land r (c m pc) :: C w val_r
\]

\[
\implies r (c m pc_{catch}) :: C w val_r
\]

\[
\land e ((c m pc_{catch}) :: C (c' m' pc') :: C' e)
\]

Rule **throw** is applied when we do not have the **try** block inside the method declaration as stated here: \( \neg (\text{try } c m pc_{start} pc_{end} e pc_{catch} \lor pc_{start} \leq pc \leq pc_{end}) \). Then we look for an exception handler in the **handler** predicate supplied to the method. On the contrary in the rule **throw-self**: if we have **try** inside the method we do not check whether the handler is supplied, as the method uses its own handler.

We have specify the rules for self catches in case of check-cast instructions in Table B.12, while those for binary operators in Table B.13 are omitted as they are similar.

The instruction **move-exception** is placed in the beginning of the exception handler. It moves the reference of the recently called extension to the register \( i \) specified in the instruction.
Example 4.11.

\[
\text{MOVE – EXCEPTION} \\
\text{move-exception } c m pc i \\
\land e ((c m pc) :: C (c' m' pc') :: C' e) \\
\land r (c m pc) :: C j ((c' m' pc') :: C' e 0) \\
\implies r (c m pc + 1) :: C i ((c' m' pc') :: C' e 0) \\
\land \not= r (c m pc + 1) :: C
\]

The fact \( e \) stores where the handler is situated: \((c m pc) :: C\). We identify the register which holds the reference to the exception object as follows: we take the exception creation point \((c' m' pc') :: C'\) and look up the register \( j \) that carries the reference to the exception object \( r (c m pc) :: C j ((c' m' pc') :: C' e 0)\). We move the creation point \((c' m' pc') :: C'\) from the fact \( e \) to \( i \), specifying version number \(0\) as we do for all objects (the only difference for exceptions is that this version will never change as we never modify the exception object): \( r (c m pc + 1) :: C i ((c' m' pc') :: C' e 0)\). We also add \( \not= i \) to the rule, specifying that all registers except for \( i \) have the same values as the result of the rule’s application.

4.6 Concurrency and Notation Agreement

In our concrete semantics definition, we explicitly omit two bytecode opcodes \texttt{monitor-enter} and \texttt{monitor-exit}. They capture memory management in the concurrent execution [Java], i.e., how shared variables are accessed. They are generated by the compiler when the instruction \texttt{synchronized} is present in the source code. We do not model the memory in our approach, and in addition we are imprecise but sound on all values for shared variables: the static stack holds an over-approximated set of those, as the information on accesses to the static heap is available only dynamically.

We include only several checks inside the rules as we would like to stress the essence of each instruction application to the program state without overloading it with additional constructions.
Chapter 5

Soundness Results

This chapter presents the soundness results for our analysis technique. Before we proceed, we introduce contractions that we use further in the proof.

Let \([\ ]\) be a function defined as follows:

\[\[P\] := \hat{p}\]

As Definition 4.1 says, \(\hat{p}\) is a set of facts in the analysis, that represent the formalization of the app \(P\) (Definition 3.1). This includes the predicate super which is based on the subtyping rules in Table A.3 from the concrete semantics. In other words, \([\ ]\) gives us the Horn clause representation of the formal app model.

Let \(\Delta_{\text{Android}}\) be the complete set of rules (i.e., definite Horn clauses) of our analysis technique (see Appendix B). In addition, \(\Delta_{\text{Android}}\) contains the Horn clause encoding of standard Android functionality, e.g., a constructor for the creation of the general Object class instance. In other words, \(\Delta_{\text{Android}}\) represents independent from the app part of the analysis technique.

In order to show soundness, first we formally define the relation between the concrete formalization of the state and its representation in the analysis.

**Definition 5.1.** Let \(\mathfrak{R}\) be a relation between the states in the concrete semantics \(\Sigma_P\) and the state in the analysis \(\Delta\), such that if \(\Sigma_P \mathfrak{R} \Delta\) then

1) we extend \(\Delta\), such that it includes both the Horn clause representation of the app \([P]\) and the independent part of the analysis technique \(\Delta_{\text{Android}}\) (the analysis rules and standard functionality):

\[\Delta_{\text{Android}} \cup [P] \subseteq \Delta\]

2) if in the concrete semantics there is a mapping between the call stack \((c.m.pc) :: C\) and the registers \(R\), such that register \(i\) is assigned the value \(val_r\), we can derive this value for the same register and call stack in our analysis:

\[\Sigma_P = \{((c.m.pc) :: C \mapsto R[i \mapsto val_r]), H, S\} \implies \Delta \vdash x (c.m.pc) :: C \ i \ val_r\]
3) if in the concrete semantics a static field $f$ of a class $c$ is assigned a value $val_r$ we can derive this value for the same field and class in our analysis:

$$\Sigma_P = (C, H, S[c.f \mapsto val_r]) \implies \Delta \vdash c \ f \ val_r$$

4) if in the concrete semantics a reference $(\ell, C', c, ver)$ on the heap points to the object instance of the class $c$ in a state $ver$ which is created at $C'$ and placed in the memory location $\ell$, and the field $f$ of this object instance is assigned the value $val_r$, we can derive this value assignment to the same field and instance which has the same reference:

$$\Sigma_P \langle C, H[(\ell, C', c, ver) \mapsto o[f \mapsto val_r], S]\rangle \implies \Delta \vdash h \ (C' \ c \ ver) \ f \ val_r$$

5) if in the concrete semantics a reference $(\ell, C', \tau_a, ver)$ on the heap points to the array instance of the type $\tau_a$ in a state $ver$ which is created at $C'$ and placed in the memory location $\ell$, and the index $i$ of this array instance is assigned the value $val_r$, we can derive this value assignment to the same index and instance which has the same reference:

$$\Sigma_P \langle C, H[(\ell, C', \tau_a, ver) \mapsto \alpha[g \mapsto val_r], S]\rangle \implies \Delta \vdash h \ (C' \ \tau_a \ ver) \ g \ val_r$$

6) if in the concrete semantics a reference $(\ell, C', \tau_a, ver)$ on the heap maps to the array instance $\alpha$, we can derive its size $|\alpha|$ in the analysis:

$$\Sigma_P \langle C, H[(\ell, C', \tau_a, ver) \mapsto \alpha], S]\rangle \implies \Delta \vdash \text{size} \ (C') \ |\alpha|$$

Remark. In the definition of $\mathfrak{X}$ in the properties 4) and 5) we soundly omit the memory location $\ell$ in our analysis as we do not model the memory. In property 6) we address the array instance by its creation point $C'$ as it is impossible to create multiple array instances with a single opcode in the Dalvik bytecode, thus, it uniquely identifies the instance.

We observe that the relation $\mathfrak{X}$ includes call stack, registers, static and dynamic heap, i.e., forms the abstraction of the concrete state in the analysis. The soundness theorem shows that this relation is an invariant for any possible in the concrete semantics execution trace.

**Theorem 5.2.** For all states $\Sigma_P$, $\Sigma'_P$ of a program $P$, there exists an analysis $\Delta$, such that if $\Sigma_P$ is in a relation $\mathfrak{X}$ with $\Delta$ and $\Sigma_P$ reduces possibly in a multiple steps to $\Sigma'_P$, then $\Sigma'_P$ is also in the relation $\mathfrak{X}$ with $\Delta$.

$$\forall \Sigma_P, \Sigma'_P, \exists \Delta : \mathfrak{X} \Delta \wedge \Sigma_P \rightarrow^* \Sigma'_P \implies \Sigma'_P \mathfrak{X} \Delta$$
Proof. The proof proceeds by induction on the execution trace and case distinction for the kind of execution steps.

1. Induction base case:

\[ \Sigma_P = \emptyset : \emptyset \xrightarrow{p} \emptyset \implies \emptyset \xrightarrow{p} \emptyset \Delta \]

\( \emptyset \xrightarrow{p} \emptyset \Delta \) holds, as for all cases 1)-6) in the relation \( \Delta \) premises are \textit{false}, because \( \Sigma_P \) is \( \emptyset \).

\( \emptyset \xrightarrow{p} \emptyset \) holds as there is no step in the concrete semantics to apply therefore we stay at the same state.

2. Our inductive hypothesis:

\[ \forall \Sigma_P, \Sigma'_P, \exists \Delta : \Sigma_P \xrightarrow{p} \Sigma'_P \implies \Sigma'_P \xrightarrow{p} \emptyset \Delta \]

3. Now we show the following:

\[ \forall \Sigma'_P, \Sigma_P, \exists \Delta : \Sigma'_P \xrightarrow{p} \Sigma_P \implies \Sigma'_P \xrightarrow{p} \emptyset \Delta \]

We apply case distinction on different execution steps \( \xrightarrow{p} \). We show that the relation \( \Sigma'_P \xrightarrow{p} \Sigma_P \) holds for each of the components of the relation \( \Delta \) as follows: we specify the untouched by the \( \xrightarrow{p} \) part of the relation (this is always the case for 1), for instance), then we show that the rest also can be derived in \( \Delta \) via specific rule which can be applied as we have its premises because \( \Sigma'_P \xrightarrow{p} \emptyset \Delta \) holds. We use 1) - 6) to address the parts of the relation \( \Delta \) that we prove, i.e., \( \Sigma'_P \xrightarrow{p} \emptyset \Delta \), and we use \( \Delta.1 \) - \( \Delta.6 \) to address parts of the relation \( \Delta \) that we have as a premise, i.e., \( \Sigma'_P \xrightarrow{p} \emptyset \Delta \).

**BINOP (BINOP-2, BINOP-LIT, UNOP, UNOP-TYPE, ...)**

1) trivially holds as not affected by the concrete BINOP rule;

2) holds by the abstract BINOP (BINOP-2, BINOP-LIT, UNOP, UNOP-TYPE, MOVE, CONST, CMP) rule which we can apply as the relation \( \Sigma' \xrightarrow{p} \emptyset \Delta \) holds, more precisely we have facts \( \text{binop} \oplus (\text{binop}-2 \oplus \text{binop-lit} \oplus \text{unop} \oplus \text{unop-type} \oplus \text{move} \oplus \text{const} \oplus \text{cmp} \mid \text{bias}) \) because \( \Delta.1 \) holds and \( r \) because \( \Delta.2 \) holds:

\[
\text{BINOP}
\]

\[ \text{binop}_{\oplus} \in m \quad c \quad m \quad pc \quad i \quad j \quad g \]
\[ \land \quad r \quad (c \quad m \quad pc) \quad C \quad j \quad \text{val}_{\oplus} \]
\[ \land \quad r \quad (c \quad m \quad pc) \quad C \quad g \quad \text{val}'_{\oplus} \]
\[ \implies \quad r \quad (c \quad m \quad pc + 1) \quad C \quad i \quad \text{val}_{\oplus} \oplus \text{val}'_{\oplus} \]
\[ \land \quad r \neq i \quad (c \quad m \quad pc + 1) \quad C \]

3) - 6) same as 1)
### NOOP (GOTO, IF-TRUE, IF-FALSE, IFZ-TRUE, IFZ-FALSE, ...)

The same as for the BINOP case (we consider another facts `noop, goto, if<, ifz<, sswitch-data, sswitch-data-item, etc., we have them as Ξ.1 holds):

\[
\text{NOOP} \quad \text{noop } c m pc \land r (c m pc) :: C i val_r \implies r (c m pc + 1) :: C i val_r
\]

### SGET

1) trivially holds as not affected by the concrete SGET rule;
2) holds by the abstract SGET rule which we can apply as the relation \( \Sigma'' \Delta \), more precisely we have facts `sget\( _r \)` because \( \Xi.1 \) holds, `r` because \( \Xi.2 \) holds, and `s` because \( \Xi.3 \) holds:

\[
\text{SGET} \quad \text{sget}_r c m pc i c' f \land \exists \ c' f \ val_r, \land r (c m pc) :: C i val_r \implies r (c m pc + 1) :: C i val_r
\]

3) - 6) same as 1)

### SPUT

1) trivially holds as not affected by the concrete SPUT rule;
2) same as 1);
3) holds by the abstract SPUT rule which we can apply as the relation \( \Sigma'' \Delta \) holds, more precisely we have facts `sput\( _r \)` because \( \Xi.1 \) holds, `r` because \( \Xi.2 \) holds:

\[
\text{SPUT} \quad \text{sput}_r c m pc i c' f \land r (c m pc) :: C i val_r \implies s c' f val_r
\]

4) - 6) same as 1)

### NEW-INSTANCE

1) trivially holds as not affected by the concrete NEW-INSTANCE rule;
2) holds by the abstract NEW-INSTANCE rule which we can apply as the relation \( \Sigma'' \Delta \) holds, more precisely we have facts `new-instance, field` because \( \Xi.1 \) holds, `r` because \( \Xi.2 \) holds:

\[
\text{NEW-INSTANCE} \quad \text{new-instance } c m pc i c' \land \text{field } c' f \land r (c m pc) :: C i val_r \implies r (c m pc + 1) :: C i \((c m pc) :: C c' 0) \land h ((c m pc) :: C c' 0) f 0 \land \neg \exists (c m pc + 1) :: C
\]
3) 5) - 6) same as 1);
4) same as 2)

**INSTANCE-OF**

1) trivially holds as not affected by the concrete NEW-INSTANCE rule;
2) holds by the abstract INSTANCE-OF rule which we can apply as $\Sigma'' R \Delta$ holds, more precisely we have facts \texttt{instance-of} and \texttt{super} because $\mathbf{R}.1$ holds, $r$ because $\mathbf{R}.2$ holds:

```latex
$$
\begin{align*}
\text{INSTANCE-OF} & \quad \text{instance-of } c \ m \ pc \ i \ j \ c' \\
& \land r (c \ m \ pc) :: C j (C' c'' v e) \\
& \Rightarrow r (c \ m \ pc + 1) :: C i \ super \ c'' c' \\
& \land \ \mathbf{r} \neq l (c \ m \ pc + 1) : C
\end{align*}
$$
```

3) - 6) same as 1)

**CHECK-CAST**

1) trivially holds as not affected by the concrete CHECK-CAST rule;
2) $\mathbf{R}.2$ holds by the abstract CHECK-CAST rule which we can apply as $\Sigma'' R \Delta$ holds, more precisely we have facts \texttt{check-cast} and \texttt{super} because $\mathbf{R}.1$ holds, $r$ because $\mathbf{R}.2$ holds:

```latex
$$
\begin{align*}
\text{CHECK-CAST} & \quad \text{check-cast } c \ m \ pc \ i \ c' \\
& \land r (c \ m \ pc) :: C i (C' c'' \ v e) \\
& \land super \ c'' c' \\
& \Rightarrow r (c \ m \ pc + 1) :: C w \ v a l_e \\
& \land \ \mathbf{r} \neq f (c \ m \ pc + 1) : C
\end{align*}
$$
```

3) - 6) same as 1)

