# Design and Analysis of Mobile Operating System Security Architecture using Formal Methods

**Hendra Gunadi**

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

November 2017

Except where otherwise indicated, this thesis is my own original work.

Hendra Gunadi
7 November 2017

to my mother, Hennywati Surya
to my brothers, Julius Halim and Himawan Hutomo
to my wife, Maria Kristiyanti Tjandradjaja

# Acknowledgments

First of all, I want to thank God for blessing me with such a great privilege that I may finish this thesis. That even though I always forget about Him (especially in the hectic times doing this thesis), He still loves me and care for me. I thank God for putting these people around me who helped me through the whole time of my Ph.D.

I also want to thank my family for their support throughout my degree. Especially to my mom, Hennywati Surya who has sacrificed so much to support me going to study here. And I want to thank my brothers, Julius and Himawan, as well, because even though we are far apart, they still accompany me (online).

One big thanks to my supervisor Alwen Tiu for helping me by directing me through this thesis. I really appreciate all the help and support that you have given me. I know that the whole situation is not really conducive since we have to do long distance correspondence, while you are also really busy with your own responsibility. You still patiently help me to work through this thesis (especially all the proof readings!).

Not also forgetting to thank my lovely wife, Maria Kristiyanti Tjandradjaja. Your involvement in my life has brought about significant changes. Thanks for all the happy and hard times together, and also for your support, making sure that I am motivated to finish this thesis.

I also want to say thanks to Rajeev Goré and Dirk Pattinson for their eagerness to help even though you are busy. Thanks Raj for helping me in administrative area and making sure that I progress throughout my Ph.D. Also thanks Dirk for your pointer and discussion in finishing the formalization in Coq.

I want to say thanks as well to all the people I know who have been involved in my life here during these 6 years in Canberra. For my Indonesian (in Indonesia) friends who keep on "annoy"- ing me, for my office mate and fellow Ph.D student Gary, for Micaiah and Elysia, and for my housemates John, Jing, and Kuangda. I also want to thanks to Simon and Annabel for catching up with me regularly, dinner gang that regularly meets every Friday, also for James for all the regularly catching up after church for quite a while.

Finally, I also want to thank the reviewers for their valuable feedback.

This research is supported by an Australian Research Training Program (RTP) Scholarship.

# Abstract

The Android operating system (OS) is now used in the majority of mobile devices. Hence, Android security is an important issue to handle. In this work, we tackle the problem using two separate approaches: directly modifying Android OS and developed a framework to provide a guarantee of non-interference.

Firstly, we present a design and an implementation of a security policy specification language based on metric linear-time temporal logic (MTL) to specify timing-dependent security policies. The design of the language is driven by the problem of runtime monitoring of applications in mobile devices. A main case of the study is the privilege escalation attack in the Android OS, where an unprivileged app gains access to privileged resource or functionalities through indirect flow. To capture these attacks, we extend MTL with recursive definitions to express call chains between apps. We then show how our language design can be used to specify policies to detect privilege escalation under various fine-grained constraints. We present a new algorithm for monitoring safety policies written in our specification language. The monitor does not need to store the entire history of events generated by the apps. We modified the Android OS kernel to allow us to insert our generated monitors modularly. We have tested the modified OS (LogicDroid) on an actual device, and show that it is effective in detecting policy violations. Furthermore, LogicDroid is able to prevent a previously unknown exploit to breach Android security which allows an unprivileged application to access certain critical and privileged functionalities of an Android phone, such as making phone calls, terminating phone calls, and sending SMS, without having to ask any permissions to do so.

Subsequently, we provided a framework to ensure non-interference properties of DEX bytecode. Each application in Android runs in an instance of the Dalvik virtual machine, which is a register-based virtual machine (VM). Most applications for Android are developed using Java, compiled to Java bytecode and further into DEX bytecode. Following a methodology that has been developed for Java bytecode certification by Barthe et al., we developed a type-based method for certifying non-interference property of a DEX program. To this end, we develop a formal operational semantics of the Dalvik VM, a type system for DEX bytecode, and prove the soundness of the type system with respect to a notion of non-interference. We have also formalized the proof of a subset of DEX in Coq for an additional guarantee that our proof is correct.

We then study the translation process from Java bytecode to DEX bytecode, as implemented in the dx tool in the Android SDK. We show that an abstracted version of the translation from Java bytecode to DEX bytecode preserves the non-interference property. More precisely, we show that if the Java bytecode is typable in Barthe et al.'s type system, then its translation is typable in our type system. This result

opens up the possibility to leverage existing bytecode verifiers for Java to certify non-interference properties of Android bytecode.

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

## 1.1 Introduction

Android is a popular mobile operating system (OS) that has been used in a range of mobile devices such as smartphones and tablet computers. According to Sta [2017], Android has the most significant market share for mobile devices, making it an attractive target for malwares, so verification of the security properties of Android apps is crucial. It uses Linux as the kernel, which is extended with an application framework (middleware). Most applications of Android are written to run on top of this middleware, and most of the Android-specific security mechanisms are enforced at this level.

The Android OS is built on top of the Linux kernel, so at the most basic level, it inherits most of the security architecture of Unix/Linux. Figure 1.1 shows Android's architecture. Android treats each application as a distinct user with a unique user ID. At the kernel level, the standard Unix permission mechanism enforces access control based based on the user id (and group id) of the app. Since Android 4.3, Android also uses SELinux (McCarty [2004]) to enforce other security policies on top of standard Unix security.

To install an application, users can download applications from Google Play or third-party app stores in the form of an Android Application Package (APK). Android's applications, however, do not run directly on Linux. Rather they run inside a virtual machine called the Dalvik Virtual Machine (DVM), that are insulated from the rest of the system. This method of *sandboxing* an application is similar to the sandboxing implemented in the Java virtual machine but does not include certain security features, such as the Security Manager. All Android applications are running in their own instance of the Dalvik virtual machine, and communication and sharing between apps are allowed only through an inter-process communication (IPC) mechanism. Android middleware provides a list of resources and API to access specific functionalities of the device, such as making phone calls, sending an SMS, querying some GPS location information, or querying unique device ID (such as the IMEI number), etc. Inter-component communication (ICC) uses the same communication mechanism within the same application, so we shall use ICC to refer to both ICC and IPC.

Android enforces access control to device's functionalities via its permission mech-

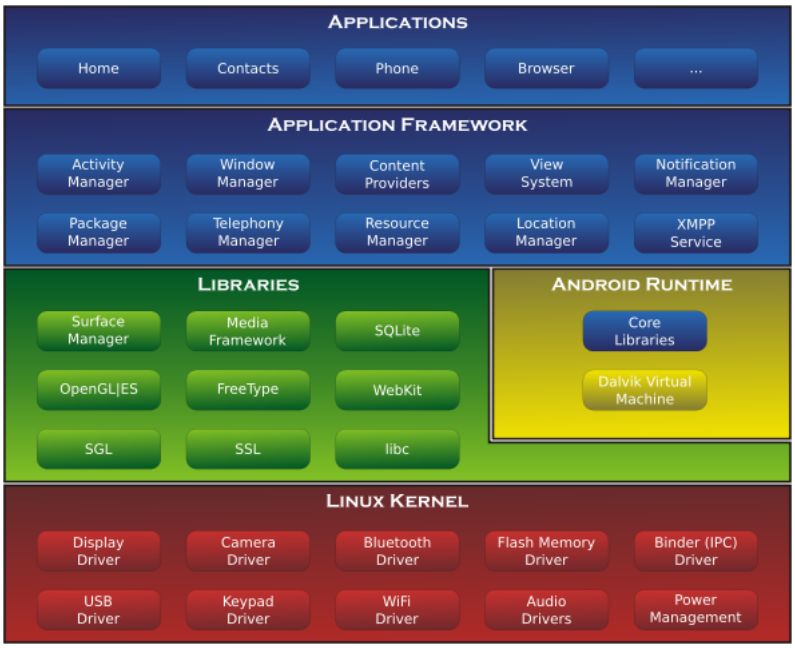



Figure 1.1: Android OS architecture. Image source: Wikipedia.

anism: each service/resource is associated with a certain unique permission tag, and each app must request permissions to the services it needs at installation time. For example, to be able to connect to the internet, an app in Android needs to have the INTERNET permission. Similarly, to access fine location information (such as that obtained through a GPS device), an app needs to have the ACCESS_FINE_LOCATION permission. For a list of permissions in Android, the reader is referred to the Android developer website.[1]. Every time app requests access to a specific service/resource, the Android's runtime security monitor checks whether the app has the required permission tags for that particular service/resource. This permission mechanism and the Linux security features Android inherits address the traditional operating system security issues. However, most of the interesting problems arise on the level of applications, and a separate security mechanism is required to deal with those. A more detailed discussion of the Android's security architecture can be found in Enck et al. [2009b].

## 1.2 Research Motivation and Direction

Most of the time, a permission on its own does not lead to a security concern. That said, security issues may arise the moment access to several sensitive resources are obtained. For example, if an app only has the capability of accessing SMS then users do not have to be worried about the loss of their privacy, but when it also has the capability of accessing the Internet, then users might be worried that the app leaks

[1] http://developer.android.com

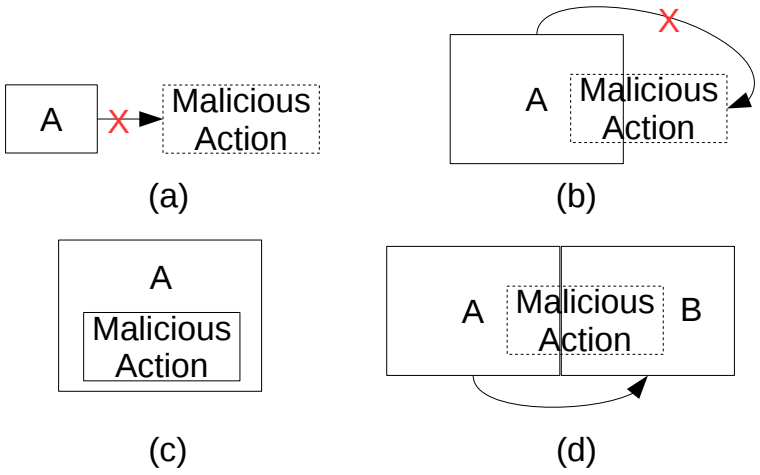



Figure 1.2: App and permissions required to do a malicious activity: (a) App is not granted any permission which can lead to a malicious activity (b) App is granted a subset of the permissions (c) App has a superset of the permissions (d) App is gaining more privilege

the content of their SMS online. Adding to the mix, an app may also gain access to sensitive resources even though it does not have permission to do so through other apps. In Android where reusability of a component is encouraged, an app (in particular its components) may expose its functionalities. It is expected that the app will make use of the Android's permission system to impose a guard on the functionalities it exposed. The problem is, this assumption does not always hold. Some apps do not impose permission checking on the caller, either intentionally (collusion attack) or unintentionally (confused deputy). This problem is known as privilege escalation, where an app gains more privilege than its granted permission.

See Figure 1.2. Ideally, Android should be able to prevent apps from doing any malicious activity, e.g., by only allowing an app to have a subset of permissions which can be misused to do a malicious activity (b), or deny the permissions altogether (a). In practice, developers generally request more permissions than what the app actually need (c), and end users have to grant such request in order to use the app. Furthermore, even when an app does not have a full set of permissions to do a malicious activity, it can gain more privilege than its granted permission (Privilege Escalation) through colluding with other apps or by exploiting unguarded functionalities of other apps (confused deputy).

To give an example, assume that the malicious activity we are interested in is leaking the content of the SMS to the attacker somewhere on the Internet. In this case, we have two particular permission that will be required in order to mount the attack: permission to read SMS and permission to access the Internet. We go through possible scenarios where an app can / cannot access a sensitive resource or perform

malicious actions:

(a) The app does not have both read SMS and access Internet permissions, and it does not try to gain the restricted access, so it is not possible for the app to do the malicious activity.

(b) The app only has either read SMS permission or access the Internet permission. The app also does not try to gain access to the other permission. So, in this case, the app is safe in that it does not have the capability of executing the malicious activity.

(c) The app has some permissions including both permissions to read SMS and access the Internet. In this case, although it may not always be the case, there is a possibility for the app to perform the malicious activity.

(d) The app only has either read SMS permission or access the Internet permission. But in this particular case, the app is trying to gain access to the other resource that it does not have. In this case, the app may make use of the resource it obtained to do the malicious activity.

In both scenario (c) and (d), Android cannot prevent the app to do the malicious activity due to the nature of the permission mechanism. In (c), the app is allowed to access both resources and Android's permission is not fine grained enough to say that only one permission may be activated at a time, e.g., if an app already read the SMS, then it should not access the Internet for some period. Android also depends on the app to secure its own interface, so in scenario (d) where another app is exposing access to a sensitive resource, Android will only check that the other app has the permission to access it, not the app that takes advantage of this exposed functionalities.

A study by Enck [2011] reveals some of these problems, e.g., leaking of phone identifiers (which can be tied to personal identification information) to advertisement and analytics servers, leaking of location information to advertisement servers, leaking of private information via IPC and logs, and applications can unsafely delegate actions. We also note that there are times when developers expose functionalities without a proper guard, e.g., a vulnerability in the Android system which allows apps without proper permission to make a call (see Section 2.6.4).

We explore centralized approach where we modify Android OS to deal with the issue of accessing resources (see the next chapter). We implement a monitor that is placed on the resources' hook to help Android regulate access to resources. Although our solution is more general in that it is designed as an access control mechanism, we decided to focus on Privilege Escalation as our case study because it has practical significance and it is not currently enforced in Android framework. We designed a policy language which can capture transitive calls, which is the essence of handling privilege escalation, and show that our approach is able to detect and prevent such problem.

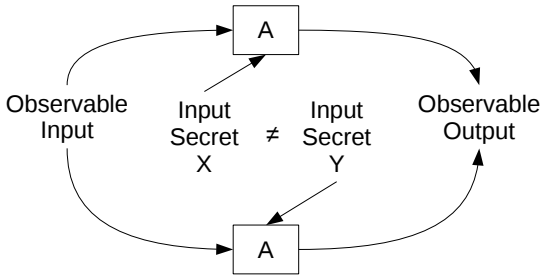



Figure 1.3: Non-Interferent Program

It is appealing to continue along the same vein to deal with the issue of apps leaking sensitive data, but we realize that there are many problems associated with centralized monitoring. To begin with, end users will not be able to adopt the centralized monitoring approach easily as it requires a modification to the OS. We also note that the modification has to be done for each version of Android, which is evolving rapidly. Hence it requires substantial effort to keep up with the latest version. Furthermore, a centralized approach to prevent leakage of data would require an active taint tracking, such as TaintDroid (Enck et al. [2014]), which incurs a heavy overhead.

Considering the shortcomings of the centralized approach, we switch our focus on the apps instead. In particular, we design a type system to ensure non-interference property for Android bytecode (DEX). A program is non-interferent when a secret / private input is not affecting (interfering with) its observable output, or to put it another way, a program is non-interferent when observable output only depends on observable inputs (See Figure 1.3). One clear advantage of this approach is that we target DEX bytecode directly, which is considerably constant through Android's evolution. Even with the introduction of the new runtime environment in Android (ART), our solution is still applicable since ART still uses DEX bytecode. The main bulk of this dissertation will be on this latter approach. We also provide implementations for both as proof of concepts.

The focus of this thesis, then, is to improve Android security using formal method approaches. The first approach is centralized monitoring using modified Metric Temporal Logic (MTL) as the policy language. The second approach is designing type system which enforces non-interference (inspired by the non-interferent type system for JVM bytecode described in Section 3.3) and proves that typable JVM bytecode is compiled (using non-optimizing dx compiler) into typable DEX bytecode. The two approaches are complementary and work together to improve the overall security of Android.