**IPUT**

1) trivially holds as not affected by the concrete IPUT rule;
2) $\mathbf{R}.2$ and $\mathbf{R}.4$ hold by the abstract IPUT rule which we can apply as $\Sigma'' R \Delta$ holds, more precisely we have facts \texttt{iput}, because $\mathbf{R}.1$ holds, $r$ because $\mathbf{R}.2$ holds:

```latex
$$
\begin{align*}
\text{IPUT} & \quad \text{iput}_c \ c \ m \ pc \ i \ j \ c' \ f \\
& \land r (c \ m \ pc) :: C j (C' c'' \ v e) \\
& \land r (c \ m \ pc) :: C i \ v a l_e \\
& \Rightarrow h (C' c'' \ v e + 1) f \ v a l_e \\
& \land r (c \ m \ pc + 1) :: C j (C' c'' \ v e + 1) \\
& \land \mathbf{f} \neq l (c \ m \ pc + 1) : C \\
& \land \mathbf{f} \neq f (C' c'' \ v e + 1)
\end{align*}
$$
```

3) 5) - 6) same as 1);
4) same as 2)
IGET

1) trivially holds as not affected by the concrete IGET rule;
2) holds by the abstract IGET rule which we can apply as $\Sigma'' \mathcal{R} \Delta$ holds, more precisely we have facts $\text{iget}_\tau$ because $\mathcal{R}.1$) holds, $\text{h}$ because $\mathcal{R}.4$) holds, $\text{r}$ because $\mathcal{R}.2$) holds:

\[
\text{IGET} \\
\text{iget}_\tau \ c \ m \ pc \ i \ j \ c' \ f \\
\land \ \text{r} (c \ m \ pc) :: C \ j \ (C' \ c' \ \text{ver}) \\
\land \ \text{h} (C' \ c' \ \text{ver}) \ f \ \text{val}_r \\
\implies \ \text{r} (c \ m \ pc + 1) :: C \ i \ \text{val}_r \\
\land \ \hat{\text{r}} \neq i (c \ m \ pc + 1 :: C)
\]

3) - 6) same as 1)

CONST-STRING (CONST-CLASS)

The same as for IPUT (we consider other facts const-string and const-class, we have them as $\mathcal{R}.1$) holds):

\[
\text{CONST} - \text{CLASS} \\
\text{const-class} \ c \ m \ pc \ i \ \text{str} \\
\land \ \text{r} (c \ m \ pc) :: C \ i \ \text{val}_r \\
\implies \ \text{h} ((c \ m \ pc) :: C \ \text{String} \ 0) \ \text{value} \ \text{str} \\
\land \ \text{r} ((c \ m \ pc) :: C \ \text{Object} \ 0) \ \text{name} ((c \ m \ pc) :: C \ \text{String} \ 0) \\
\land \ \text{h} ((c \ m \ pc) :: C \ i ((c \ m \ pc) :: C \ \text{Object} \ 0) \\
\land \ \hat{\text{r}} \neq i (c \ m \ pc + i :: C)
\]

NEW-ARRAY

1) trivially holds as not affected by the concrete NEW-ARRAY rule;
2) holds by the abstract NEW-ARRAY rule which we can apply as $\Sigma'' \mathcal{R} \Delta$ holds, more precisely we have facts new-array because $\mathcal{R}.1$) holds, $\text{r}$ because $\mathcal{R}.2$) holds:

\[
\text{NEW} - \text{ARRAY} \\
\text{new-array} \ c \ m \ pc \ i \ j \ \tau_a \\
\land \ \text{r} (c \ m \ pc) :: C \ j \ \text{val}_{\text{int}} \\
\implies \ \text{r} (c \ m \ pc + 1) :: C \ i ((c \ m \ pc) :: C \ \tau_a \ 0) \\
\land \ \text{size} ((c \ m \ pc) :: C) \ \text{val}_{\text{int}} \\
\land \ \text{h} ((c \ m \ pc) :: C \ \tau_a \ 0) \ \text{ind} \ 0 \\
\land \ \text{h} \ ((c \ m \ pc) :: C) \ \text{ind} \ 0 \\
\land \ 0 \leq \text{ind} \leq \text{val}_{\text{int}} - 1 \\
\land \ \hat{\text{r}} \neq i (c \ m \ pc + i :: C)
\]

3) same as 1);
4) same as 1);
5) same as 2);
6) same as 2)
ARRAY-LENGTH

1) trivially holds as not affected by the concrete ARRAY-LENGTH rule;
2) holds by the abstract ARRAY-LENGTH rule which we can apply as $\Sigma'' \mathbf{R} \Delta$
holds, more precisely we have facts $\text{array-length}$ because $\mathbf{R}.1$ holds, $r$
because $\mathbf{R}.2$ holds, $\text{size}$ because $\mathbf{R}.6$ holds:

$$\text{ARRAY-LENGTH}
\begin{array}{c}
\text{array-length} \ e \ m \ pc \ i \ j \\
\land r (e \ m \ pc) :: C \ j (C' \ \tau_a \ \text{ver}) \\
\land \text{size} (C') \ val_{\text{int}}
\end{array}
\implies
r (e \ m \ pc + 1) :: C \ i \ \text{val}_{\text{int}}
\land \exists \not= (e \ m \ pc + 1) :: C$$

3) - 6) same as 1)

CHECK-CAST-ARRAY

The same as for CHECK-CAST (we consider another fact $\text{check-cast-array}$, we have them as $\mathbf{R}.1$ holds):

$$\text{CHECK-CAST-ARRAY}
\begin{array}{c}
\text{check-cast} \ c \ m \ pc \ i \ \tau_a' \\
\land r (e \ m \ pc) :: C \ i (C' \ \tau_a \ \text{ver}) \\
\land \text{super} \ \tau_a \ \tau_a' \\
\land r (e \ m \ pc) :: C \ w \ \text{val}_r
\end{array}
\implies
r (e \ m \ pc + 1) :: C \ w \ \text{val}_r$$

APUT

The same as for IPUT (we consider another fact $\text{aput}_r$, we have it as $\mathbf{R}.1$ holds; 4) is preserved, 5) is proved similar to 2) in IPUT):

$$\text{APUT}
\begin{array}{c}
\text{aput}_r \ c \ m \ pc \ i \ j \ g \\
\land r (e \ m \ pc) :: C \ j (C' \ \tau_a \ \text{ver}) \\
\land r (e \ m \ pc) :: C \ w \ \text{val}_r \\
\land r (e \ m \ pc) :: C \ g \ \text{val}_b' \\
\text{super} \ \tau \ \tau_a
\end{array}
\implies
h (C' \ \tau_a \ \text{ver} + 1) \ \text{val}_b' \ \text{val}_r
\land r (e \ m \ pc + 1) :: C \ j (C' \ \tau_a \ \text{ver} + 1)
\land \exists \not= (e \ m \ pc + 1) :: C \\
\land \exists \not= \text{val}_b' (C' \ \tau_a \ \text{ver} + 1)$$

AGET

The same as for IGET (we consider another fact $\text{aget}_r$, we have it as $\mathbf{R}.1$ holds; 4) is preserved, 5) is proved similar to 2) in IGET):

$$\text{AGET}
\begin{array}{c}
\text{aget}_r \ c \ m \ pc \ i \ j \ g \\
\land r (e \ m \ pc) :: C \ j (C' \ \tau_a \ \text{ver}) \\
\land r (e \ m \ pc) :: C \ g \ \text{val}_b' \\
\land h (C' \ \tau_a \ \text{ver}) \ \text{val}_b' \ \text{val}_r
\end{array}
\implies
r (e \ m \ pc) :: C + 1 \ i \ \text{val}_r
\land \exists \not= (e \ m \ pc + 1) :: C$$

INSTANCE-OF-ARRAY

The same as for INSTANCE-OF (we consider another fact $\text{instance-of-array}$, we have it as $\mathbf{R}.1$ holds:
INSTANCE – OF – ARRAY
\[
\text{instance-of-array } c \ m \ pc \ i \ j \ \tau_a \\
\land r (c m pc) :: C j (C' \tau_a' \text{ver}) \implies r (c m pc + 1) :: C i \text{super} \tau_a' \tau_a \\
\land \hat{r} (c m pc + 1) :: C
\]

FILL-ARRAY-DATA

1) trivially holds as not affected by the concrete FILL-ARRAY-DATA rule;
2) holds by the abstract FILL-ARRAY-DATA rule which we can apply as \(\Sigma'' \Delta\) holds, more precisely we have facts fill-array-data, data, data-item because \(\Delta 1\) holds, \(\Delta 2\) because to \(\Delta 2\) holds, size because \(\Delta 6\) holds:

FILL – ARRAY – DATA
\[
\text{fill-array-data } c \ m \ pc \ i \ pc' \\
\land r (c m pc) :: C i (C' \tau_a' \text{ver}) \\
\land \text{data } c \ m \ pc' \text{val}_\text{int} \\
\land \text{data-item } c \ m \ pc' \text{ind} \text{val}_b \\
\land \text{size } (C') \text{val}'_\text{int} \\
\land \text{val}_\text{int} \leq \text{val}'_\text{int} \\
\land \text{super } b \tau_a \\
\implies r (c m pc + 1) :: C i (C' \tau_a \text{va} + 1) \\
\land \text{h} (C' \tau_a \text{va} + 1) \text{ind} \text{val}_b \\
\land \hat{\text{h}} (c m pc + 1) :: C \\
\land \hat{\text{h}} \text{ind } (C' \tau_a \text{ver} + 1)
\]

3) same as 1);
4) same as 1);
5) same as 2);
6) same as 1)

FILLED-NEW-ARRAY

1) trivially holds as not affected by the concrete FILLED-NEW-ARRAY rule;
2) holds by the abstract FILLED-NEW-ARRAY rule which we can apply as \(\Sigma'' \Delta\) holds, more precisely we have facts filled-new-array, par, move-result because \(\Delta 1\) holds, \(\Delta 2\) because \(\Delta 2\) holds:

FILLED – NEW – ARRAY
\[
\text{filled-new-array } c \ m \ pc \ \tau_a \\
\land \text{num}_\text{par } c \ m \ pc \text{val}_\text{int} \\
\land \text{par } c \ m \ pc \ i \text{ind} \\
\land r (c m pc) :: C i \text{val}_b \\
\land \text{super } b \tau_a \\
\implies \text{size } ((c m pc) :: C) \text{val}_\text{int} \\
\land \text{h } ((c m pc) :: C \tau_a \text{va} \text{ind} \text{val}_b \\
\land \text{a } C \text{ret } ((c m pc) :: C \tau_a \text{va} \\
\land \hat{\text{r}} \text{ret } (c m pc + 1) :: C
\]

3) same as 1);
4) same as 1);
5) same as 2);
6) same as 2)

**INVOKE (INVoke-REF)**

1) trivially holds as not affected by the concrete INVOKE (INVoke-REF) rule;
2) hold by the abstract INVOKE (INVoke-REF) rule which we can apply as $\Sigma'' \Delta$ holds, more precisely we have facts $\texttt{invoke}$, $\texttt{numloc}$, $\texttt{par}$ because $\mathfrak{X}.1$ holds, $r$ because $\mathfrak{X}.2$ holds:

\[
\text{INVOKE - REF}
\]

\[
\begin{align*}
\text{invoke-ref } & c \ m \ pc \ c' \ m' \\
& \land \texttt{par } c \ m \ pc \ i \ ind \\
& \land \texttt{r } (c \ m \ pc) :: C i \ val_r \\
& \land \texttt{par } c \ m \ pc \ j \ 0 \\
& \land \texttt{r } (c \ m \ pc) :: C j (C' c'' \ ver) \\
& \land \texttt{super } c' \ c'' \\
& \land \texttt{numloc } c' \ m' \ \text{val}_\text{last}
\end{align*}
\]

\[
\Rightarrow \texttt{r } (c' \ m' \ 0) :: (c \ m \ pc) :: C \ \text{ind} + \text{val}_\text{last} \ \text{val}_r
\]

3) same as 1);
4) same as 1);
5) same as 1);
6) same as 1)

**RETURN-VOID**

1) trivially holds as not affected by the concrete RETURN-VOID rule;
2) holds by the abstract RETURN-VOID rule which we can apply as $\Sigma'' \Delta$ holds, more precisely we have fact $\texttt{return-void}$ because $\mathfrak{X}.1$ holds, $r$ because $\mathfrak{X}.2$ holds:

\[
\text{RETURN - VOID}
\]

\[
\begin{align*}
\text{return-void } & c \ m \ pc \\
& \land \texttt{r } (c \ m \ pc) :: (c' \ m' \ pc') :: C i \ val_r \\
\end{align*}
\]

\[
\Rightarrow \texttt{r } (c' \ m' \ pc' + 1) :: C i \ val_r
\]

3) - 6) same as 1)

**RETURN**

1) trivially holds as not affected by the concrete RETURN rule;
2) holds by the abstract RETURN rule which we can apply as $\Sigma'' \Delta$ holds, more precisely we have fact $\texttt{return}$ because $\mathfrak{X}.1$ holds, $r$ because $\mathfrak{X}.2$ holds:
3) - 6) same as 1)

**MOVE-RESULT**

1) trivially holds as not affected by the concrete MOVE-RESULT rule;
2) holds by the abstract MOVE-RESULT rule which we can apply as \( \Sigma'' \mathbf{X} \Delta \) holds, more precisely we have fact \( \text{move-result}_r \) because \( \mathbf{X}.1 \) holds, \( r \) because \( \mathbf{X}.2 \) holds:

\[
\text{MOVE-RESULT}
\]

\[
\begin{align*}
\text{move-result}_r c m pc i \\
\land r (c m pc) :: C \text{ ret val}_r \\
\implies r (c' m' pc' + 1) :: C \text{ ret val}_r \\
\land \bar{\bar{r}} (c' m' pc' + 1) :: C
\end{align*}
\]

3) - 6) same as 1)

This concludes the proof.
Chapter 6

Implementation

In this Chapter we introduce how we implement our analysis technique in a prototype. The prototype consists of two main blocks: the pre-analysis and the analyzer itself.

6.1 Pre-analysis: Intermediate Representation

The input of the pre-analysis block is given in the .apk format which contains the .dex file with the DVM bytecode already mentioned in Section 2.1. This format is a general one for all applications on the Google Play store [Gooa]. We decode instructions from the initial format to .smali using apktool [Anda]. It creates a representation of the bytecode which is suitable for the analysis, where every class is represented in a separate file and all generated instructions correspond to the standard declared at [Dala]. Our pre-analysis unit translates the input to the internal representation which is used further by the analyzer. The pre-analysis starts with Algorithm D.1. It receives a folder specified in path. The folder contains .smali files with Dalvik bytecode instructions. Algorithm D.1 extracts the data about the app from files to the object arrays of the following classes in Python: Class, Field, Method, Instruction, and ArrData. These classes form the intermediate representation that mitigates the complexity of the algorithms and we also use them to provide human-readable information about the results of the analysis. In other words, we use numbers instead of strings for classes, methods and other program structures in order to lower the memory consumption, which results in potential support of more complicated types of analysis. The results of the analysis can be reinterpreted back to the strings providing human readability.