## 1.3 Thesis Outline

The next chapter outlines our attempt to tackle Android information flow security from the perspective of centralized monitoring. Section 2.3 introduces a policy lan-

guage RMTL, an extension (restriction) to Metric Temporal Logic (MTL). In Section 2.4, We present the monitoring algorithm for RMTL and state its correctness. Section 2.5 describes some example policies. Section 2.6 discusses our implementation of the monitors for RMTL, and the required modification of Android OS kernel to integrate our monitor into the OS. Details of the implementation of the monitor generator and the binaries of the modified Android OS, which constitutes LogicDroid, are available online. [2] We also give details of the exploit in the Android component `com.android.phone` that leads to privilege escalation, and how LogicDroid mitigates this exploit in Section 2.6.4.

The rest of the thesis outlines our attempt to tackle Android information flow security from the perspective of securing the apps. It is organized as follows: we start by providing the background of this approach in Chapter 3 which contains Barthe et al.'s non-interferent type system for JVM bytecode (Section 3.3). The following chapter outlines the non-interferent type system for DEX (Chapter 4) and shows in Section 4.2 that the DEX type system we proposed enforce non-interference, i.e., if a DEX program is typable then it is non-interferent. We formalize the proof of the type system soundness for a subset of DEX in Coq, and we give more detail on the formalization in Chapter 5

Chapter 6 describes the non-optimized compilation of the dx tool (Section 6.1) and the proof of typability preserving compilation of non-optimized dx tool (Section 6.2). We also give examples to demonstrate how our methodology is able to detect interference by the failure of typability in Section 4.1.4.

The next section (Section 6.3) provides an overview of the certification framework while Section 6.3.1 details each component in the architecture: JVM naive type inference, DEX type checker, certificate translation and certificate structure.

The dissertation chapters are based upon the following published / under submission papers:

- Gunadi, H and Tiu, A. Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System. In International Symposium on Formal Methods, pp. 296-311. Springer International Publishing, 2014.

- Gunadi, H; Tiu, A; and Gore, R. . Formal certification of android bytecode. arXiv preprint arXiv:1504.01842.

A note about the centralized monitoring is that the initial part of LogicDroid implementation was done in my Master's thesis. Since then it has been improved in many ways. For example, we streamlined the theory, give correctness proof of the monitoring algorithm, and we significantly revise the design and the implementation of the monitoring framework due to the problem of implementing hook of the socket, which must be done at the Linux kernel level. As a result of this design improvement, LogicDroid becomes practically usable.

[2] http://users.cecs.anu.edu.au/~hengunadi/LogicDroid.html.

Chapter 2

# Efficient Runtime Monitoring with Metric Temporal Logic

Despite all the security mechanisms mentioned in the introduction, Android is still vulnerable to various attacks. We shall focus mostly on attacks that target the permission mechanism at the application level. A weakness in the Android permission mechanism lies in the so-called *permission leakage* problem described in Grace et al. [2012]. In Android, an app can provide a "service" to another app. Through this provision of services, an app can "leak" certain capabilities to an unprivileged app. For example, an app which has access to the ability to make phone calls could act as a proxy for another app. This leads to the problem of *privilege escalation*, i.e., an app obtains a permission it was not granted by exploiting other apps.

Obviously privilege escalation is a common problem of every OS, e.g., when a kernel bug is exploited to gain root access. However, in Android, privilege escalation is possible even when apps are running in the confines of Android sandboxes (see Lineberry et al. [2010]; Davi et al. [2011]; Bugiel et al. [2012]). According to Bugiel et al. [2012], there are two types of attacks that can lead to privilege escalation: the *confused deputy attack* and the *collusion attack*. In the confused deputy attack, a legitimate app (the deputy) has permissions to certain services, e.g., sending SMS, and exposes an interface to this functionality without any guards. This interface can then be exploited by a malicious app to send SMS, even though the malicious app does not have the permission. Recent studies, e.g., Lineberry et al. [2010]; Grace et al. [2012]; Chan et al. [2012], show some system and consumer apps expose critical functionalities that can be exploited to launch confused deputy attacks. The collusion attack requires two or more malicious apps to collaborate. We have yet to encounter such malware, either in the Google Play market or the third party markets, although a proof-of-concept malware with such properties, called SoundComber (Schlegel et al. [2011]), has been constructed.

A recent study by Grace et al. [2012] on major brands of Android phones shows that there are some built-in apps that ship with the phones that expose critical interfaces without any permission enforcements that can be exploited to perform privilege escalation attack. Even assuming that the app manages to guard all these interface adequately, there still exist simple exploits that cannot be easily fixed, simply because

it will affect users experience significantly. For example, by default, in Android any app can launch the default browser without requiring any permission. With some work, a malicious app can be developed to launch a browser undetected (e.g., when the device screen is off) to leak sensitive data, as demonstrated in Lineberry et al. [2010].

Several security extensions to Android have been proposed to deal with privilege escalation attacks (see Dietz et al. [2011]; Felt et al. [2011]; Bugiel et al. [2012]). Unlike these works, we aim at designing a high-level policy language that is expressive enough to capture privilege escalation attacks but is also able to express more refined policies (see Section 2.5). Moreover, we aim at designing a lightweight monitoring framework, where policy specifications can be modified easily and enforced efficiently. Thus we aim to automate the generation of security monitors from a given policy that can efficiently enforce input policies written in our specification language.

On the specific problem of detecting privilege escalation, it is essentially a problem of tracking (runtime) control flow, which is, in general, a difficult problem and would require a certain amount of static analysis, e.g., Denning and Denning [1977]; Enck et al. [2010]. So we adopt a 'lightweight' heuristic to ascertain causal dependency between ICC calls: we consider two successive calls, say from A to B, followed by a call from B to C, as causally dependent if they happen within a certain reasonably short time frame. This heuristic can be easily circumvented if B is a colluding app. So the assumption that we make here is that B is honest, i.e., the confused deputy. For example, a privilege escalation attack mentioned in Lineberry et al. [2010] involves a malicious app, with no permission to access the internet, using the built-in browser (the deputy) to communicate with a server. In our model, the actual connection (i.e., the network socket) is treated as a virtual app, so the browser here acts as a deputy that calls (opens) the network socket on behalf of the malicious app. In such a scenario, it is reasonable to expect that the honest deputy would not intentionally delay the opening of sockets. So our heuristic seems sensible in the presence of confused deputy attacks but can be of course circumvented by colluding apps (collusion attacks). There is probably no general solution to detect collusion attacks that can be effective in all cases, e.g., when covert channels are involved (Schlegel et al. [2011]), so we shall restrict to addressing the confused deputy attacks.

The core of our policy language, called RMTL, is essentially a past-fragment of metric linear temporal logic (MTL) (see Alur and Henzinger [1990]; Thati and Rosu [2005]; Basin et al. [2008]). We consider only the fragment of MTL with past-time operators, as this is sufficient for our purpose to enforce history-sensitive access control. This also means that we can only enforce some, but not all, safety properties specified in Lichtenstein et al. [1985], e.g., policies capturing obligations as in, e.g., Basin et al. [2008], cannot be enforced in our framework. Temporal operators are useful in this setting to enforce access control on apps based on histories of their executions; see Section 2.5. Such a history-dependent policy cannot be expressed in the policy languages used in Dietz et al. [2011]; Felt et al. [2011]; Bugiel et al. [2012].

MTL by itself is, however, insufficient to express transitive closures of relations, which is needed to specify ICC call chains between apps, among others. To deal

with this, we extend MTL with recursive definitions, e.g., one would be able to write a definition such as:

$$trans(x,y) \coloneqq call(x,y) \vee \exists z.\ \diamondsuit_n\, trans(x,z) \wedge call(z,y), \tag{2.1}$$

where *call* denotes the ICC event, and $x, y, z$ denote the apps. This equation defines *trans* as the transitive closure of *call*. The metric operator $\diamondsuit_n \phi$ means intuitively $\phi$ holds within $n$ time units in the past; we shall see a more precise definition of the operators in Section 2.3. Readers familiar with modal $\mu$-calculus Bradfield and Stirling [2007] will note that this is but a syntactic sugar for $\mu$-expressions for (least) fixed points.

To be practically enforceable in Android, the RMTL monitoring algorithm must satisfy an important constraint, i.e., the algorithm must be *trace-length independent* (Bauer et al. [2013]). This is because the number of events generated by Android can range in the thousands per hour, so if the monitor must keep all the events generated by Android, its performance will degrade significantly over time. Another practical consideration also motivates a restriction to metric operators that we adopt in RMTL. More specifically, MTL allows a metric version of the 'since' operator of the form $\phi_1\ \mathsf{S}_{[m,n)}\ \phi_2$, where $[m,n)$ specifies a half-closed (discrete) time interval from $m$ to $n$. The monitoring algorithm for MTL in Thati and Rosu [2005] works by first expanding this formula into formulas of the form $\phi_1\ \mathsf{S}_{[m',n')}\ \phi_2$ where $[m',n')$ is a moving window of the interval (with the minimum value of 0). A similar expansion is also used implicitly in monitoring for first-order MTL in Basin et al. [2008], i.e., in their incremental automatic structure extension in their first-order logic encoding for the 'since' and 'until' operators. In general, if we have $k$ nested occurrences of metric operators, each with interval $[m,n)$, the number of formulas produced by this expansion is bounded by $O(n^k)$ In Android, event timestamps are in milliseconds, so this kind of expansion is not practically feasible. For example, suppose we have a policy that monitors three successive ICC calls that happen within 10 seconds between successive calls. This requires two nested metric operators with intervals $[0, 10^4)$ to specify. The above naive expansion would produce around $10^8$ formulas, and assuming the truth value of each formula is represented with 1 bit, this would require around 100 MB of storage to store all their truth values, something which is not preferable in the setting of smartphones where storage is limited.

An improvement to the expansion mentioned above is proposed in Basin et al. [2012]; Reinbacher et al. [2013], where one keeps a sequence of timestamps for each metric temporal operator occurring in the policy. This solution, although avoids the exponential expansion, is strictly speaking not trace-length independent. This solution seems optimal, so it is hard to improve it without further restriction to the policy language. We show that, if one restricts the intervals of metric operators to the form $[0,n)$, one only needs to keep one timestamp for each metric operator in monitoring; see Section 2.4.

To summarise, our contributions are as follows:

1. Regarding results in runtime verification, our contribution is in the design of

a new logic-based policy language that extends MTL with recursive definitions, that avoids exponential expansion of metric operators, and for which the policy enforcement is trace-length independent. In Bauer et al. [2013], a policy language based on first-order LTL and a general monitoring algorithm are given, but they do not allow recursive definitions nor metric operators. Such definitions and operators could perhaps be encoded using first-order constructs (e.g., encoding recursion via Horn clauses, and define timestamps explicitly as a predicate), but the resulting monitoring procedure is not guaranteed to be trace-length independent.

2. Regarding the application domain, ours is the first implementation of a logic-based runtime security monitor for Android that can enforce history-based access control policies, including those that concern privilege escalations. Our monitoring framework can express temporal and metric-based policies not possible in existing works Dietz et al. [2011]; Felt et al. [2011]; Bugiel et al. [2012].

## 2.1 Related Work

There is a large body of works in this area, more than what we can reasonably survey here, so we shall focus on the most relevant ones to our work, i.e., those that deal with privilege escalation. For a more comprehensive survey on other security extensions or analysis, the interested reader can consult Backes et al. [2014]. QUIRE, by Dietz et al. [2011], is an application-centric approach to privilege escalation, done by tagging the intent objects with the caller's UID. Thus, the recipient application can check the permission of the source of the call chain. Felt et al. [2011] propose IPC Inspection, another application-centric solution that works by reducing the privilege of the recipient application when it receives a communication from a less privileged application.

At the time of the development of our solution, the closest work to our solution is XManDroid Bugiel et al. [2012], which is also a system-centric solution just like ours. Its security monitor maintains a call graph between apps. The difference in our solution lies in that we are using temporal logic to specify a policy, and our policy can be modified modularly. This way, a system administrator can have flexibility in designing a policy that is suited to the system in question. Moreover, should an attacker find a way to circumvent the current monitor, we can easily modify the monitor to enforce a different policy that addresses the security hole.

Our policy language is also more expressive than XManDroid, as we can specify both temporal and metric properties. As a result, XManDroid will have better performance in general (exploiting the persistent link in the graph by using cache), yet there are policies that our monitor can enforce but XManDroid cannot. For example, consider Policy 4 in Section 2.5. XManDroid can only express whether an application has the permission to access contact database, but not the fact that contact database was accessed in the past. So, in this case, XManDroid would forbid an app with permission to access contact to connect to the internet, whereas in our case, we prevent

the connection to the internet only after contact was actually accessed.

TaintDroid Enck et al. [2010] is another system-centric solution, but it is designed to track data flow, rather than control flow, via taint analysis. In this case, we can infer privilege escalation happened from leakage of data.

Since then, there are many solutions developed such as Scippa (Backes et al. [2014]) and IacDroid (Zhang et al. [2016]) which deal with the problem of privilege escalation by maintaining call chain through an extension of the binder mechanism. IEM is also an interesting solution (Yagemann and Du [2016]) although it is not dealing with the problem of privilege escalation directly. Instead, they propose a framework to intercept intents and allows a user space application to specify their own policy, which can be geared toward handling privilege escalation.

Apart from work on monitoring M(FO)TL by Thati and Rosu [2005]; Basin et al. [2008, 2012]; Reinbacher et al. [2013] mentioned in the introduction, it is also worth mentioning prècis (a work by Chowdhury et al. [2014]). In their work, they are considering both past and future fragment of the MFOTL. They first analyze buildable formulas (B formulas) and then incrementally build the summary structures for B formulas. In the case where the formula is not a B formula, it then recursively calls on the sub-formulas and computes the substitutions brute force. Even though this means that prècis has a great capability, we do not really benefit much since we are only considering past fragment of the MTL. Moreover, by considering both intervals for building a summary structure, it is not necessarily trace-length independent, one of the main criteria that we require.

## 2.2 Background

**Linear Temporal Logic**

LTL (Blackburn et al. [2007]) is a logic defined in Kripke semantics where each event corresponds to a world. We first need to define LTL formally because we want to use its interpretation of infinite event traces. LTL extends propositional logic with temporal operators. Since we are only interested in the past call events, we only consider the fragment of LTL with past operators. The set of LTL formulas is defined as follows as follows :

$$\phi \;=\; \top \mid \bot \mid p \in P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \bullet\phi \mid \diamond^{-1}\phi \mid \phi_1 S \phi_2 \mid \diamondsuit^{-1}\phi$$

We do not have $\square^{-1}\phi$ here as $\square^{-1}\phi = \neg \diamond^{-1} \neg\phi$

We view history of calls a sequence of worlds in Kripke semantics. We denote that the $1^{st}$ world corresponds to the $1^{st}$ event in the history, $2^{nd}$ world corresponds to the $2^{nd}$ event, and so on. We denote this world as a pair $(h, i)$ where $h$ is the whole history of the events, and $i$ is the index of an event from the history. For simplicity, we can denote this pair $(h, i)$ in a shorter notation as $h_i$. Index 1 ($h_1$) is used to denote the initial world where the history is empty.

The symbol '$\models$' is used to denote that certain predicate or formula holds in that particular world. We distinguish between event predicates $P$ and static predicates $S$,

where we define an event predicate as a predicate whose interpretation may differ for each world, and we define a static predicate as a predicate whose interpretation is fixed. We need this static predicate to express predicates that persists in all the worlds, for example, a predicate that indicates whether an application is a system application.