6.2 Theorem Prover Z3

There is a need to mention that before we finally choose a theorem prover as the basis for our prototype implementation, we performed several experiments with other tools, e.g., Boogie [Boo]. It is able to prove different properties of programming code, however, the expressiveness is quite limited. In fact, this tool is a higher order tool which uses
a theorem prover to perform verification tasks. With Boogie, our experiment was to see whether we can generate annotations (Boogie needs annotations in order to perform verification tasks; here we have also tried to use ghost variables [DGJN11] in order to encode the state of the execution) automatically and as well to see how many of them are needed. However, we realized that it lacks the expressiveness of low-level encoding that is usually allowed in theorem provers and also it cannot be fully automated.

As a consequence, our analyzer creates an input in so-called SMT 2.0 format [BST+10] which is accepted by a number of theorem provers (examples can be found at [SMT]). This flexibility is a strong side of our analysis technique since we can exploit a particular theorem prover to perform a task according to its performance.

The theorem prover of our choice is Z3 [DMB08]. This prover is used in a variety of successful verification tools, even in the most recent ones that allow for working with JavaScript [FSC+13, BFG+14]. In addition, it has two important features that we make use of: a fixed-point engine and widening operator support.

### 6.2.1 Z3 Syntax Description

The theorem prover Z3 receives the input files in so-called SMT 2.0 notation. In order to simplify the description we use the simplified version of the actual encoding.

Let us consider the following toy example in the SMT 2.0 notation:

**Example 6.1.**

```plaintext
( declare - var i Int )
( declare - var j Int )
( declare - var pc Int )
( declare - var vali Int )
( declare - var valj Int )
( declare - rel add-int2(Int Int Int))
( declare - rel reg-int(Int Int Int))
```

These are so-called declarations. The first five ones define variables, for instance, i, so the analysis reasons about i as an arbitrary integer that can be replaced by any integer number. The two last ones are the predicate `reg-int` for encoding the register value and the predicate `add-int2` for encoding the sum of two values from the registers. The next example clarifies their usage:

**Example 6.2.**

```plaintext
(rule (add-int2 (1 0 1)))
(rule (=> (and (add-int2 pc i j) (reg-int pc i vali) (reg-int pc j valj))
         (reg-int pc+1 i vali+valj)))
```

Here we consider a toy app that contains a single method. Therefore, the program counter can be represented as a single integer pc. The first string from Example 6.2 starts with the word rule, however, as it has no premises it encodes a Horn clause of the type fact. More precisely, we state the logical fact that at program counter 1 in the analyzed toy app there is an instruction `add-int2`. Both in the concrete semantics and the analysis we use the following shortcut for binary operations of such kind: $\text{binop-2}_b$. Here we
replace $\oplus$ by add and $b$ by int. So, this is how all facts about the program are represented in the SMT 2.0. The second string declares the simplified version of the binop-2$\oplus$ rule of the analysis described in Table B.3. The statement add-int2 pc i j gives the information that in some app there is an instruction add-int2 at some pc. The two elements reg-int pc i vali and reg-int pc j valj show that it is true that we have two register with integer values (as the binary operation in Dalvik bytecode takes as input only integers as specified in the analysis and the concrete semantics). Then, as the consequence of all facts mentioned above, we can derive the fact reg-int pc+1 i vali+valj, so register i holds as a value the result of the sum operator over the values given in the instruction registers. In case of the given fact add-int2(1 0 1), these values are from the registers 0 and 1.

6.2.2 Termination
As we have already mentioned in Section 2.4, termination is the desired property for any abstract interpretation analysis. The Z3 engine offers support for fixed-points. However, they are no used by default: we have to explicitly specify several options in order to use the fixed-point engine (we discover this in a series of experiments):

- force the compiler to use widening on recursive predicates (e.g., infer the termination property of the guard from the cycle structure);
- the Z3 engine must explicitly use an unbound compressor for avoiding free variables in rule heads;
- we have to specify the predicate representation for the if and ifz predicates allowing them to use the reduced product of two representations - intervals and bounds, otherwise, Z3 returns nothing, not even unknown as it cannot distinguish these two special predicates for cycle termination from the others.

We have to do this option instrumentation only once and it does not affect the automation of our analysis.

We have performed a number of experiments. They show that we can tackle even extremely complicated cases of cycles. For instance, we can effectively model a cycle with an unknown initial value of the guard (as it comes from the user) while being modified inside another function. We were able to show the property the guard must satisfy in order to leave the cycle (or never enter it).

We remark that Z3 does not prove the inductive cases, only its fixed-point engine provides this functionality.

6.3 Analysis
The analysis processes blocks entailed from the Horn clauses’ types. We create an SMT 2.0 notation version for the facts about the app, rules about the app execution
derived from the analysis description, and queries that reflect our questions to the theorem prover. We show the principle of SMT 2.0 encoding in Subsection 6.2.1. In the following, we point out the general similarities and differences between the result and the analysis specification that demonstrates the fact that the soundness is preserved:

- The mapping between the pair of types \( \text{int}, \text{long} \) in the analysis is mapped to the type \( \text{Int} \) in Z3 although it is supertype of both; in the same manner \( \text{float} \) and \( \text{double} \) are mapped to \( \text{Real} \).

- To model the stack from the analysis description we use a type \( \text{List} \) that is able to express recursive structures.

- A rule for type \( \tau \) is decomposed into several rules for each of the possible types.

- The opcodes for Dalvik have no specific type (\( \text{int/long, float/double} \)) in some operations (e.g., with arrays \( \text{aget, aput} \)), their type should be inferred from their usage. For type inference we can adopt the algorithm from [EOMC11] that uses the Hindley-Millner algorithm [Mil78] which is proved to be complete: we infer the type for the operation with unknown types from the follow-up operations with known types. This is done locally for each method (as every method has its own set of local registers; we always know the type of the supplied parameters from the method’s signature).

- There is no function in Z3 for the logical encoding of the function \( \text{cmp} \) that performs floating point or long comparison.

\[
\text{cmp}_{\tau, \cdot} | \text{bias}(\text{val}_\tau, \text{val}'_\tau) = \begin{cases} 
0 & : \text{val}_\tau = \text{val}'_\tau \\
1 & : \text{val}_\tau < | \text{bias} \text{val}'_\tau \\
-1 & : \text{val}_\tau > | \text{bias} \text{val}'_\tau 
\end{cases}
\]

\[
\text{bias} ::= \geq, \leq
\]

Determines how to treat \( \text{NaN} \) values

The theorem prover Z3 has a support for Not-a-Number (\( \text{NaN} \)) values, however, we have not found a way to encode \( \text{bias} \) that manages the \( \text{NaN} \) values treatment.
Therefore, we consider the over-approximation for this opcode, i.e., our analysis technique results in the set of all possible outcomes, namely 0, 1 and -1.

The algorithms for the analysis part are depicted in Appendix D.

**Entry points.** The entire execution of the program contains a series of method calls. The execution starts from the entry point - the first method which is usually called from the outside the app, i.e., by the operating system. In Java such a method is called `main`. Therefore, we have a deterministic entry point. In Android apps there exist a number of entry points that can be potentially called. The existence of multiple entry points is reflected in our analysis by adding the calls to all possible kinds of entry points from an unspecified program counter in the artificial method of the `android` class. These entry points are:

1. constructors for sub-classes of activities, services, broadcast receivers, and content providers;
2. `onSomeEvent` methods in the Android API.

The created method can be seen as an artificial `main` method.

**Auxiliary code.** Some code is not included in the app itself like standard libraries for Java and Android. But their functionality are known, for instance, the source code for Android is available online [Ande], hence, we use it in order to model the platform in Z3 as a library for the execution. In other words, we pre-compile standard functionality in the set of facts before the analysis starts, e.g., the constructor for class `Object` that is called for instance by every class, as all classes inherit from it.
Chapter 7

Performance Evaluation

We conduct a performance evaluation for a case study that introduces an activity on the phone which must have certain security properties. More specifically, we implemented a voting client that allows user to cast a vote on the Android device. We choose this special case as it reflects the property of the intentional leakage that we have discussed before: the user enters the vote in. The app and eventually the user’s vote is leaked to the Internet in order to be counted. However, if the vote was properly secured there is no problem. In this section, we detail this example, especially showing what it means for the vote to be secured properly.

7.1 System Setup and Experiments

We develop our voting client app as a part of the ad-hoc electronic voting system called Civitas [CCM08]. Civitas is the first remote voting system which is proved to be coercion-resistant. The entire system (and the original client for it) is implemented to run only on PC as it can be compiled only by a modified Java compiler called Jif [jif]. Hence, we have implemented our own voting client for Android.

We strictly follow the Civitas specification for the voting client. This specification is a formal description of the functionality that the voting client has to achieve the secrecy of the vote and other strong properties guaranteed by the voting system. As a part of the voting client we implement ElGamal [EG85] encryption and two Zero-Knowledge [GMW86] protocols resulting in $\approx 100$ classes.

We achieve full compatibility with the complex Civitas system. In order to show that our voting client is well integrated with all components of the voting system we have performed a number of experiments that are performed as follows with minor differences:

1. we create a Perl [Per] script that runs the server side of Civitas on the PC;
2. we simultaneously execute our voting client on the Android device and cast a vote;
3. the vote is delivered via a WiFi network to the server;
4. the script continues on the server side to finish the voting procedure;
5. Civitas’ server side performs a number of checks to have the guarantee that the vote has a certain form and is encrypted with a certain key;

6. the experiment finishes storing the result in the server’s file system.

As all the experiments were successful we can state that we have implemented all needed functionality for a voter client for Civitas which runs fast (≈1 second from the vote cast to the delivery) on the Android device. We state this as a separate contribution of this thesis.

7.2 Tested Property and Results

As we define the subject for our evaluation, we also have to fix the property that we want to verify. Since our analysis is general, it can be used in order to show different properties of the analyzed app. For our voting client we have chosen the following property: "in any execution trace, if the output of the vote happens, the encryption function was applied to the output before". We do not tackle verification of the properties for the encryption function itself: similar functionality has proven to work in the original client, so we consider it as trusted.

The preliminary results for a not fully-fletched implementation of the analysis technique shows that it is possible to verify the tested property. The running time on the machine with an Intel Core 2 Duo processor is 7 seconds.
Chapter 8

Conclusion

We have presented a novel static analysis technique for Android apps. It is based on the concept of abstract interpretation and uses Horn Clauses as the base primitives. The presented analysis technique has a number of strong properties. Our analysis is precise on the values of local registers operated by each method. We introduce a state for the instances accessed on the heap. Therefore, our analysis technique is capable of determining which values the fields of the object have during the execution. The presented analysis technique is context-sensitive, i.e., we precisely model the call chain and can determine the exact point of the return after the function invocation. All these analysis features allow us to perform analysis of the complicated properties of the app, e.g., the intentional leakage, as we are able to model not only the call chain but we also model the other components of the execution state with precision.

The presented approach is built upon the full formalization of the Dalvik bytecode. This formalization is detailed and of independent interest for different areas of research, for instance, software verification.

We provide soundness results for our analysis technique. We determine the abstraction relation between the concrete semantics and our analysis technique. Our proof shows that this relation is invariant: during the state evolution in the concrete semantics, we are able to show the same state properties in our analysis technique for any kind of transition step between states in the concrete semantics.

The applicability of the approach is demonstrated by the prototype which implements the encoding of the analysis technique into the general format supported by the number of modern theorem provers. We use the modern features of the reasonably chosen theorem prover to achieve high precision and termination results even for the cases with infinite loops.

We apply our analysis technique implementation to a specially designed case study: the Android app which acts as a remote voting client for a state-of-the-art remote voting system. This case study allows us to study the power of our analysis technique for performing verification of security-relevant component usage.
Future Work. We want to extend our analysis technique in two ways.

The first extension is that we want to instrument the Inter-Component Communication and privilege escalation detection in our analysis technique. We must instrument the part of the Android functionality that manages the communication between components as a library encoded in a set of facts supplied to the theorem prover. Afterwards we consider all the apps installed on the Android device as a potential attack surface for the privilege escalation, so we do not require attacker typability as we know the attacker. We assign each app the level according to the permissions that we extract from the app’s manifest file [Andb]. Then we have to analyze all inter-component calls that are possible on the device, given the set of installed apps. We plan to reason about the problem of Inter-Component Communication as of the Interprocedural Distributive Environment (IDE) [SRH96] problem, incorporating this idea from the recent paper [OMJ+13], but we work with the bytecode of the apps installed on the device. As the result of the analysis we will have the complete graph of all possible communications and, hence, statically detect a privilege escalation if there is a path from a lower level component to a higher level component.

The second extension to the analysis is that we want to make our analysis able to reason about weaker non-interference [LMP12], where the non-interference result holds depending on what is declassified and where declassification happens. This task is close in spirit to the studied problem of the intentional/safe leakage, where we can allow it when the securing operations preceding it are proved to be performed. As we base our analysis on the general construct of Horn clauses we foresee that we have enough expressiveness to encode non-interference properties in our analysis technique.
Bibliography


## Appendix A

### Concrete Semantics

<table>
<thead>
<tr>
<th>Table A.1 Simplified Dalvik bytecode syntax: general</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c, f, m, str \in \text{string}$</td>
</tr>
<tr>
<td>$i, j, g, z, w, pc \in \mathbb{N}$</td>
</tr>
<tr>
<td>$b ::= \text{byte}</td>
</tr>
<tr>
<td>$\tau ::= b</td>
</tr>
<tr>
<td>$\tau_a ::= \text{array of } \tau$</td>
</tr>
<tr>
<td>$\text{ref} ::= \tau</td>
</tr>
<tr>
<td>$\text{P} ::= (\text{c.m.ppc}</td>
</tr>
<tr>
<td>$\tau_c ::= (c_{\text{sup}}, c_{\text{imp}}, f, m)$</td>
</tr>
<tr>
<td>$\text{field} ::= \tau$</td>
</tr>
<tr>
<td>$\text{method} ::= (\text{numloc}, \tau_{\text{arg}}, \tau_{\text{ret}}</td>
</tr>
<tr>
<td>$\text{numloc} \in \mathbb{N}$</td>
</tr>
<tr>
<td>$\text{inst} ::= \text{opcode}</td>
</tr>
<tr>
<td>$\text{name} ::= \text{string}$</td>
</tr>
<tr>
<td>$\text{arrayData} ::= \text{val}_b$</td>
</tr>
<tr>
<td>$\text{packedSwitch} ::= (f_{\text{st}}, p_{\text{st}})$</td>
</tr>
<tr>
<td>$\text{sparseSwitch} ::= \text{key} \mapsto \text{pc}$</td>
</tr>
<tr>
<td>$\text{key}(f_{\text{st}}) \in \mathbb{N}$</td>
</tr>
<tr>
<td>$p_{\text{st}} ::= \text{pc}$</td>
</tr>
</tbody>
</table>
### Table A.2 Concrete semantic domains