The semantics of the operators are the following :

- $(h,i) \vDash true$ is always true
- $(h,i) \vDash false$ is always false
- $(h,i) \vDash p$, where $p \in P$, iff $p \in h_i$
- $(h,i) \vDash s$, where $s \in S$, iff $s$ is true
- $(h,i) \vDash \neg\phi$ iff $(h,i) \nvDash \phi$
- $(h,i) \vDash \phi_1 \wedge \phi_2$ iff $(h,i) \vDash \phi_1$ and $(h,i) \vDash \phi_2$
- $(h,i) \vDash \phi_1 \vee \phi_2$ iff $(h,i) \vDash \phi_1$ or $(h,i) \vDash \phi_2$
- $(h,i) \vDash \bullet\phi$ iff for $i > 1, (h,i-1) \vDash \phi$ otherwise false
- $(h,i) \vDash \diamond^{-1}\phi$ iff $\exists_{j \leq i}(h,j) \vDash \phi$
- $(h,i) \vDash \circledcirc^{-1}\phi$ iff $(h,i) \vDash \bullet \diamond^{-1} \phi$
- $(h,i) \vDash \phi_1 S \phi_2$ iff $\exists_{j \leq i}(h,j) \vDash \phi_2$, and $(h,k) \vDash \phi_1$ for all $j < k \leq i$

The problem of monitoring in our case is essentially a word problem, where we want to check whether a given word contains a "bad" prefix. Here, word corresponds to the history of events. We adopt the approach where we incrementally process the event as they come. More formally, we define the word problem as checking whether for any history $(h,i)$ and an LTL formula $\Phi$, $(h,i) \vDash \Phi$. Once a "bad" prefix is detected, no matter how the events unfold the property will always be violated. Hence, once an action which causes "bad" prefix is detected, we preempt the execution of such action.

**First-Order Extension** This extends the LTL described before with first-order quantifiers and predicates (see Bauer et al. [2009]; Basin et al. [2010]). In this case, we have a domain $D$ (in this case a list of all applications), list of constants $C$, list of predicates $R$ with its arity list $A$, and variable $V$, and the universal and existential operators. A constant can be seen as a predicate with zero arity. This extension enables us to state a policy where it applies to all the applications. The semantics of this extension is as follows :

- $(h,i) \vDash p(c_1,...,c_n) \in P$ iff $c_1,...,c_n \in D$ and $p(c_1,...,c_n) \in h_i$
- $(h,i) \vDash \forall_{x_1,...,x_n}\phi$ iff for all $c_1,...,c_n \in D$, $(h,i) \vDash \phi[x_1 := c_1,...,x_n := c_n]$
- $(h,i) \vDash \exists_{x_1,...,x_n}\phi$ iff there exists $c_1,...,c_n \in D$, $(h,i) \vDash \phi[x_1 := c_1,...,x_n := c_n]$

Note that the notation $\phi[x_1 := c_1, ..., x_n := c_n]$ means that for every occurrence of variable $x_1, ..., x_n$, they were replaced with the corresponding object $c_1, ..., c_n \in D$

Since we assume that the domain is finite, we can enumerate the quantifiers. That is, we instantiate the variable as the ground truth and treat the formula as a propositional logic formula. Then we will have the existential quantifier as a big disjunction, while the universal quantifier as a big conjunction, over the subformula where every occurrence of the variable is replaced with the object from the domain.

**Metric Extension** The extension of MTL over LTL is that we give each world a time stamp ($\tau_i$ = time stamp at world $i$) of when it occurred, and add the notation of interval accompanying the temporal operators. The time stamp related to each of the worlds has the property of $\tau_i \leq \tau_j$ whenever $i \leq j$ (events in history are sorted according to the time it happened). The form of the interval is $[m, n)$ where $m, n$ are natural numbers with $m < n$. $m$ is the lower bound, and $n$ is the upper bound of a formula to be still included in the evaluation (the difference between the time stamp of the worlds). The original LTL is a special case of MTL where the temporal operators have $m = 0$ and $n = \infty$.

More formally, the semantics of this metric extension are the followings :

- $(h, i) \vDash \bullet_{[m,n)}\phi$ iff for $i \geq 1$, $(h, i-1) \vDash \phi$ and $m \leq \tau_i - \tau_{i-1} < n$, otherwise false
- $(h, i) \vDash \diamond^{-1}_{[m,n)}\phi$ iff $\exists_{j \leq i}(h, j) \vDash \phi$ and $m \leq \tau_i - \tau_j < n$
- $(h, i) \vDash \diamondsuit^{-1}_{[m,n)}\phi$ iff $\exists_{j < i}(h, j) \vDash \phi$ and $m \leq \tau_i - \tau_j < n$
- $(h, i) \vDash \phi_1 S_{[m,n)}\phi_2$ iff $\exists_{j \leq i}(h, j) \vDash \phi_2$ and $m \leq \tau_i - \tau_j < n$, and $(h, k) \vDash \phi_1$ for all $j < k \leq i$

Note that we explicitly say there are metric operators $\diamond$ and $\diamondsuit$ because they have a subtle difference compared to that of the original LTL one, and we cannot reconcile the definition by using $\diamondsuit \Leftrightarrow \bullet\diamond$ either. The main difference is due to the time difference to the reference world. Let's say that we have $(h, i) \vDash \phi$, then we will have $(h, j) \vDash \bullet_{[0,10]} \diamond_{[0,10]} \phi$ but $(h, j) \nvDash \diamondsuit_{[0,10]}\phi$ for $j > i$, $\tau_i = 0$, and $\tau_j = 10$.

In the following, we will discuss the modifications to the logic so that we can accommodate the requirements to track transitivities and be able to do so efficiently. Then, we will describe the decision procedure to process the incoming events. We will extend the approach to dynamic programming in Thati and Rosu [2004] for this monitoring process. We will do this inductively, where the base case is when the first world ($h_1$) is created before receiving any input. In the case of handling recursive definition, since it is "guarded" and is processed for every incoming event, we can treat it as a normal predicate within the temporal operator.

## 2.3 The policy specification language RMTL

Our policy specification language, which we call RMTL, is based on an extension of metric linear-time temporal logic (MTL) by Thati and Rosu [2004]. The semantics

of LTL (Pnueli [1977] ) is defined in terms of models which are sequences of states (or worlds). In our case, we restrict to finite sequences of states. MTL extends LTL models by adding timestamps to each state and adding temporal operators that incorporate timing constraints. For example, MTL features temporal operators such as $\Diamond_{[0,3)}\phi$ which expresses that $\phi$ holds in some state in the future, and the timestamp of that world is within 0 to 3-time units from the current timestamp. We restrict to a model of MTL that uses discrete time, i.e., timestamps, in this case, are non-negative integers. We shall also restrict to the past-time fragment of MTL.

We extend MTL with two additional features: first-order quantifiers and recursive definitions. Our first-order language is a multi-sorted one. We consider only two sorts, which we call *prop* (for 'properties') and *app* (for denoting applications). Sorts are ranged over by $\alpha$. We first fix a *signature* $\Sigma$ for our first-order language, which is used to express terms and predicates of the language. We consider only constant symbols and predicate symbols, but no function symbols. We distinguish two types of predicate symbols: *defined* predicates and *undefined* ones. The defined predicate symbols are used to write recursive definitions, and to each of such symbols, we associate a formula as its definition.

Constant symbols are ranged over by $a$, $b$, and $c$, undefined predicate symbols are ranged over by $p$, $q$, and $r$, and defined predicate symbols are ranged over by $P$, $Q$, and $R$. We assume an infinite set of sorted variables $\mathcal{V}$, whose elements are ranged over by $x$, $y$, and $z$. We sometimes write $x_\alpha$ to say that $\alpha$ is the sort of variable $x$. A $\Sigma$-*term* is either a constant symbol $c \in \Sigma$ or a variable $x \in \mathcal{V}$. We use $s$, $t$, and $u$ to range over terms. To each symbol in $\Sigma$, we associate a sort information. We shall write $c : \alpha$ when $c$ is a constant symbol of sort $\alpha$. A predicate symbol of arity $n$ has sort of the form $\alpha_1 \times \cdots \times \alpha_n$, and such a predicate can only be applied to terms of sorts $\alpha_1, \ldots, \alpha_n$.

Constant symbols are used to express permissions in the Android OS, e.g., reading contacts, sending SMS, etc., and user ids of apps. Predicate symbols are used to express events such as ICC calls between apps, and properties of an app, such as whether it is a system app, a trusted app (as determined by the user). As standard in first-order logic (see e.g., Fitting [1996]), the semantics of terms and predicates are given in terms of a first-order structure, i.e., a set $\mathcal{D}_\alpha$, called a *domain*, for each sort $\alpha$, and an interpretation function $I$ assigning each constant symbol $c : \alpha \in \Sigma$ an element of $c^I \in \mathcal{D}_\alpha$ and each predicate symbol $p : \alpha_1 \times \cdots \times \alpha_n \in \Sigma$ an $n$-ary relation $p^I \subseteq \mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n}$. We shall assume constant domains in our model, i.e., every world has the same domain.

We define the formulas of RMTL via the following grammar:

$$\begin{aligned} F \coloneqq\ & \bot \mid p(t_1, \ldots, t_m) \mid P(t_1, \ldots, t_n) \mid F \vee F \mid \neg F \mid \bullet F \mid F \mathbin{\mathsf{S}} F \mid \blacklozenge F \mid \Diamond F \mid \\ & \bullet_n F \mid F \mathbin{\mathsf{S}}_n F \mid \blacklozenge_n F \mid \Diamond_n F \mid \exists_\alpha x.F \end{aligned}$$

where $m$ and $n$ are natural numbers. The existential quantifier is annotated with a sort information $\alpha$. For most of our examples and applications, we only quantify over variables of sort *app*. The operators indexed by $n$ are *metric temporal operators*. The $n \geq 1$ here denotes the interval $[0, n)$, so these are special cases of the more

general MTL operators in Thati and Rosu [2004], where intervals can take the form $[m, n)$, for $n \geq m \geq 0$. We use $\phi$, $\varphi$ and $\psi$ to range over formulas. We assume that unary operators bind stronger than the binary operators, so $\bullet\phi \vee \psi$ means $(\bullet\phi) \vee \psi$. We write $\phi(x_1, \ldots, x_n)$ to denote a formula whose free variables are among $x_1, \ldots, x_n$. Given such a formula, we write $\phi(t_1, \ldots, t_n)$ to denote the formula obtained by replacing $x_i$ with $t_i$ for every $i \in \{1, \ldots, n\}$.

To each defined predicate symbol $P : \alpha_1 \times \cdots \times \alpha_n$, we associate a formula $\phi_P$, which we call the *definition* of $P$. Notationally, we write $P(x_1, \ldots, x_n) := \phi_p(x_1, \ldots, x_n)$. We require that $\phi_P$ is *guarded*, i.e., every occurrence of any recursive predicate $Q$ in $\phi_P$ is prefixed by either $\bullet$, $\bullet_m$, $\diamondsuit$ or $\diamondsuit_n$. This guardedness condition is important to guarantee termination of recursion in model checking.

Given the above logical operators, we can define additional operators via their negation, e.g., $\top$ is defined as $\neg\bot$, $\phi \wedge \psi$ is defined as $\neg(\neg\phi \vee \neg\psi)$, $\phi \rightarrow \psi$ is defined as $\neg\phi \vee \psi$, and $\forall_\alpha x.\phi$ is defined as $\neg(\exists_\alpha x.\neg\phi)$, etc.

Before proceeding to the semantics of RMTL, we first define a well-founded ordering on formulae of RMTL, which will be used later.

**Definition 2.3.1.** *We define a relation $\prec_S$ on the set RMTL formulae as the smallest relation satisfying the following conditions:*

1. *For any formula $\phi$ of the form $p(\vec{t})$, $\bot$, $\bullet\psi$, $\bullet_n\psi$, $\diamondsuit\psi$ and $\diamondsuit_n\psi$, there is no $\phi'$ such that $\phi' \prec_S \phi$.*
2. *For every recursive definition $P(\vec{x}) := \phi_P(\vec{x})$, we have $\phi_P(\vec{t}) \prec_S P(\vec{t})$ for every terms $\vec{t}$.*
3. *$\psi \prec_S \psi \vee \psi'$, $\psi \prec_S \psi' \vee \psi$, $\psi \prec_S \neg\psi$, and $\psi \prec_S \exists x.\psi$.*
4. *$\psi_i \prec_S \psi_1 \,\mathsf{S}\, \psi_2$, and $\psi_i \prec_S \psi_1 \,\mathsf{S}_n\, \psi_2$, for $i \in \{1, 2\}$*

*We denote with $\prec$ the reflexive and transitive closure of $\prec_S$ .*

**Lemma 2.3.1.** *The relation $\prec$ on RMTL formulas is a well-founded partial order.*

*Proof.* We first show that $\prec$ is well-founded. Suppose otherwise: then there is an infinite descending chain of formulas:

$$\cdots \prec_S \phi_n \prec_S \phi_{n-1} \prec_S \cdots \prec_S \phi_2 \prec_S \phi_1.$$

Obviously, none of the $\phi_i$'s can be a bottom element (i.e., those that take the form as specified in clause (1) of Definition 2.3.1). Furthermore, there must be an $i$ such that $\phi_i = P(\vec{t})$ and $\phi_{i+1} = \phi_P(\vec{t})$ where $P$ is a recursive predicate defined by $P(\vec{x}) := \phi_P(\vec{x})$. If no such $i$ exists then all the instances of the relation $\prec_S$ in the chain must be instances of clause (3) and (4) in Definition 2.3.1, and the chain would be finite as those two clauses relate only strict subformulas. So without loss of generality, let us assume that $\phi_1 = P(\vec{t})$. We claim that for every $j > 1$, every occurrence of any recursive predicate in $\phi_j$ is guarded. We prove this by induction on $j$. If $j = 2$ then we have $\phi_2 \prec_S P(\vec{t})$. In this case, $\phi_2$ must be $\phi_P(\vec{t})$, and by the guardedness condition, all recursive predicates in $\phi_P$ are guarded. If $j > 2$, then we have $\phi_j \prec_S \phi_{j-1}$. By induction

hypothesis, all recursive predicates in $\phi_{j-1}$ are guarded. In this case, the relation $\phi_j \prec_S \phi_{j-1}$ must be an instance of either clause (3) or clause (4) of Definition 2.3.1, and therefore $\phi_j$ also satisfies the guardedness condition.

So now we have that none of $\phi_i$'s is recursive predicates. This means that all instances of $\prec_S$ in the chain must be instances of clause (3) and (4) in Definition 2.3.1, and consequently, the size of the formulas in the chain must be strictly decreasing. Thus the chain cannot be infinite, contrary to the assumption.

Anti-symmetry follows immediately from well-foundedness. Suppose $\prec$ is not anti-symmetric. Then we have a chain

$$\phi = \phi_1 \prec_S \phi_2 \prec_S \phi_3 \prec_S \cdots \prec_S \phi_n = \phi$$

where $n > 1$. We can repeat this chain to form an infinite descending chain, which contradicts the well-foundedness of $\prec$. □

For our application, we shall restrict to finite domains. Moreover, we shall restrict to an interpretation $I$ which is injective, i.e., mapping every constant $c$ to a unique element of $\mathcal{D}_\alpha$. In effect we shall be working in the term model, so elements of $\mathcal{D}_\alpha$ are just constant symbols from $\Sigma$. So we shall use a constant symbol, say $c : \alpha$, to mean both $c \in \Sigma$ and $c^I \in \mathcal{D}_\alpha$. With this fix interpretation, the definition of the semantics (i.e., the satisfiability relation) can be much simplified, e.g., we do not need to consider valuations of variables. A *state* is a set of undefined atomic formulas of the form $p(c_1, \ldots, c_n)$. Given a sequence $\sigma$, we write $|\sigma|$ to denote its length, and we write $\sigma_i$ to denote the $i$-th element of $\sigma$ when it is defined, i.e., when $1 \leq i \leq |\sigma|$. A *model* is a pair $(\pi, \tau)$ of a sequence of *states* $\pi$ and a sequence of *timestamps*, which are natural numbers, such that $|\pi| = |\tau|$ and $\tau_i \leq \tau_j$ whenever $i \leq j$.