<table>
<thead>
<tr>
<th>Domain</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ret$</td>
<td>$L ::= c.m.pc$</td>
<td>Special register for the return</td>
</tr>
<tr>
<td>$L$</td>
<td>$C :: L :: C</td>
<td>L$</td>
</tr>
<tr>
<td>$C$</td>
<td>$R ::= (i</td>
<td>ret) \mapsto val_r$</td>
</tr>
<tr>
<td>$R$</td>
<td>$o ::= f \mapsto val_r$</td>
<td>Register</td>
</tr>
<tr>
<td>$o$</td>
<td>$\alpha ::= val_r$</td>
<td>Objects</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$H ::= (\ell, C, c</td>
<td>\tau_a, \text{ver}) \mapsto (o</td>
</tr>
<tr>
<td>$H$</td>
<td>$S ::= c.f \mapsto val_r$</td>
<td>Heap</td>
</tr>
<tr>
<td>$S$</td>
<td>$\Sigma_P ::= \langle (C \mapsto R), H, S \rangle$</td>
<td>Static heap</td>
</tr>
<tr>
<td>$\Sigma_P$</td>
<td></td>
<td>Program state</td>
</tr>
</tbody>
</table>

### Table A.3 Subtyping

<table>
<thead>
<tr>
<th>Subtyping</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \leq_P c$</td>
<td>$b \leq_P b$</td>
</tr>
<tr>
<td>SUB-REFL</td>
<td>SUB-REFL-STANDARD</td>
</tr>
<tr>
<td>$c \leq_P c'$</td>
<td>$c' \leq_P c''$</td>
</tr>
<tr>
<td>SUB-TRANS</td>
<td>SUB-SUPER</td>
</tr>
<tr>
<td>$c \leq_P c'$</td>
<td>$c' \leq_P c''$</td>
</tr>
<tr>
<td>$c \leq_P c''$</td>
<td>$c \leq_P c_{sup}$</td>
</tr>
<tr>
<td>$c \leq_P c_{sup}$</td>
<td></td>
</tr>
<tr>
<td>$c \leq_P c_{sup}$</td>
<td></td>
</tr>
</tbody>
</table>

### Table A.4 Execution

<table>
<thead>
<tr>
<th>Execution</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC-STEP</td>
<td>EXEC-EDGE</td>
</tr>
<tr>
<td>$\Sigma_P \rightarrow_P \Sigma_P'$</td>
<td>$\Sigma_P \rightarrow^*_{p} \Sigma_P'$</td>
</tr>
<tr>
<td>$\Sigma_P \rightarrow_{p} \Sigma_P'$</td>
<td>$\Sigma_P \rightarrow^*_{p} \Sigma_P'$</td>
</tr>
<tr>
<td>$\Sigma_P \rightarrow_{p} \Sigma_P'$</td>
<td>$\Sigma_P \rightarrow^*_{p} \Sigma_P'$</td>
</tr>
<tr>
<td>Operation</td>
<td>Rule Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| **BINOP** / \( P(c.m.pc) = \text{BINOP}_b \odot i \ j \ g \) | \[
R[j] = \text{val}_b \\
R[g] = \text{val}'_b \\
R' = R[i \mapsto \text{val}_b \oplus \text{val}'_b]
\]
\[
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_p \langle ((c.m.pc + 1) :: C \mapsto R'), H, S \rangle
\] |
| **BINOP-2** / \( P(c.m.pc) = \text{BINOP-2}_b \odot i \ j \) | \[
R[i] = \text{val}_b \\
R[j] = \text{val}'_b \\
R' = R[i \mapsto \text{val}_b \oplus \text{val}'_b]
\]
\[
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_p \langle ((c.m.pc + 1) :: C \mapsto R'), H, S \rangle
\] |
| **BINOP-LIT** / \( P(c.m.pc) = \text{BINOP-LIT}_b \odot i \ j \ \text{val}_b \) | \[
R[j] = \text{val}_b \\
R' = R[i \mapsto \text{val}_b \oplus \text{val}'_b]
\]
\[
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_p \langle ((c.m.pc') :: C \mapsto R'), H, S \rangle
\] |
| **UNOP** / \( P(c.m.pc) = \text{UNOP}_b \odot i \ j \) | \[
R[j] = \text{val}_b \\
R' = R[i \mapsto \odot \text{val}_b]
\]
\[
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_p \langle ((c.m.pc + 1) :: C \mapsto R'), H, S \rangle
\] |
| **UNOP-TYPE** / \( P(c.m.pc) = \text{UNOP}_b \ b' \odot i \ j \) | \[
R[j] = \text{val}_b \\
R' = R[i \mapsto (\text{val}_b)_{b'}]
\]
\[
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_p \langle ((c.m.pc + 1) :: C \mapsto R'), H, S \rangle
\] |
| **MOVE** / \( P(c.m.pc) = \text{MOVE}_b \odot i \ j \) | \[
R[j] = \text{val}_b \\
R' = R[i \mapsto \text{val}_b]
\]
\[
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_p \langle ((c.m.pc + 1) :: C \mapsto R'), H, S \rangle
\] |
| **CONST** / \( P(c.m.pc) = \text{CONST}_b \odot i \ \text{val}_b \) | \[
R[i] = \text{val}_b
\]
\[
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_p \langle ((c.m.pc + 1) :: C \mapsto R'), H, S \rangle
\] |
| **CMP** / \( P(c.m.pc) = \text{CMP}_{\tau, \cdot} \odot i \ j \ g \) | \[
\tau \in \{\text{float}, \text{double}, \text{long}\} \\
R[j] = \text{val}_\tau \\
R[g] = \text{val}'_\tau \\
R' = R[i \mapsto (\text{cmp}_{\tau,\odot\text{bias}}(\text{val}_\tau, \text{val}'_\tau))_{\text{byte}}]
\]
\[
\langle ((c.m.pc) :: C \mapsto R), H, S \rangle \\
\rightarrow_p \langle ((c.m.pc + 1) :: C \mapsto R'), H, S \rangle
\] |
<table>
<thead>
<tr>
<th>NOOP</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(c.m.pc) = \text{noop}$</td>
<td>$P(c.m.pc) = \text{goto } pc'$</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle \langle \langle c.m.pc \rangle :: C \mapsto R \rangle, H, S \rangle \rightarrow P \langle \langle (c.m.pc + 1) \rangle :: C \mapsto R \rangle, H, S \rangle$</td>
<td>$\langle \langle \langle c.m.pc \rangle :: C \mapsto R \rangle, H, S \rangle \rightarrow P \langle \langle (c.m.pc') \rangle :: C \mapsto R \rangle, H, S \rangle$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF-TRUE</th>
<th>IF-FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(c.m.pc) = \text{if } i \ j \ pc'$</td>
<td>$P(c.m.pc) = \text{if } i \ j \ pc'$</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle \langle \langle c.m.pc \rangle :: C \mapsto R \rangle, H, S \rangle \rightarrow P \langle \langle (c.m.pc') \rangle :: C \mapsto R \rangle, H, S \rangle$</td>
<td>$\langle \langle \langle c.m.pc \rangle :: C \mapsto R \rangle, H, S \rangle \rightarrow P \langle \langle (c.m.pc + 1) \rangle :: C \mapsto R \rangle, H, S \rangle$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IFZ-TRUE</th>
<th>IFZ-FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(c.m.pc) = \text{ifz } i \ pc'$</td>
<td>$P(c.m.pc) = \text{ifz } i \ pc'$</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle \langle \langle c.m.pc \rangle :: C \mapsto R \rangle, H, S \rangle \rightarrow P \langle \langle (c.m.pc') \rangle :: C \mapsto R \rangle, H, S \rangle$</td>
<td>$\langle \langle \langle c.m.pc \rangle :: C \mapsto R \rangle, H, S \rangle \rightarrow P \langle \langle (c.m.pc + 1) \rangle :: C \mapsto R \rangle, H, S \rangle$</td>
</tr>
</tbody>
</table>
### Table A.7 Instruction switch

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPARSE-SWITCH-TRUE</strong></td>
<td>( P(c.m.p c) = \text{SPARSE-SWITCH } i , pc' )</td>
<td>( P(c.m.p c') = \text{sparseSwitch} ) ( R[i] \in \text{sparseSwitch} ) ( \text{sparseSwitch}[R[i]] = pc'' )</td>
</tr>
<tr>
<td></td>
<td>( (((c.m.p c) \mapsto R), H, S) )</td>
<td>( \rightarrow_p (((c.m.p c') \mapsto R), H, S) )</td>
</tr>
<tr>
<td><strong>SPARSE-SWITCH-FALSE</strong></td>
<td>( P(c.m.p c) = \text{SPARSE-SWITCH } i , pc' )</td>
<td>( P(c.m.p c') = \text{sparseSwitch} ) ( R[i] \notin \text{sparseSwitch} )</td>
</tr>
<tr>
<td></td>
<td>( (((c.m.p c) \mapsto R), H, S) )</td>
<td>( \rightarrow_p (((c.m.p c') + 1) \mapsto R), H, S) )</td>
</tr>
<tr>
<td><strong>PACKED-SWITCH-TRUE</strong></td>
<td>( P(c.m.p c) = \text{PACKED-SWITCH } i , pc' )</td>
<td>( P(c.m.p c') = \text{packedSwitch} ) ( id = R[i] - \text{packedSwitch}.fst ) ( id \in \text{packedSwitch}.pst ) ( \text{packedSwitch}.pst[id] = pc'' )</td>
</tr>
<tr>
<td></td>
<td>( (((c.m.p c) \mapsto R), H, S) )</td>
<td>( \rightarrow_p (((c.m.p c') \mapsto R), H, S) )</td>
</tr>
<tr>
<td><strong>PACKED-SWITCH-FALSE</strong></td>
<td>( P(c.m.p c) = \text{PACKED-SWITCH } i , pc' )</td>
<td>( P(c.m.p c') = \text{packedSwitch} ) ( id = R[i] - \text{packedSwitch}.fst ) ( id \notin \text{packedSwitch}.pst )</td>
</tr>
<tr>
<td></td>
<td>( (((c.m.p c) \mapsto R), H, S) )</td>
<td>( \rightarrow_p (((c.m.p c') + 1) \mapsto R), H, S) )</td>
</tr>
</tbody>
</table>

### Table A.8 Operations on static heap

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SGET</strong></td>
<td>( P(c.m.p c) = \text{SGET } i , c , f )</td>
<td>( S[c.f] = \text{val}_r ) ( R' = R[i \mapsto \text{val}_r] )</td>
</tr>
<tr>
<td></td>
<td>( (((c.m.p c) \mapsto R), H, S) )</td>
<td>( \rightarrow_p (((c.m.p c + 1) \mapsto R'), H, S) )</td>
</tr>
<tr>
<td><strong>SPUT</strong></td>
<td>( P(c.m.p c) = \text{SPUT } i , c , f )</td>
<td>( R[i] = \text{val}_r ) ( S'[i] = S[c.f \mapsto \text{val}_r] )</td>
</tr>
<tr>
<td></td>
<td>( (((c.m.p c) \mapsto R), H, S) )</td>
<td>( \rightarrow_p (((c.m.p c + 1) \mapsto R), H, S') )</td>
</tr>
</tbody>
</table>
Table A.9 Operations with objects

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW-INSTANCE</td>
<td>(P(c.m.pc) = \text{NEW-INSTANCE} i c')</td>
</tr>
<tr>
<td></td>
<td>(c'.f \in P(c'.f) \quad o = f \mapsto 0)</td>
</tr>
<tr>
<td></td>
<td>(H' = H[(\ell, (c.m.pc) :: C, c', 0) \mapsto o] \quad R' = R[i \mapsto (\ell, (c.m.pc) :: C, c', 0)])</td>
</tr>
<tr>
<td></td>
<td>(\langle((c.m.pc) :: C \mapsto R), H, S\rangle \quad \langle((c.m.pc + 1) :: C \mapsto R'), H', S\rangle)</td>
</tr>
<tr>
<td>INSTANCE-OF</td>
<td>(P(c.m.pc) = \text{INSTANCE-OF} i j c')</td>
</tr>
<tr>
<td></td>
<td>(R[j] = (\ell, C', c'', \text{ver}) \quad R' = R[i \mapsto (c'' \leq_p c')_\text{bool}])</td>
</tr>
<tr>
<td></td>
<td>(\langle((c.m.pc) :: C \mapsto R), H, S\rangle \quad \langle((c.m.pc + 1) :: C \mapsto R'), H, S\rangle)</td>
</tr>
<tr>
<td>CHECK-CAST</td>
<td>(P(c.m.pc) = \text{CHECK-CAST} i c')</td>
</tr>
<tr>
<td></td>
<td>(R[j] = (\ell, C', c'', \text{ver}) \quad c'' \leq_p c')</td>
</tr>
<tr>
<td></td>
<td>(\langle((c.m.pc) :: C \mapsto R), H, S\rangle \quad \langle((c.m.pc + 1) :: C \mapsto R), H, S\rangle)</td>
</tr>
<tr>
<td>IPUT</td>
<td>(P(c.m.pc) = \text{IPUT}_\tau i j c' f)</td>
</tr>
<tr>
<td></td>
<td>(R[i] = \text{val}_\tau \quad R[j] = (\ell, C', c', \text{ver}))</td>
</tr>
<tr>
<td></td>
<td>(H[(\ell, C', c', \text{ver})] = o \quad o' = o[f \mapsto \text{val}_\tau] \quad H' = H[(\ell, C', c', \text{ver} + 1) \mapsto o'])</td>
</tr>
<tr>
<td></td>
<td>(R' = R[j \mapsto (\ell, C', c', \text{ver} + 1)])</td>
</tr>
<tr>
<td></td>
<td>(\langle((c.m.pc) :: C \mapsto R), H, S\rangle \quad \langle((c.m.pc + 1) :: C \mapsto R'), H', S\rangle)</td>
</tr>
<tr>
<td>IGET</td>
<td>(P(c.m.pc) = \text{IGET}_\tau i j c' f)</td>
</tr>
<tr>
<td></td>
<td>(R[j] = (\ell, C', c', \text{ver}))</td>
</tr>
<tr>
<td></td>
<td>(H[(\ell, C', c', \text{ver})] = o \quad o[f] = \text{val}_\tau)</td>
</tr>
<tr>
<td></td>
<td>(R' = R[i \mapsto \text{val}_\tau])</td>
</tr>
<tr>
<td></td>
<td>(\langle((c.m.pc) :: C \mapsto R), H, S\rangle \quad \langle((c.m.pc + 1) :: C \mapsto R), H, S\rangle)</td>
</tr>
<tr>
<td>CONST-STRING</td>
<td>(P(c.m.pc) = \text{CONST-STRING} i \text{str})</td>
</tr>
<tr>
<td></td>
<td>(o = [\text{value} \mapsto \text{str}])</td>
</tr>
<tr>
<td></td>
<td>(H' = H[(\ell, (c.m.pc) :: C, \text{String}, 0) \mapsto o] \quad R' = R[i \mapsto (\ell, (c.m.pc) :: C, \text{String}, 0)])</td>
</tr>
<tr>
<td></td>
<td>(\langle((c.m.pc) :: C \mapsto R), H, S\rangle \quad \langle((c.m.pc + 1) :: C \mapsto R'), H', S\rangle)</td>
</tr>
<tr>
<td>CONST-CLASS</td>
<td>(P(c.m.pc) = \text{CONST-CLASS} i \text{str})</td>
</tr>
<tr>
<td></td>
<td>(o = [\text{value} \mapsto \text{str}] \quad o' = [\text{name} \mapsto (\ell, (c.m.pc) :: C, \text{String}, 0)])</td>
</tr>
<tr>
<td></td>
<td>(H' = H[(\ell, (c.m.pc) :: C, \text{String}, 0) \mapsto o, (\ell', (c.m.pc) :: C, \text{Object}, 0) \mapsto o'])</td>
</tr>
<tr>
<td></td>
<td>(R' = R[i \mapsto (\ell', (c.m.pc) :: C, \text{Object}, 0)])</td>
</tr>
<tr>
<td></td>
<td>(\langle((c.m.pc) :: C \mapsto R), H, S\rangle \quad \langle((c.m.pc + 1) :: C \mapsto R'), H', S\rangle)</td>
</tr>
</tbody>
</table>
Table A.10 Operations with arrays

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW-ARRAY / ( P(c.m.pc) = \text{NEW-ARRAY} i j \tau_a )</td>
<td>( \alpha = 0 ) [R[j] = val_{\text{int}} \quad</td>
<td>\alpha</td>
</tr>
<tr>
<td>ARRAY-LENGTH / ( P(c.m.pc) = \text{ARRAY-LENGTH} i j )</td>
<td>( R[j] = (\ell, C', \tau_a, \text{ver}) )</td>
<td>( H[\ell, C', \tau_a, \text{ver}] = \alpha \quad</td>
</tr>
<tr>
<td>INSTANCE-OF-ARRAY / ( P(c.m.pc) = \text{INSTANCE-OF} i j \tau_a )</td>
<td>( R[j] = (\ell, C', \tau_a', \text{ver}) )</td>
<td>( R' = R[i \mapsto (\tau_a' \leq \tau_a)_{\text{bool}}] )</td>
</tr>
<tr>
<td>CHECK-CAST-ARRAY / ( P(c.m.pc) = \text{CHECK-CAST} i \tau_a )</td>
<td>( R[i] = (\ell, C', \tau_a', \text{ver}) )</td>
<td>( \tau_a' \leq \tau_a )</td>
</tr>
<tr>
<td>APUT / ( P(c.m.pc) = \text{APUT}_\tau i j g )</td>
<td>( R[i] = val_{\tau} ) [R[j] = (\ell, C', \tau_a, \text{ver}) ]</td>
<td>( H[\ell, C', \tau_a, \text{ver}] = \alpha ) [ H' = H[\ell, C', \tau_a, \text{ver} + 1) \mapsto \alpha'] ]</td>
</tr>
<tr>
<td>AGET / ( P(c.m.pc) = \text{AGET}_\tau i j g )</td>
<td>( R[j] = (\ell, C', \tau_a, \text{ver}) )</td>
<td>( H[\ell, C', \tau_a, \text{ver}] = \alpha ) [ \alpha[R[g]] = val_{\tau} ] [ R' = R[i \mapsto val_{\tau}] ]</td>
</tr>
</tbody>
</table>
Table A.11 Operations with arrays: continuation

\[
\text{FILL-ARRAY-DATA} / P(c.m.pc) = \text{FILL-ARRAY-DATA} i pc'
\]

\[
R[i] = (\ell, C', \tau_a, \text{ver})
\]

\[
H[(\ell, C', \tau_a, \text{ver})] = \alpha \quad P(c.m.pc') = \text{arrayData} \quad |\text{arrayData}| \leq |\alpha|
\]

\[
\text{arrayData}[i_0] \ldots \text{arrayData}[|\text{arrayData}| - 1] = \text{val}_b \ldots \text{val}'_b \quad b \leq \tau_a
\]

\[
\alpha' = \alpha[\theta \mapsto \text{val}_b \ldots |\text{arrayData}| - 1 \mapsto \text{val}'_b]
\]

\[
H' = H[(\ell, C', \tau_a, \text{ver} + 1) \mapsto \alpha']
\]

\[
R' = R[i \mapsto (\ell, C', \tau_a, \text{ver} + 1)]
\]

\[
\langle ((c.m.pc) \mapsto R), H, S \rangle \rightarrow_P \langle ((c.m.pc + 1) \mapsto R'), H', S \rangle
\]

\[
\text{FILLED-NEW-ARRAY} / P(c.m.pc) = \text{FILLED-NEW-ARRAY} i_1 \ldots i_n \tau_a
\]

\[
R[i_1] = \text{val}_b \ldots R[i_n] = \text{val}'_b \quad b \leq \tau_a \quad \alpha = [\theta \mapsto \text{val}_b \ldots n - 1 \mapsto \text{val}'_b]
\]

\[
|\alpha| = n \quad H' = H[(\ell, C', \tau_a, 0) \mapsto \alpha] \quad R' = R[\text{ret} \mapsto (\ell, C', \tau_a, 0)]
\]

\[
\langle ((c.m.pc) \mapsto R), H, S \rangle \rightarrow_P \langle ((c.m.pc + 1) \mapsto R'), H', S \rangle
\]
<table>
<thead>
<tr>
<th>Method invocation and return</th>
</tr>
</thead>
</table>

**INVOKE / P(c.m) = INVOKE c' m' i₁ ... iₙ**

\[
P(c'.m').\text{numloc} = \text{val}_{\text{int}}\quad R[i_1] = \text{val}_r\ldots R[i_n] = \text{val}_{r'}
\]

\[
R' = [0 \mapsto 0 \ldots \text{val}_{\text{int}} - 1 \mapsto 0, \text{val}_{\text{int}} \mapsto \text{val}_r \ldots \text{val}_{\text{int}} + n - 1 \mapsto \text{val}_{r'}]
\]

\[
C' = (c.m.p) :: C
\]

\[
\langle ((c.m.p) :: C \mapsto R), H, S \rangle
\]

\[
\rightarrow_p \langle ((c'.m'.0) :: C' \mapsto R'), H, S \rangle
\]

**INVOKE-REF / P(c.m) = INVOKE-REF c' m' i₁ ... iₙ**

\[
c' \leq_P c''
\]

\[
P(c'.m').\text{numloc} = \text{val}_{\text{int}}\quad R[i_1] = \text{val}_r\ldots R[i_n] = \text{val}_{r'}
\]

\[
R' = [0 \mapsto 0 \ldots \text{val}_{\text{int}} - 1 \mapsto 0, \text{val}_{\text{int}} \mapsto \text{val}_r \ldots \text{val}_{\text{int}} + n - 1 \mapsto \text{val}_{r'}]
\]

\[
C' = (c.m.p) :: C
\]

\[
\langle ((c.m.p) :: C \mapsto R), H, S \rangle
\]

\[
\rightarrow_p \langle ((c'.m'.0) :: C' \mapsto R'), H, S \rangle
\]

**RETURN-VOID / P(c.m) = RETURN-VOID**

\[
\Sigma_P((c'.m'.p)c') :: C) = R'
\]

\[
\langle ((c.m.p) :: (c'.m'.p)c') :: C \mapsto R), H, S \rangle
\]

\[
\rightarrow_p \langle ((c'.m'.p)c' + 1) :: C \mapsto R'), H, S \rangle
\]

**RETURN / P(c.m) = RETURNᵣ i**

\[
\Sigma_P((c'.m'.p)c') :: C) = R'
\]

\[
\langle ((c.m.p) :: (c'.m'.p)c') :: C \mapsto R), H, S \rangle
\]

\[
\rightarrow_p \langle ((c'.m'.p)c' + 1) :: C \mapsto R'[\text{ret} \mapsto R[i]], H, S \rangle
\]

**MOVE-RESULT / P(c.m) = MOVE-RESULTᵣ i**

\[
R[\text{ret}] = \text{val}_r\quad R' = R[i \mapsto \text{val}_r]
\]

\[
\langle ((c.m.p) :: C \mapsto R), H, S \rangle
\]

\[
\rightarrow_p \langle ((c.m.p + 1) :: C \mapsto R'), H, S \rangle
\]
Appendix B

Abstract Semantics

Table B.1 Abstract syntax

\[
\begin{align*}
p & ::= \text{field } c' f \mid \text{move, } c \ m \ pc \ i \ j \\
    & \mid \text{sparse-switch } c \ m \ pc \ i \ pc' \\
    & \mid \text{sswitch-data } c \ m \ pc' \\
    & \mid \text{invoke-ref } c \ m \ pc \ c' \ m' \\
    & \mid \text{par } c \ m \ pc \ i \ ind \\
\end{align*}
\]

\[\hat{p}\]  
App

\[e \in \text{string}\]  
Exception name

\[e ::= (C \ C' \ e)\]  
Exception

Table B.2 Abstract analysis predicates

\[
\begin{align*}
L & ::= c \ m \ pc \\
C & ::= L :: C \mid L \\
x & ::= C \ (i \mid \text{ret}) \ val_r \\
h & ::= (C \ (c \mid \tau_a) \ ver) \ (f \mid i) \ val_r \\
s & ::= c \ f \ val_r \\
e & ::= (C \ e) \\
\Delta & ::= (\hat{r}, \hat{h}, \hat{s}, \hat{e})
\end{align*}
\]

Local state

Call stack

Register predicate

Heap predicate

Static heap predicate

Exception predicate

Analysis state
Table B.3 Binary and unary operations, move, const, cmp

BINOP
\[
\text{binop}_b \oplus c m pc i j g \\
\land \tau (c m pc) :: C j \text{val}_b \\
\land \tau (c m pc) :: C g \text{val}'_b
\implies \tau (c m pc + 1) :: C i \text{val}_b \oplus \text{val}'_b \\
\land \hat{\text{r}} \neq i (c m pc+1) :: C
\]

BINOP – 2
\[
\text{binop-2}_b \oplus c m pc i j \\
\land \tau (c m pc) :: C i \text{val}_b \\
\land \tau (c m pc) :: C j \text{val}'_b
\implies \tau (c m pc + 1) :: C i \text{val}_b \oplus \text{val}'_b \\
\land \hat{\text{r}} \neq i (c m pc+1) :: C
\]

BINOP – LIT
\[
\text{binop-lit}_b \oplus c m pc i j \text{val}'_b \\
\land \tau (c m pc) :: C j \text{val}_b
\implies \tau (c m pc + 1) :: C i \text{val}_b \oplus \text{val}'_b \\
\land \hat{\text{r}} \neq i (c m pc+1) :: C
\]

UNOP
\[
\text{unop}_b \odot c m pc i \text{val}_b \\
\land \tau (c m pc) :: C j \text{val}_b
\implies \tau (c m pc + 1) :: C i \odot \text{val}_b \\
\land \hat{\text{r}} \neq i (c m pc+1) :: C
\]

UNOP – TYPE
\[
\text{unop}_b v c m pc i j \\
\land \tau (c m pc) :: C j \text{val}_b
\implies \tau (c m pc + 1) :: C i (\text{val}_b)_v \\
\land \hat{\text{r}} \neq i (c m pc+1) :: C
\]

MOVE
\[
\text{move}_\tau c m pc i j \land \tau (c m pc) :: C j \text{val}_\tau
\implies \tau (c m pc + 1) :: C i \text{val}_\tau \\
\land \hat{\text{r}} \neq i (c m pc+1) :: C
\]

CONST
\[
\text{const}_b c m pc i \text{val}_b \\
\land \tau (c m pc) :: C i \text{val}'_\tau
\implies \tau (c m pc + 1) :: C i \text{val}_b \\
\land \hat{\text{r}} \neq i (c m pc+1) :: C
\]

CMP
\[
\text{cmp}_\tau, | \text{bias} c m pc i j g \\
\land \tau \in \{\text{float, double, long}\} \\
\land \tau (c m pc) :: C j \text{val}_\tau \\
\land \tau (c m pc) :: C g \text{val}'_\tau
\implies \tau (c m pc + 1) :: C i \text{cmp}_\tau, | \text{bias} (\text{val}_\tau, \text{val}'_\tau) \\
\land \hat{\text{r}} \neq i (c m pc+1) :: C
\]
Table B.4 Conditional and unconditional jumps, noop

**NOOP**
\[
\text{noop } c m pc \land r(c m pc) :: C i val_r \implies r(c m pc + 1) :: C i val_r
\]

**GOTO**
\[
goto c m pc pc' \land r(c m pc) :: C i val_r \implies r(c m pc') :: C i val_r
\]

**IF – TRUE**
\[
\text{if}_\odot c m pc i j pc' \\
\land r(c m pc) :: C i val_r \\
\land r(c m pc) :: C j val'_r \\
\land r(c m pc) :: C g val''_r \\
\land val_r \odot val'_r
\implies r(c m pc') :: C g val''_r
\]

**IF – FALSE**
\[
\text{if}_\odot c m pc i j pc' \\
\land r(c m pc) :: C i val_r \\
\land r(c m pc) :: C j val'_r \\
\land r(c m pc) :: C g val''_r \\
\land val_r \odot val'_r \\
\land \neg(val_r \odot val'_r)
\implies r(c m pc + 1) :: C g val''_r
\]

**IFZ – TRUE**
\[
\text{ifz}_\odot c m pc i pc' \\
\land r(c m pc) :: C i val_r \\
\land r(c m pc) :: C g val''_r \\
\land val_r \odot 0
\implies r(c m pc') :: C g val''_r
\]

**IFZ – FALSE**
\[
\text{ifz}_\odot c m pc i pc' \\
\land r(c m pc) :: C i val_r \\
\land r(c m pc) :: C g val''_r \\
\land \neg(val_r \odot 0)
\implies r(c m pc + 1) :: C g val''_r
\]
Table B.5 Instruction switch

**SPARSE − SWITCH − TRUE**

\[
\begin{align*}
\text{sparse-switch } & c \ m \ pc \ i \ pc' \\
\land & \text{sswitch-data } c \ m \ pc' \\
\land & r (c \ m \ pc) :: C \ i \ val_{\text{int}} \\
\land & \text{sswitch-data-item } c \ m \ pc' \ i \ val_{\text{int}} \ pc'' \\
\land & r (c \ m \ pc) :: C \ w \ val_r \\
\implies & r (c \ m \ pc'') :: C \ w \ val_r
\end{align*}
\]

**SPARSE − SWITCH − FALSE**

\[
\begin{align*}
\text{sparse-switch } & c \ m \ pc \ i \ pc' \\
\land & \text{sswitch-data } c \ m \ pc' \\
\land & r (c \ m \ pc) :: C \ i \ val_{\text{int}} \\
\land & \text{sswitch-data-item } c \ m \ pc' \ i \ val_{\text{int}} \ pc'' \\
\land & \neg(val_{\text{int}} = val'_{\text{int}}) \\
\land & r (c \ m \ pc) :: C \ w \ val_r \\
\implies & r (c \ m \ pc + 1) :: C \ w \ val_r
\end{align*}
\]

**PACKED − SWITCH − TRUE**

\[
\begin{align*}
\text{packed-switch } & c \ m \ pc \ i \ pc' \\
\land & \text{packed-data } c \ m \ pc' \ i \ val'_{\text{int}} \\
\land & r (c \ m \ pc) :: C \ i \ val_{\text{int}} \\
\land & \text{packed-data-item } c \ m \ pc' \ i \ val_{\text{int}} - val'_{\text{int}} \ pc'' \\
\land & r (c \ m \ pc) :: C \ w \ val_r \\
\implies & r (c \ m \ pc'') :: C \ w \ val_r
\end{align*}
\]