The restriction to the application domain is good enough for our purpose since we assume that during a monitor run, the applications will be relatively constant. In the case where there is an update, i.e., there an app installed / uninstalled, then the monitor is reloaded to accommodate the change in the domain.

Let $<$ denote the total order on natural numbers. Then we can define a well-order on pairs $(i, \phi)$ of natural numbers and formulas by taking the lexicographical ordering $(<, \prec)$. The satisfiability relation between a model $\rho = (\pi, \tau)$, a *world* $i \geq 1$ (which is a natural number) and a *closed* formula $\phi$ (i.e., $\phi$ contains no free variables), written $(\rho, i) \vDash \phi$, is defined by induction on the pair $(i, \phi)$ as follows, where we write $(\rho, i) \nvDash \phi$ when $(\rho, i) \vDash \phi$ is false.

- $(\rho, i) \nvDash \bot$
- $(\rho, i) \vDash \neg\phi$ iff $(\rho, i) \nvDash \phi$.
- $(\rho, i) \vDash p(c_1, \ldots, c_n)$ iff $p(c_1, \ldots, c_n) \in \pi_i$.
- $(\rho, i) \vDash P(c_1, \ldots, c_n)$ iff $(\rho, i) \vDash \phi(c_1, \ldots, c_n)$ where $P(\vec{x}) \coloneqq \phi(\vec{x})$.
- $(\rho, i) \vDash \phi \vee \psi$ iff $(\rho, i) \vDash \phi$ or $(\rho, i) \vDash \psi$.

- $(\rho, i) \vDash \bullet\phi$ iff $i > 1$ and $(\rho, i-1) \vDash \phi$.
- $(\rho, i) \vDash \blacklozenge\phi$ iff there exists $j \leq i$ s.t. $(\rho, j) \vDash \phi$.
- $(\rho, i) \vDash \diamondsuit\phi$ iff $i > 1$ and there exists $j < i$ s.t. $(\rho, j) \vDash \phi$.
- $(\rho, i) \vDash \phi_1 \mathbin{\mathbb{S}} \phi_2$ iff there exists $j \leq i$ such that $(\rho, j) \vDash \phi_2$ and $(\rho, k) \vDash \phi_1$ for every $k$ s.t. $j < k \leq i$.
- $(\rho, i) \vDash \bullet_n\phi$ iff $i > 1$, $(\rho, i-1) \vDash \phi$ and $\tau_i - \tau_{i-1} < n$.
- $(\rho, i) \vDash \blacklozenge_n\phi$ iff there exists $j \leq i$ s.t. $(\rho, j) \vDash \phi$ and $\tau_i - \tau_j < n$.
- $(\rho, i) \vDash \diamondsuit_n\phi$ iff $i > 1$ and there exists $j < i$ s.t. $(\rho, j) \vDash \phi$ and $\tau_i - \tau_j < n$.
- $(\rho, i) \vDash \phi_1 \mathbin{\mathbb{S}_n} \phi_2$ iff there exists $j \leq i$ such that $(\rho, j) \vDash \phi_2$, $(\rho, k) \vDash \phi_1$ for every $k$ s.t. $j < k \leq i$, and $\tau_i - \tau_j < n$.
- $(\rho, i) \vDash \exists_\alpha x.\phi(x)$ iff there exists $c \in \mathcal{D}_\alpha$ s.t. $(\rho, i) \vDash \phi(c)$.

Note that due to the guardedness condition in recursive definitions, our semantics for recursive predicates is much simpler than the usual definition as in $\mu$-calculus, which typically involves the construction of a (semantic) fixed point operator. Note also that some operators are redundant, e.g., $\blacklozenge\phi$ can be defined as $\top \mathbin{\mathbb{S}} \phi$, and $\diamondsuit\phi$ can be defined as $\bullet\blacklozenge\phi$. This holds for some metric operators, e.g., $\blacklozenge_n\phi$ and $\diamondsuit_n\phi$ can be defined as, respectively, $\top \mathbin{\mathbb{S}_n} \phi$ and

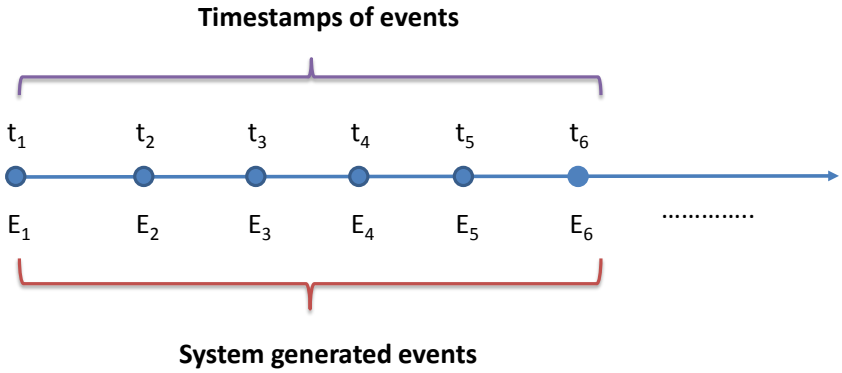
$$\diamondsuit_n\phi = \bigvee_{i+j=n} \bullet_i \blacklozenge_j \phi \tag{2.2}$$

This operator will be used to specify an active call chain, as we shall see later, so it is convenient to include it in our policy language.

In the next section, we shall assume that $\blacklozenge$, $\diamondsuit$, $\blacklozenge_n$ are derived connectives. Since we consider only finite domains, $\exists_\alpha x.\phi(x)$ can be reduced to a big disjunction $\bigvee_{c \in \mathcal{D}_\alpha} \phi(c)$, so we shall not treat the $\exists$-quantifier explicitly. This can be problematic if the domain of quantification is big, as it suffers the same kind of exponential explosion as with the expansion of metric operators in MTL Thati and Rosu [2005]. We shall defer the explicit treatment of quantifiers to future work.

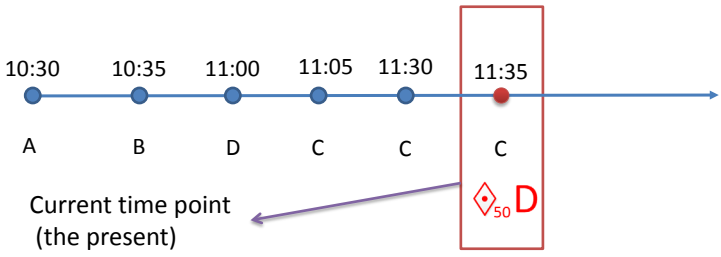
Formulas of RMTL are interpreted in a semantic model which consists of a sequence of events, temporally ordered. In LTL, one is only concerned with the relative ordering of events, but not the exact time and date an event occurred. So one can talk about event $A$ occurred before event $B$, but not that event $A$ occurred at a specific time, or within a specific interval of time from event $B$. Graphically, the semantic

model of RMTL can be presented as follows:



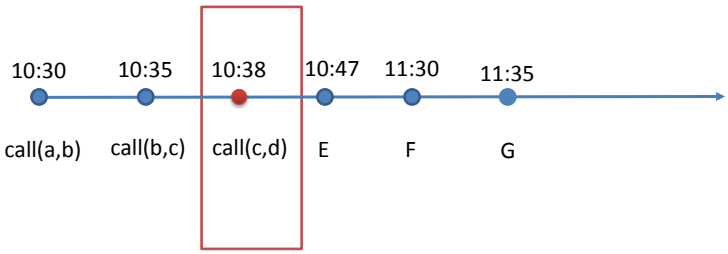

Each dot represents a time point in the timeline. At each time point, we keep two kinds of information: the event that takes place at that particular time point, and the timestamp of that event. Given such a model, we can then specify certain relations between events in the timeline, from the perspective of a particular time point (e.g., the present time). The operators of RMTL provides the building blocks to specify a rich collection of patterns of relations one can specify. We give a couple of examples next.