**PACKED − SWITCH − FALSE**

\[
\begin{align*}
\text{packed-switch } & c \ m \ pc \ i \ pc' \\
\land & \text{packed-data } c \ m \ pc' \ i \ val'_{\text{int}} \\
\land & r (c \ m \ pc) :: C \ i \ val_{\text{int}} \\
\land & \text{packed-data-item } c \ m \ pc' \ i \ val_{\text{int}} - val'_{\text{int}} \\
\land & \neg(id = val_{\text{int}} - val'_{\text{int}}) \\
\land & r (c \ m \ pc) :: C \ w \ val_r \\
\implies & r (c \ m \ pc + 1) :: C \ w \ val_r
\end{align*}
\]
### Table B.6 Operations on static heap

**SGET**

\[
\text{sget}_r \ c \ m \ pc \ i \ c' \ f \\
\land \ s \ c' \ f \ \text{val}_r \\
\land \ r \ (c \ m \ pc) :: C \ i \ \text{val}_r'
\implies \ r \ (c \ m \ pc + 1) :: C \ i \ \text{val}_r \\
\land \ \widehat{r} \neq \ i \ (c \ m \ pc+1) :: C
\]

**SPUT**

\[
\text{sput}_r \ c \ m \ pc \ i \ c' \ f \\
\land \ r \ (c \ m \ pc) :: C \ i \ \text{val}_r
\implies \ s \ c' \ f \ \text{val}_r
\]
Table B.7 Operations with objects

**NEW – INSTANCE**

\[
\text{new-instance } c \ m \ pc \ i \ c' \\
\land \text{ field } c' \ f \\
\land r (c m pc) :: C i \text{ val}_r \\
\Rightarrow r (c m pc + 1) :: C i ((c m pc) :: C c' 0) \\
\land h ((c m pc) :: C c' 0) f 0 \\
\land \widehat{r}_{\neq i} (c m pc + 1) :: C
\]

**INSTANCE – OF**

\[
\text{instance-of } c \ m \ pc \ i j \ c' \\
\land r (c m pc) :: C j (C c'' ver) \\
\land \text{ super } c'' c' \\
\land r (c m pc) :: C i \text{ val}_r \\
\Rightarrow r (c m pc + 1) :: C w \text{ val}_r \\
\land \widehat{r}_{\neq i} (c m pc + 1) :: C
\]

**CHECK – CAST**

\[
\text{check-cast } c \ m \ pc \ i \ c' \\
\land r (c m pc) :: C i (C c'' \text{ ver}) \\
\land \text{ super } c'' c' \\
\land r (c m pc) :: C w \text{ val}_r \\
\Rightarrow h (C' c' \text{ ver } + 1) f \text{ val}_r \\
\land r (c m pc + 1) :: C j (C c' \text{ ver } + 1) \\
\land \widehat{r}_{\neq j} (c m pc + 1) :: C \\
\land h_{\neq f} (C' c' \text{ ver } + 1)
\]

**IPUT**

\[
\text{iput}_r c m pc i j c' \ f \\
\land r (c m pc) :: C j (C c' \text{ ver}) \\
\land r (c m pc) :: C i \text{ val}_r \\
\Rightarrow h (C' c' \text{ ver }) f \text{ val}_r \\
\land r (c m pc + 1) :: C j (C c' \text{ ver } + 1) \\
\land \widehat{r}_{\neq j} (c m pc + 1) :: C \\
\land h_{\neq f} (C' c' \text{ ver } + 1)
\]

**IGET**

\[
\text{iget}_r c m pc i j c' \ f \\
\land h (C' c' \text{ ver}) f \text{ val}_r \\
\land h (C' c' \text{ ver}) \\
\Rightarrow r (c m pc + 1) :: C i \text{ val}_r \\
\land \widehat{r}_{\neq i} (c m pc + 1) :: C
\]

**CONST – STRING**

\[
\text{const-string } c m pc \ i \ str \\
\land r (c m pc) :: C i \text{ val}_r \\
\Rightarrow h ((c m pc) :: C \text{ String } 0) \text{ value str} \\
\land r (c m pc + 1) :: C i ((c m pc) :: C \text{ String } 0) \\
\land \widehat{r}_{\neq i} (c m pc + 1) :: C
\]

**CONST – CLASS**

\[
\text{const-class } c m pc \ i \ str \\
\land r (c m pc) :: C i \text{ val}_r \\
\Rightarrow h ((c m pc) :: C \text{ String } 0) \text{ value str} \\
\land h ((c m pc) :: C \text{ Object } 0) \text{ name ((c m pc) :: C \text{ String } 0)} \\
\land r (c m pc) :: C i ((c m pc) :: C \text{ Object } 0) \\
\land \widehat{r}_{\neq i} (c m pc + 1) :: C
\]
Table B.8 Operations with arrays

**NEW – ARRAY**

\[
\text{new-array } c \ m \ pc \ i \ j \ \tau_a \\
\land \ r (c \ m \ pc) :: C \ j \ val_{\text{int}} \\
\implies \ r (c \ m \ pc + 1) :: C \ i \ ((c \ m \ pc) :: C \ \tau_a \ 0) \\
\land \ size ((c \ m \ pc) :: C) \ val_{\text{int}} \\
\land \ h ((c \ m \ pc) :: C \ \tau_a \ 0) \ ind \ 0 \\
\land \ 0 \leq \ ind \leq \ val_{\text{int}} - 1 \\
\land \ r \neq i \ ((c \ m \ pc+1) :: C)
\]

**ARRAY – LENGTH**

\[
\text{array-length } c \ m \ pc \ i \ j \\
\land \ r (c \ m \ pc) :: C \ j (C' \ \tau_a \ \text{ver}) \\
\land \ size (C') \ val_{\text{int}} \\
\implies \ r (c \ m \ pc + 1) :: C \ i \ val_{\text{int}} \\
\land \ r \neq i \ ((c \ m \ pc+1) :: C)
\]

**INSTANCE – OF – ARRAY**

\[
\text{instance-of-array } c \ m \ pc \ i \ j \ \tau_a \\
\land \ r (c \ m \ pc) :: C \ j (C' \ \tau_a' \ \text{ver}) \\
\implies \ r (c \ m \ pc + 1) :: C \ i \ super \ \tau_a' \ \tau_a \\
\land \ r \neq i \ ((c \ m \ pc+1) :: C)
\]

**CHECK – CAST – ARRAY**

\[
\text{check-cast } c \ m \ pc \ i \ \tau_a' \\
\land \ r (c \ m \ pc) :: C \ j (C' \ \tau_a \ \text{ver}) \\
\land \ super \ \tau_a \ \tau_a' \\
\land \ r (c \ m \ pc) :: C \ w \ val_r \\
\implies \ r (c \ m \ pc + 1) :: C \ w \ val_r \\
\land \ r \neq i \ ((c \ m \ pc+1) :: C)
\]

**APUT**

\[
\text{aput}_r \ c \ m \ pc \ i \ j \ g \\
\land \ r (c \ m \ pc) :: C \ j (C' \ \tau_a \ \text{ver}) \\
\land \ r (c \ m \ pc) :: C \ i \ val_r \\
\land \ r (c \ m \ pc) :: C \ g \ val'_b \\
\land \ super \ \tau \ \tau_a \\
\implies \ h (C' \ \tau_a \ \text{ver} + 1) \ val'_b \ val_r \\
\land \ r (c \ m \ pc + 1) :: C \ j (C' \ \tau_a \ \text{ver} + 1) \\
\land \ r \neq j \ ((c \ m \ pc+1) :: C) \\
\land \ h \neq val'_b \ (C' \ \tau_a \ \text{ver} + 1)
\]

**AGET**

\[
\text{aget}_r \ c \ m \ pc \ i \ j \ g \\
\land \ r (c \ m \ pc) :: C \ j (C' \ \tau_a \ \text{ver}) \\
\land \ r (c \ m \ pc) :: C \ g \ val'_b \\
\land \ h (C' \ \tau_a \ \text{ver}) \ val'_b \ val_r \\
\implies \ r (c \ m \ pc) :: C + 1 \ i \ val_r \\
\land \ r \neq i \ ((c \ m \ pc+1) :: C)
\]
Table B.9 Operations with arrays: continuation

**FILL – ARRAY – DATA**

<table>
<thead>
<tr>
<th>fill-array-data c m pc i pc'</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\land r (c m pc) :: C i (C' \tau_a \text{ ver}))</td>
</tr>
<tr>
<td>(\land \text{data } c m pc' \text{ val}_{\text{int}})</td>
</tr>
<tr>
<td>(\land \text{data-item } c m pc' \text{ ind } \text{val}_b)</td>
</tr>
<tr>
<td>(\land \text{size } (C') \text{ val}'_{\text{int}})</td>
</tr>
<tr>
<td>(\land \text{val}<em>{\text{int}} \leq \text{val}'</em>{\text{int}})</td>
</tr>
<tr>
<td>(\land \text{super } b \tau_a)</td>
</tr>
</tbody>
</table>

\[ \Rightarrow r (c m pc + 1) :: C i (C' \tau_a \text{ ver + } 1) \]
\[ \land h (C' \tau_a \text{ ver + } 1) \text{ ind } \text{val}_b\]
\[ \land \text{super } b \tau_a\]

**FILLED – NEW – ARRAY**

<table>
<thead>
<tr>
<th>filled-new-array c m pc (\tau_a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\land \text{numpar } c m pc \text{ val}_{\text{int}})</td>
</tr>
<tr>
<td>(\land \text{par } c m pc i \text{ ind})</td>
</tr>
<tr>
<td>(\land r (c m pc) :: C i \text{ val}_b)</td>
</tr>
<tr>
<td>(\land \text{super } b \tau_a)</td>
</tr>
</tbody>
</table>

\[ \Rightarrow \text{size } ((c m pc) :: C) \text{ val}_{\text{int}}\]
\[ \land h ((c m pc) :: C \tau_a 0) \text{ ind } \text{val}_b\]
\[ \land r (c m pc + 1) :: C \text{ ret } ((c m pc) :: C \tau_a 0)\]
\[ \land \text{super } b \tau_a\]

\[ \Rightarrow \text{size } ((c m pc) :: C) \text{ val}_{\text{int}}\]
\[ \land h ((c m pc) :: C \tau_a 0) \text{ ind } \text{val}_b\]
\[ \land r (c m pc + 1) :: C \text{ ret } ((c m pc) :: C \tau_a 0)\]
\[ \land \text{super } b \tau_a\]
Table B.10 Method invocation and return

**INVOKE**

\[
\text{invoke } c \ m \ pc \ c' \ m' \\
\land \ par \ c \ m \ pc \ i \ ind \\
\land \ r \ (c \ m \ pc) :: C \ i \ val_r \\
\land \ numloc \ c' \ m' \ val_{\text{int}}
\Rightarrow \quadr (c' \ m' \ 0) :: (c \ m \ pc) :: C \ ind + val_{\text{int}} \ val_r \\
\land \ r (c' \ m' \ 0) :: (c \ m \ pc) :: C \ ind' \ 0 \\
\land \ 0 \leq \ ind' \leq \ val_{\text{int}}
\]

**INVOCEREF**

\[
\text{invoke-ref } c \ m \ pc \ c' \ m' \\
\land \ par \ c \ m \ pc \ i \ ind \\
\land \ r \ (c \ m \ pc) :: C \ i \ val_r \\
\land \ par \ c \ m \ pc \ j \ 0 \\
\land \ r \ (c \ m \ pc) :: C \ j \ (C' \ c'' \ \text{ver}) \\
\land \ super \ c'' \ c' \\
\land \ numloc \ c' \ m' \ val_{\text{int}} \\
\Rightarrow \quadr (c' \ m' \ 0) :: (c \ m \ pc) :: C \ ind + val_{\text{int}} \ val_r \\
\land \ r (c' \ m' \ 0) :: (c \ m \ pc) :: C \ ind' \ 0 \\
\land \ 0 \leq \ ind' \leq \ val_{\text{int}}
\]

**RETURN - VOID**

\[
\text{return-void } c \ m \ pc \\
\land \ r \ (c \ m \ pc) :: (c' \ m' \ pc') :: C \ i \ val_r \\
\Rightarrow \quadr (c' \ m' \ pc' + 1) :: C \ i \ val_r
\]

**RETURN**

\[
\text{return}_r c \ m \ pc \ i \\
\land \ r \ (c \ m \ pc) :: (c' \ m' \ pc') :: C \ i \ val_r \\
\land \ r \ (c' \ m' \ pc') :: C \ j \ val'_r, \\
\Rightarrow \quadr (c' \ m' \ pc' + 1) :: C \ ret \ val_r \\
\land \ r (c' \ m' \ pc' + 1) :: C \ j \ val'_r,
\]

**MOVE - RESULT**

\[
\text{move-result}_r c \ m \ pc \ i \\
\land \ r \ (c \ m \ pc) :: C \ ret \ val_r \\
\Rightarrow \quadr (c \ m \ pc + 1) :: C \ i \ val_r \\
\land \ \hat{x} \neq x (c \ m \ pc + i) :: C
\]
Table B.11 Analysis of throw and move-exception instructions

THROW

\[
\begin{align*}
\text{throw } c \ m \ pc \ i \\
\land r (c m pc) &:: C i ((c' m' pc') :: C' e 0) \\
\land \neg(\text{try } c m pc_{\text{start}} pc_{\text{end}} \ e pc_{\text{catch}}' \vee pc_{\text{start}}' \leq pc \leq pc_{\text{end}}' ) \\
\land \text{handler } c m C e (c'' m'' pc_{\text{catch}}') :: C'' \\
\land r (c m pc) &:: C w val_r \\
\implies r (c'' m'' pc_{\text{catch}}') :: C'' w val_r \\
\land e ((c'' m'' pc_{\text{catch}}') :: C'' (c' m' pe') :: C' e)
\end{align*}
\]

THROW – SELF

\[
\begin{align*}
\text{throw } c \ m \ pc \ i \\
\land r (c m pc) &:: C i ((c' m' pc') :: C' e 0) \\
\land \text{try } c m pc_{\text{start}} pc_{\text{end}} e pc_{\text{catch}} \\
\land pc_{\text{start}} \leq pc \leq pc_{\text{end}} \\
\land r (c m pc) &:: C w val_r \\
\implies r (c m pc_{\text{catch}}) :: C w val_r \\
\land e ((c m pc_{\text{catch}}) :: C (c' m' pe') :: C' e)
\end{align*}
\]

MOVE – EXCEPTION

\[
\begin{align*}
\text{move-exception } c \ m \ pc \ i \\
\land e ((c m pc) :: C (c' m' pe') :: C' e) \\
\implies r (c m pc + 1) :: C i ((c' m' pe') :: C' e 0) \\
\land \hat{\hat{r}} \neq i (c m pc+1) :: C
\end{align*}
\]
Table B.12 Static analysis of check-cast instructions

CHECK – CAST – FALSE

check-cast $c \; m \; pc \; i \; c'$
$\land \; r \; (c \; m \; pc) :: C \; i \; (C' \; c'' \; ver)$
$\land \; \neg (super \; c'' \; c')$
$\land \; \neg (try \; c \; m \; p_{start} \; p_{end}' \; e \; p_{catch}' \; \land \; p_{start}' \leq pc \leq p_{end}')$
$\land \; handler \; c \; m \; C \; ClassCast \; (c'' \; m'' \; p_{catch}) :: C''$
$\land \; r \; (c \; m \; pc) :: C' \; w \; val_r$

$\Rightarrow \; r \; (c'' \; m'' \; p_{catch}) :: C'' \; w \; val_r$
$\land \; e \; ((c'' \; m'' \; p_{catch}) :: C'' \; (c \; m \; pc) :: C \; ClassCast)$

CHECK – CAST – FALSE – SELF

check-cast $c \; m \; pc \; i \; c'$
$\land \; r \; (c \; m \; pc) :: C \; i \; (C' \; c'' \; ver)$
$\land \; \neg (super \; c'' \; c')$
$\land \; try \; c \; m \; p_{start} \; p_{end}' \; e \; p_{catch}$
$\land \; p_{start} \leq pc \leq p_{end}'$

$\Rightarrow \; r \; (c \; m \; p_{catch}) :: C \; w \; val_r$
$\land \; e \; ((c \; m \; p_{catch}) :: C \; (c \; m \; pc) :: C \; ClassCast)$

CHECK – CAST – ARRAY – FALSE

check-cast $c \; m \; pc \; i \; \tau_a'$
$\land \; r \; (c \; m \; pc) :: C' \; i \; (C' \; \tau_a \; v_a)$
$\land \; \neg (super \; \tau_a \; \tau_a')$
$\land \; \neg (try \; c \; m \; p_{start} \; p_{end}' \; e \; p_{catch}' \; \land \; p_{start}' \leq pc \leq p_{end}')$
$\land \; handler \; c \; m \; C \; ClassCast \; (c'' \; m'' \; p_{catch}) :: C''$
$\land \; r \; (c \; m \; pc) :: C' \; w \; val_r$

$\Rightarrow \; r \; (c'' \; m'' \; p_{catch}) :: C'' \; w \; val_r$
$\land \; e \; ((c'' \; m'' \; p_{catch}) :: C'' \; (c \; m \; pc) :: C \; ClassCast)$

CHECK – CAST – ARRAY – FALSE – SELF

check-cast $c \; m \; pc \; i \; \tau_a'$
$\land \; r \; (c \; m \; pc) :: C' \; i \; (C' \; \tau_a \; v_a)$
$\land \; \neg (super \; \tau_a \; \tau_a')$
$\land \; try \; c \; m \; p_{start} \; p_{end}' \; e \; p_{catch}$
$\land \; p_{start} \leq pc \leq p_{end}'$
$\land \; r \; (c \; m \; pc) :: C' \; w \; val_r$

$\Rightarrow \; r \; (c \; m \; p_{catch}) :: C \; w \; val_r$
$\land \; e \; ((c \; m \; p_{catch}) :: C \; (c \; m \; pc) :: C \; ClassCast)$
Table B.13 Static analysis of binary operations with exceptions

BINOP – EXCEPTION

\[ \begin{align*}
&\text{\texttt{BINOP}} - \text{\texttt{EXCEPTION}} \\
&\text{binop}_b \oplus c \ m \ pc \ i \ j \ g \\
&\land \ r \ (c \ m \ pc) :: C \ j \ val_b \\
&\land \ r \ (c \ m \ pc) :: C \ g \ val'_b \\
&\land \ \tau \in \{\text{long, int}\} \land \oplus \in \{/?, \%\} \land \text{val}'_b = 0 \\
&\land \ \neg(\text{try} \ c \ m \ pc'_{\text{start}} \ pc'_{\text{end}} e \ pc'_{\text{catch}} \lor \text{pc'_{start}} \leq \text{pc} \leq \text{pc'_{end}}) \\
&\land \ \text{handler} \ c \ m \ C \ \text{Arithmetic} \ (c'' \ m'' \ pc'_{\text{catch}}) :: C'' \\
&\land \ r \ (c \ m \ pc) :: C \ w \ val_r \\
&\implies r \ (c'' \ m'' \ pc'_{\text{catch}}) :: C'' \ w \ val_r \\
&\land \ e \ ((c'' \ m'' \ pc'_{\text{catch}}) :: C'' \ (c \ m \ pc) :: C \ \text{Arithmetic})
\end{align*} \]

BINOP – 2 – EXCEPTION

\[ \begin{align*}
&\text{\texttt{BINOP}} - 2 - \text{\texttt{EXCEPTION}} \\
&\text{binop}_2 \oplus c \ m \ pc \ i \ j \\
&\land \ r \ (c \ m \ pc) :: C \ i \ val_b \\
&\land \ r \ (c \ m \ pc) :: C \ j \ val'_b \\
&\land \ \tau \in \{\text{long, int}\} \land \oplus \in \{/?, \%\} \land \text{val}'_b = 0 \\
&\land \ \neg(\text{try} \ c \ m \ pc'_{\text{start}} \ pc'_{\text{end}} e \ pc'_{\text{catch}} \lor \text{pc'_{start}} \leq \text{pc} \leq \text{pc'_{end}}) \\
&\land \ \text{handler} \ c \ m \ C \ \text{Arithmetic} \ (c'' \ m'' \ pc'_{\text{catch}}) :: C'' \\
&\land \ r \ (c \ m \ pc) :: C \ w \ val_r \\
&\implies r \ (c'' \ m'' \ pc'_{\text{catch}}) :: C'' \ w \ val_r \\
&\land \ e \ ((c'' \ m'' \ pc'_{\text{catch}}) :: C'' \ (c \ m \ pc) :: C \ \text{Arithmetic})
\end{align*} \]

BINOP – LIT – EXCEPTION

\[ \begin{align*}
&\text{\texttt{BINOP}} - \text{\texttt{LIT}} - \text{\texttt{EXCEPTION}} \\
&\text{binop}_{\text{lit}} \oplus c \ m \ pc \ i \ j \ \text{val'}_b \\
&\land \ r \ (c \ m \ pc) :: C \ j \ val_b \\
&\land \ \tau \in \{\text{long, int}\} \land \oplus \in \{/?, \%\} \land \text{val}'_b = 0 \\
&\land \ \neg(\text{try} \ c \ m \ pc'_{\text{start}} \ pc'_{\text{end}} e \ pc'_{\text{catch}} \lor \text{pc'_{start}} \leq \text{pc} \leq \text{pc'_{end}}) \\
&\land \ \text{handler} \ c \ m \ C \ \text{Arithmetic} \ (c'' \ m'' \ pc'_{\text{catch}}) :: C'' \\
&\land \ r \ (c \ m \ pc) :: C \ w \ val_r \\
&\implies r \ (c'' \ m'' \ pc'_{\text{catch}}) :: C'' \ w \ val_r \\
&\land \ e \ ((c'' \ m'' \ pc'_{\text{catch}}) :: C'' \ (c \ m \ pc) :: C \ \text{Arithmetic})
\end{align*} \]
Table B.14 Static analysis of method calls with exceptions

**INVOKE**

\[
\text{invoke } c \quad m \quad pc \quad c' \quad m' \\
\land \quad \text{par } c \quad m \quad pc \quad i \quad \text{ind} \\
\land \quad r \quad (c \quad m \quad pc) \quad :: \quad C \quad i \quad val_r \\
\land \quad \text{numloc } c' \quad m' \quad val_{\text{int}} \\
\]

\[
\begin{align*}
\text{r} \quad (c' \quad m' \quad 0) & \quad :: \quad (c \quad m \quad pc) \quad :: \quad C \quad ind + val_{\text{int}} \quad val_r \\
\land \quad \text{r} \quad (c' \quad m' \quad 0) & \quad :: \quad (c \quad m \quad pc) \quad :: \quad C \quad ind' \quad 0 \\
\land \quad 0 \quad \leq \quad ind'' & \quad \leq \quad val_{\text{int}} \\
\land \quad (\text{try } c \quad m \quad pc_{\text{start}} \quad pc_{\text{end}} \quad e \quad pc_{\text{catch}}) \\
\Rightarrow \quad \land \quad pc_{\text{start}} \quad \leq \quad pc \quad \leq \quad pc_{\text{end}} \\
\Rightarrow \quad \text{handler } c' \quad m' \quad (c \quad m \quad pc) \quad :: \quad C \quad e \quad (c \quad m \quad pc_{\text{catch}}) \quad :: \quad C' \\
\land \quad (\text{handler } c \quad m \quad C \quad e' \quad (c'' \quad m'' \quad pc''_{\text{catch}}) \quad :: \quad C') \\
\land \quad \neg(\quad e' = \quad e) \\
\Rightarrow \quad \text{handler } c' \quad m' \quad (c \quad m \quad pc) \quad :: \quad C \quad e' \quad (c'' \quad m'' \quad pc''_{\text{catch}}) \quad :: \quad C')
\end{align*}
\]

**INVOKE – REF**

\[
\text{invoke-ref } c \quad m \quad pc \quad c' \quad m' \\
\land \quad \text{par } c \quad m \quad pc \quad i \quad \text{ind} \\
\land \quad r \quad (c \quad m \quad pc) \quad :: \quad C \quad i \quad val_r \\
\land \quad \text{par } c \quad m \quad pc \quad j \quad 0 \\
\land \quad r \quad (c \quad m \quad pc) \quad :: \quad C \quad j \quad (C' \quad c'' \quad ver) \\
\land \quad \text{super } c'' \quad c' \\
\land \quad \text{numloc } c' \quad m' \quad val_{\text{int}} \\
\]

\[
\begin{align*}
\text{r} \quad c' \quad m' \quad 0 & \quad :: \quad (c \quad m \quad pc) \quad :: \quad C \quad ind + val_{\text{int}} \quad val_r \\
\land \quad \text{r} \quad c' \quad m' \quad 0 & \quad :: \quad (c \quad m \quad pc) \quad :: \quad C \quad ind' \quad 0 \\
\land \quad 0 \quad \leq \quad ind'' & \quad \leq \quad val_{\text{int}} \\
\land \quad (\text{try } c \quad m \quad pc_{\text{start}} \quad pc_{\text{end}} \quad e \quad pc_{\text{catch}}) \\
\Rightarrow \quad \land \quad pc_{\text{start}} \quad \leq \quad pc \quad \leq \quad pc_{\text{end}} \\
\Rightarrow \quad \text{handler } c' \quad m' \quad (c \quad m \quad pc) \quad :: \quad C \quad e \quad (c \quad m \quad pc_{\text{catch}}) \quad :: \quad C' \\
\land \quad (\text{handler } c \quad m \quad C \quad e' \quad (c'' \quad m'' \quad pc''_{\text{catch}}) \quad :: \quad C') \\
\land \quad \neg(\quad e' = \quad e) \\
\Rightarrow \quad \text{handler } c' \quad m' \quad (c \quad m \quad pc) \quad :: \quad C \quad e' \quad (c'' \quad m'' \quad pc''_{\text{catch}}) \quad :: \quad C')
\end{align*}
\]
## Appendix C

### Instruction Generalization

<table>
<thead>
<tr>
<th>Op-code</th>
<th>Mnemonic / Syntax</th>
<th>Generalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>00...3e</td>
<td>nop</td>
<td>noop</td>
</tr>
<tr>
<td></td>
<td>move</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move/from16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move-wide</td>
<td>move&lt;sub&gt;r&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>move-wide/from16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move-wide/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move-object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move-object/from16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move-object/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move-result</td>
<td>move-result&lt;sub&gt;r&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>move-result-wide</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move-result-object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>move-exception</td>
<td>move-exception</td>
</tr>
<tr>
<td></td>
<td>return-void</td>
<td>return-void</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>return&lt;sub&gt;r&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>return-wide</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return-object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const/4</td>
<td>const&lt;sub&gt;r&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>const/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const vAA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const/high16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const-wide/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const-wide/32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const-wide</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const-wide/high16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const-string</td>
<td>const-string</td>
</tr>
<tr>
<td></td>
<td>const-string/jumbo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const-class</td>
<td>const-class</td>
</tr>
<tr>
<td>Instruction</td>
<td>Generalization</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------</td>
<td></td>
</tr>
<tr>
<td>check-cast</td>
<td>check-cast, check-cast-array</td>
<td></td>
</tr>
<tr>
<td>instance-of</td>
<td>instance-of, instance-of-array</td>
<td></td>
</tr>
<tr>
<td>array-length</td>
<td>array-length</td>
<td></td>
</tr>
<tr>
<td>new-instance</td>
<td>new-instance</td>
<td></td>
</tr>
<tr>
<td>new-array</td>
<td>new-array</td>
<td></td>
</tr>
<tr>
<td>filled-new-array</td>
<td>filled-new-array</td>
<td></td>
</tr>
<tr>
<td>filled-new-array/range</td>
<td>filled-new-array/range</td>
<td></td>
</tr>
<tr>
<td>fill-array-data</td>
<td>fill-array-data</td>
<td></td>
</tr>
<tr>
<td>throw</td>
<td>throw</td>
<td></td>
</tr>
<tr>
<td>goto</td>
<td>goto</td>
<td></td>
</tr>
<tr>
<td>goto/16</td>
<td>goto</td>
<td></td>
</tr>
<tr>
<td>goto/32</td>
<td>goto</td>
<td></td>
</tr>
<tr>
<td>packed-switch</td>
<td>packed-switch</td>
<td></td>
</tr>
<tr>
<td>sparse-switch</td>
<td>sparse-switch</td>
<td></td>
</tr>
<tr>
<td>cmpl-float</td>
<td>cmpl-float $</td>
<td>bias$</td>
</tr>
<tr>
<td>cmpg-float</td>
<td>$cmp_{r_c}$</td>
<td></td>
</tr>
<tr>
<td>cmpl-double</td>
<td>$cmp_{r_c}$</td>
<td></td>
</tr>
<tr>
<td>cmpg-double</td>
<td>$cmp_{r_c}$</td>
<td></td>
</tr>
<tr>
<td>cmp-long</td>
<td>$cmp_{r_c}$</td>
<td></td>
</tr>
<tr>
<td>if-eq</td>
<td>if-eq</td>
<td></td>
</tr>
<tr>
<td>if-ne</td>
<td>if-ne</td>
<td></td>
</tr>
<tr>
<td>if-lt</td>
<td>if-lt</td>
<td></td>
</tr>
<tr>
<td>if-ge</td>
<td>if-ge</td>
<td></td>
</tr>
<tr>
<td>if-gt</td>
<td>if-gt</td>
<td></td>
</tr>
<tr>
<td>if-le</td>
<td>if-le</td>
<td></td>
</tr>
<tr>
<td>if-eqz</td>
<td>if-eqz</td>
<td></td>
</tr>
<tr>
<td>if-nez</td>
<td>if-nez</td>
<td></td>
</tr>
<tr>
<td>if-ltz</td>
<td>if-ltz</td>
<td></td>
</tr>
<tr>
<td>if-gez</td>
<td>if-gez</td>
<td></td>
</tr>
<tr>
<td>if-gtz</td>
<td>if-gtz</td>
<td></td>
</tr>
<tr>
<td>if-lez</td>
<td>if-lez</td>
<td></td>
</tr>
<tr>
<td>ifz</td>
<td>ifz</td>
<td></td>
</tr>
<tr>
<td>3e ... 43 not used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44 ... 72</td>
<td>44 ... 72</td>
<td></td>
</tr>
<tr>
<td>aget</td>
<td>aget $r$</td>
<td></td>
</tr>
<tr>
<td>aget-wide</td>
<td>aget $r$</td>
<td></td>
</tr>
<tr>
<td>aget-object</td>
<td>aget $r$</td>
<td></td>
</tr>
<tr>
<td>aget-boolean</td>
<td>aget $r$</td>
<td></td>
</tr>
<tr>
<td>aget-byte</td>
<td>aget $r$</td>
<td></td>
</tr>
<tr>
<td>aget-char</td>
<td>aget $r$</td>
<td></td>
</tr>
<tr>
<td>aget-short</td>
<td>aget $r$</td>
<td></td>
</tr>
<tr>
<td>aput</td>
<td>aput $r$</td>
<td></td>
</tr>
<tr>
<td>aput-wide</td>
<td>aput $r$</td>
<td></td>
</tr>
<tr>
<td>aput-object</td>
<td>aput $r$</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>( \text{aput}_r )</td>
<td>( \text{iget}_r )</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>aput-boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aput-byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aput-char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aput-short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iget</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iget-wide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iget-object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iget-boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iget-byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iget-char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iget-short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iput</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iput-wide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iput-object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iput-boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iput-byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iput-char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iput-short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sget</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sget-wide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sget-object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sget-boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sget-byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sget-char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sget-short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sput</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sput-wide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sput-object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sput-boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sput-byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sput-char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sput-short</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoke-virtual</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoke-super</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoke-direct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoke-static</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoke-interface</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

73 not used

<p>| 74...78              |                     |                     |                     |                     |                     |                             |
| invoke-virtual/range  |                     |                     |                     |                     |                     | invoke-ref                 |
| invoke-super/range    |                     |                     |                     |                     |                     | invoke-ref                 |
| invoke-direct/range   |                     |                     |                     |                     |                     |                             |
| invoke-static/range   |                     |                     |                     |                     |                     | invoke                      |
| invoke-interface/range|                     |                     |                     |                     |                     | invoke-ref                 |</p>
<table>
<thead>
<tr>
<th>Instruction Generalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>7b ... e2</td>
</tr>
<tr>
<td>neg-int</td>
</tr>
<tr>
<td>not-int</td>
</tr>
<tr>
<td>neg-long</td>
</tr>
<tr>
<td>not-long</td>
</tr>
<tr>
<td>neg-float</td>
</tr>
<tr>
<td>neg-double</td>
</tr>
<tr>
<td>int-to-long</td>
</tr>
<tr>
<td>int-to-float</td>
</tr>
<tr>
<td>int-to-double</td>
</tr>
<tr>
<td>long-to-int</td>
</tr>
<tr>
<td>long-to-float</td>
</tr>
<tr>
<td>long-to-double</td>
</tr>
<tr>
<td>float-to-int</td>
</tr>
<tr>
<td>float-to-long</td>
</tr>
<tr>
<td>float-to-double</td>
</tr>
<tr>
<td>double-to-int</td>
</tr>
<tr>
<td>double-to-long</td>
</tr>
<tr>
<td>double-to-float</td>
</tr>
<tr>
<td>int-to-byte</td>
</tr>
<tr>
<td>int-to-char</td>
</tr>
<tr>
<td>int-to-short</td>
</tr>
<tr>
<td>add-int</td>
</tr>
<tr>
<td>sub-int</td>
</tr>
<tr>
<td>mul-int</td>
</tr>
<tr>
<td>div-int</td>
</tr>
<tr>
<td>rem-int</td>
</tr>
<tr>
<td>and-int</td>
</tr>
<tr>
<td>or-int</td>
</tr>
<tr>
<td>xor-int</td>
</tr>
<tr>
<td>shl-int</td>
</tr>
<tr>
<td>shr-int</td>
</tr>
<tr>
<td>ushr-int</td>
</tr>
<tr>
<td>add-long</td>
</tr>
<tr>
<td>sub-long</td>
</tr>
<tr>
<td>mul-long</td>
</tr>
<tr>
<td>div-long</td>
</tr>
<tr>
<td>rem-long</td>
</tr>
<tr>
<td>and-long</td>
</tr>
<tr>
<td>or-long</td>
</tr>
<tr>
<td>xor-long</td>
</tr>
<tr>
<td>shl-long</td>
</tr>
</tbody>
</table>

79 ... 7a not used
<table>
<thead>
<tr>
<th>shr-long</th>
<th>ushr-long</th>
<th>add-float</th>
<th>sub-float</th>
<th>mul-float</th>
<th>div-float</th>
<th>rem-float</th>
<th>add-double</th>
<th>sub-double</th>
<th>mul-double</th>
<th>div-double</th>
<th>rem-double</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>binop₈ ⊕</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add-int/2addr</td>
<td>sub-int/2addr</td>
<td>mul-int/2addr</td>
<td>div-int/2addr</td>
<td>rem-int/2addr</td>
<td>and-int/2addr</td>
<td>or-int/2addr</td>
<td>xor-int/2addr</td>
<td>shl-int/2addr</td>
<td>shr-int/2 addr</td>
<td>ushr-int/2addr</td>
<td>add-long/2addr</td>
</tr>
</tbody>
</table>
### Table C.1: Generalization of Dalvik opcodes

<table>
<thead>
<tr>
<th>rem-double/2addr</th>
<th>binop-2 b ⊕</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-int/lit16</td>
<td></td>
</tr>
<tr>
<td>rsub-int/lit16</td>
<td></td>
</tr>
<tr>
<td>mul-int/lit16</td>
<td></td>
</tr>
<tr>
<td>div-int/lit16</td>
<td></td>
</tr>
<tr>
<td>rem-int/lit16</td>
<td></td>
</tr>
<tr>
<td>and-int/lit16</td>
<td></td>
</tr>
<tr>
<td>or-int/lit16</td>
<td></td>
</tr>
<tr>
<td>xor-int/lit16</td>
<td></td>
</tr>
<tr>
<td>add-int/lit8</td>
<td>binop-lit b ⊕</td>
</tr>
<tr>
<td>rsub-int/lit8</td>
<td></td>
</tr>
<tr>
<td>mul-int/lit8</td>
<td></td>
</tr>
<tr>
<td>div-int/lit8</td>
<td></td>
</tr>
<tr>
<td>rem-int/lit8</td>
<td></td>
</tr>
<tr>
<td>and-int/lit8</td>
<td></td>
</tr>
<tr>
<td>or-int/lit8</td>
<td></td>
</tr>
<tr>
<td>xor-int/lit8</td>
<td></td>
</tr>
<tr>
<td>shl-int/lit8</td>
<td></td>
</tr>
<tr>
<td>shr-int/lit8</td>
<td></td>
</tr>
<tr>
<td>ushr-int/lit8</td>
<td></td>
</tr>
</tbody>
</table>

e3...ff not used

### Table C.2 Operators

| ⊕ τ ::= + | − | × | ÷ | mod | AND | OR | XOR | ≪ | ≫ | Binary operators |
| ⊙ τ ::= − | ∼ | τ | Unary operators |
| ⊙ ::= < | > | ≤ | ≥ | = | ≠ | Comparison operators |
### Table C.3 Definition of cmp function

\[
\text{cmp}_{\tau, \mid \text{bias}}(\text{val}_r, \text{val}'_r) = \begin{cases} 
0 & : \text{val}_r = \text{val}'_r \\
1 & : \text{val}_r < \mid \text{bias} \mid \text{val}'_r \\
-1 & : \text{val}_r > \mid \text{bias} \mid \text{val}'_r 
\end{cases}
\]

\(\text{bias} ::= \geq, \leq\)  
Determines how to treat NaN values
Appendix D

Algorithms

Algorithm D.1 Application data extraction

Precondition: \( \text{paths}[] \)

1: \textbf{procedure} \textsc{ExtractApp}(\text{paths}[])  
2: \hspace{1em} \textit{app}_n \leftarrow 0  
3: \hspace{1em} \textbf{for path in} \text{paths}[] \textbf{do}  
4: \hspace{2em} \text{apps} \leftarrow \text{apps} \oplus \text{App} (\text{app}_n, \text{name}, \ldots) \triangleright \oplus: \text{add to array; name} \text{ is specified in path}  
5: \hspace{1em} \textsc{ExtractAppData}(\text{app}_n, \text{appfiles}) \triangleright \text{appfiles} \text{ are taken from the folder in path}  
6: \hspace{1em} \textit{app}_n \leftarrow \textit{app}_n + 1  

1: \textbf{procedure} \textsc{ExtractAppData}(\textit{app}_n, \text{appfiles})  
2: \hspace{1em} \textbf{for file in} \text{appfiles} \textbf{do}  
3: \hspace{2em} \text{classes} \leftarrow \text{classes} \oplus \text{Class} (\textit{class}_n, \textit{app}_n, \ldots)  
4: \hspace{1em} \textbf{for class in} \text{classes} \textbf{do}  
5: \hspace{2em} \text{fields} \leftarrow \text{fields} \oplus \text{Field} (\textit{field}_n, \textit{class}_n, \ldots)  
6: \hspace{2em} \text{methodsSig} \leftarrow \text{methods} \oplus \text{MethodSignature} (\textit{methodSig}_n, \textit{class}_n, \ldots)  
7: \hspace{2em} \text{methodsDec} \leftarrow \text{methods} \oplus \text{MethodDeclaration} (\textit{methodDec}_n, \textit{class}_n, \ldots)  
8: \hspace{2em} \text{methods} \leftarrow \text{methods} \oplus \text{Method} (\textit{method}_n, \textit{class}_n, \ldots)  
9: \hspace{1em} \textbf{for method in} \text{methods} \textbf{do}  
10: \hspace{2em} \text{instruction} \leftarrow \text{instructions} \oplus \text{Instruction} (\textit{pc}_n, \textit{method}_n, \ldots)  
11: \hspace{2em} \ldots \text{(extracting information about array and switch instructions)}  

Postcondition: \text{App data collected in} \text{apps}[], \text{classes}[], \ldots
Algorithm D.2 Entry points

Precondition: methods[], entryPoints[]

1: procedure CreateEntryPointsCalls(methods[], entryPoints[])
2:   i ← LEN(methods[]) + 1 \(\triangleright\) add new method i to the existing ones
3:   methodsSig ← methods \(\oplus\) MethodSignature(methodSig_i, android, ...)
4:   methodsDec ← methods \(\oplus\) MethodDeclaration(methodDec_i, android, ...)
5:   methods ← methods \(\oplus\) Method(method_i, android, ...)
6:   for method in methods[] do
7:     if method.name in entryPoints then
8:       if method.kind == virtual then
9:         methods[i].instructions[] ← methods[i].instructions[] \(\oplus\)
10:          Instruction('invoke − virtual', method.argTypes[], ...) \(\triangleright\) add a call to the method i
11:     else
12:       if method.kind == direct then
13:         ...
14:     else
15:       if method.kind == static then
16:         ...

Algorithm D.3 Classes hierarchy

Precondition: Analysis, classes[]

1: procedure CreateRelationSuper(superClass_n, class_n)
2:   if super in Analysis.addedRelations then
3:     Analysis.code ← Analysis.code \(\oplus\) "(rule (super superClass_n class_n))"
4:   else
5:     Analysis.addedRelations ← Analysis.addedRelations \(\oplus\) super
6:     Analysis.code ← Analysis.code \(\oplus\) "(declare-rel super (Int Int))"
7:     Analysis.code ← Analysis.code \(\oplus\) "(rule (super superClass_n class_n))" \(\triangleright\) Facts are rules without => in Z3
8:     Analysis.code ← Analysis.code \(\oplus\) "(rule (=> (and (super x y) (super y z)) (super x z )))" \(\triangleright\) Transitive inheritance

1: procedure LogicClassesHierarchy(classes[])
2:   for class in classes[] do
3:     CreateRelationSuper(class.super, class.number) \(\triangleright\) some classes will have artificial bottom super class

Postcondition: relation super(superClass, class)
Algorithm D.4 Program initialization

Precondition: \( \text{Analysis, fields}[] \)

1: procedure CreateRelationS(fields[])
2:     for field in fields[] do
3:         if field.initValue <> null then
4:             Analysis.code \leftarrow \text{Analysis.code} + "(rule (s field.Class_n field.n field.initValue))"
5:         else
6:             Analysis.code \leftarrow \text{Analysis.code} + "(rule (s field.Class_n field.n Some-Sort()))"

Algorithm D.5 Generate instruction facts

Precondition: \( \text{Analysis, methods}[] \)

1: procedure CreateFactConst(instruction)
2:     m \leftarrow \text{instruction.arg}[1] \triangleright \text{method index}
3:     pc \leftarrow \text{instruction.arg}[2] \triangleright \text{program counter}
4:     regNumber \leftarrow \text{instruction.arg}[3] \triangleright \text{register we are writing to}
5:     value \leftarrow \text{instruction.arg}[4] \triangleright \text{value register will have after the operation}
6:     Analysis.code \leftarrow \text{Analysis.code} + "(rule (const m pc regNumber value))"

1: procedure GenerateInstructions(methods[])
2:     for method in methods[] do
3:         for instruction in method.instructions[] do
4:             if instruction.name == "const" then
5:                 if "const" not in Analysis.addedMethods[] then
6:                     Analysis.addedRelations \leftarrow Analysis.addedRelations \oplus instruction.name
7:                     Analysis.code \leftarrow Analysis.code + "(declare-rel const (Int Int Int Int))"
8:             end
9:         end
10:     end
11: else
12:     CreateFactConst(instruction)
13: end
Algorithm D.6 Method call

Precondition: Analysis, instructions[]

1: procedure CREATEFACTCALL(instruction, virtual, type) \(\triangleright\) virtual is true for resolution
2: \(m \leftarrow\) instruction.arg[1] \(\triangleright\) method index
3: \(pc \leftarrow\) instruction.arg[2] \(\triangleright\) program counter
4: \(\text{regNumber} \leftarrow\) instruction.arg[3] \(\triangleright\) class reference register
5: \(m_c \leftarrow\) instruction.arg[4] \(\triangleright\) method we are calling
6: \(\text{numLoc}_{m_c} \leftarrow m_c\.numLocals\) \(\triangleright\) number of local reg in \(m_c\)
7: \(\text{par[]} \leftarrow\) instruction.arg[5] \(\triangleright\) array with parameters
8: ADDINSTRUCTIONCALL\((m, pc, \text{regNumber}, m_c)\) \(\triangleright\) add fact invoke-type to the prover
9: for \(\text{par}[i]\) in \(\text{par}[]\) do \(\triangleright\) i determines order of parameters
10: if \(i == 1\) and (virtual or type = "direct") then
11: skip \(\triangleright\) first argument is a reference to the (resolved) class
12: else
13: CREATEFACTINVOKE\((m, pc, \text{par}[i], i, m_c)\) \(\triangleright\) add fact invoke to the prover