For the first example, consider the formula $\diamond_{50} D$. Intuitively, this formula states that event $D$ was observed sometime in the past within 50-time units from the presence. Graphically, this is illustrated as follows:




Temporal operators alone are not enough to specify some recursive patterns needed to capture privilege escalation. For example, consider a sequence of ICC

calls, as represented in the following model:




In this case, there is a call chain, starting with $a$ and ending in $d$, such that each successive call happened within 10-time units. One can imagine a generalization of this case to an arbitrary number of successive calls that are chained together. To deal with such a possibility, we introduce a notion of recursive definition. In this case, we create a new predicate, called *trans*, that effectively defines the transitive closure of the *call* events. This can be expressed in RMTL as follows:

$$trans(x,y) := call(x,y) \vee \exists z.\ \Diamond_n\, trans(x,z) \wedge call(z,y)$$

where $n$ is a time interval. Intuitively, this definition of $trans(x,y)$ says that there is a call chain from $x$ to $y$ if and only if either $x$ calls $y$ directly, or there is a call chain from $x$ to some $z$, and there is a direct call from $z$ to $y$.

## 2.4 Trace-length independent monitoring

The problem of monitoring is essentially a problem of model checking, i.e., to decide whether $(\rho, i) \vDash \phi$, for any given $\rho = (\pi, \tau)$, $i$ and $\phi$. In the context of Android runtime monitoring, a state in $\pi$ can be any events of interest that one would like to capture, e.g., ICC call events, queries related to location information or contacts, etc. To simplify discussions, and because our main interest is in privilege escalation through ICC, the only type of event we consider in $\pi$ is the ICC event, which we model with the predicate $call : app \times app$.

Given a policy specification $\phi$, a naive monitoring algorithm that enforces this policy would store the entire event history $\pi$ and every time a new event arrives at time $t$, it would check $(([\pi; e], [\tau; t]), |\rho| + 1) \vDash \phi$. where ; denotes concatenation. This is easily shown decidable but is of course rather inefficient. In general, the model checking problem for RMTL (with finite domains) can be shown to be PSPACE-hard following the same argument as in Bauer et al. [2009]. One of the design criteria of RMTL is that enforcement of policies does not depend on the length of history of events, i.e., at any time the monitor only needs to keep track of a fixed number of states. Following Bauer et al. [2013], we call a monitoring algorithm that satisfies this property *trace-length independent*.

For PTLTL, trace-length independent monitoring algorithm exists, e.g., the algo-

rithm by Havelund and Rosu [2002], which depends only on two states in a history. That is, satisfiability of $(\rho, i+1) \vDash \phi$ is a boolean function of satisfiability of $(\rho, i+1) \vDash \psi$, for every strict subformula $\psi$ of $\phi$, and satisfiability of $(\rho, i) \vDash \psi'$, for every subformula $\psi'$ of $\phi$. This works for PTLTL because the semantics of temporal operators in PTLTL can be expressed in a recursive form, e.g., the semantics of S can be equally expressed as Havelund and Rosu [2002]: $(\rho, i+1) \vDash \phi_1 \ \mathsf{S} \ \phi_2$ iff $(\rho, i+1) \vDash \phi_2$, or $(\rho, i+1) \vDash \phi_1$ and $(\rho, i) \vDash \phi_1 \ \mathsf{S} \ \phi_2$. This is not the case for MTL. For example, satisfiability of the unrestricted 'since' operator $\mathsf{S}_{[m,n)}$ can be equivalently expressed as:

$$(\rho, i+1) \vDash \phi_1 \ \mathsf{S}_{[m,n)} \ \phi_2 \text{ iff } \quad \begin{array}{l} m = 0, n > 1, \text{ and } (\rho, i+1) \vDash \phi_2, \text{ or} \\ (\rho, i+1) \vDash \phi_1 \text{ and } (\rho, i) \vDash \phi_1 \ \mathsf{S}_{[m',n')} \ \phi_2 \end{array} \tag{2.3}$$

where $m' = min(0, m - \tau_{i+1} + \tau_i)$ and $n' = min(0, n - \tau_{i+1} + \tau_i)$. Since $\tau_{i+1}$ can vary, the value of $m'$ and $n'$ can vary, depending on the history $\rho$. We avoid the expansion of metric operators in monitoring by restricting the intervals in the metric operators to the form $[0, n)$. We show that clause (2.3) can be brought back to a purely recursive form. The key to this is the following lemma:

**Lemma 2.4.1** (Minimality)**.** *If $(\rho, i) \vDash \phi_1 \ \mathsf{S}_n \ \phi_2$ [$(\rho, i) \vDash \diamondsuit_n \phi$] then there exists an $m \leq n$ such that $(\rho, i) \vDash \phi_1 \ \mathsf{S}_m \ \phi_2$ [resp. $(\rho, i) \vDash \diamondsuit_m \phi$] and such that for every $k$ such that $0 < k < m$, we have $(\rho, i) \nvDash \phi_1 \ \mathsf{S}_k \ \phi_2$ [resp., $(\rho, i) \nvDash \diamondsuit_k \phi$].*

*Proof.* We show a case for $(\rho, i) \vDash \phi_1 \ \mathsf{S}_n \ \phi_2$; the other case is straightforward. We prove this by induction on $i$.

- Base case: $i = 1$. Since $(\rho, i) \vDash \phi_1 \ \mathsf{S}_n \ \phi_2$, it must be the case that $(\rho, i) \vDash \phi_2$. In this case, let $m = 1$. Obviously $(\rho, i) \vDash \phi_1 \ \mathsf{S}_m \ \phi_2$ and $m$ is minimal.

- Inductive case: $i > 1$. We have $(\rho, i) \vDash \phi_1 \ \mathsf{S}_n \ \phi_2$. By the definition of $\vDash$, there exists $j \leq i$ such that $(\rho, j) \vDash \phi_2$ and $(\rho, k) \vDash \phi_1$ for every $k$ s.t. $j < k \leq i$ and $\tau_i - \tau_j < n$. If $(\rho, i) \vDash \phi_2$, then we have $(\rho, i) \vDash \phi_1 \ \mathsf{S}_1 \ \phi_2$. In this case, let $m = 1$. If $(\rho, i) \nvDash \phi_2$, then it must be the case that $j < i$. It is not difficult to see that in this case, we must have

  $$(\rho, i-1) \vDash \phi_1 \ \mathsf{S}_{n-(\tau_i - \tau_{i-1})} \ \phi_2.$$

  By the induction hypothesis, we have there is an $m'$ such that

  $$(\rho, i-1) \vDash \phi_1 \ \mathsf{S}_{m'} \ \phi_2$$

  and for every $l$ s.t. $l < m'$, we have $(\rho, i-1) \nvDash \phi_1 \ \mathsf{S}_l \ \phi_2$. In this case, we let $m = m' + \tau_i - \tau_{i-1}$. It is straightforward to check that $(\rho, i) \vDash \phi_1 \ \mathsf{S}_m \ \phi_2$.

  Now, we claim that this $m$ is minimal. Suppose otherwise, i.e., there exists $k < m$ s.t. $(\rho, i) \vDash \phi_1 \ \mathsf{S}_k \ \phi_2$. Since $(\rho, i) \nvDash \phi_2$, we must have $(\rho, i-1) \vDash \phi_1 \ \mathsf{S}_{k-(\tau_i - \tau_{i-1})} \ \phi_2$. However, $(k - (\tau_i - \tau_{i-1})) < m'$, so this contradicts the minimality of $m'$.

□

**Lemma 2.4.2** (Monotonicity)**.** *If* $(\rho, i) \vDash \phi_1 \, \mathbb{S}_n \, \phi_2$ *[resp.,* $(\rho, i) \vDash \bullet_n \phi$ *and* $(\rho, i) \vDash \diamond_n \phi$*] then for every* $m \geq n$*, we have* $(\rho, i) \vDash \phi_1 \, \mathbb{S}_m \, \phi_2$ *[resp.,* $(\rho, i) \vDash \bullet_m \phi$ *and* $(\rho, i) \vDash \diamond_m \phi$*].*

*Proof.* Straightforward from the definition of $\vDash$. □

Given $\rho$, $i$ and $\phi$, we define a function $\mathfrak{m}$ as follows:

$$\mathfrak{m}(\rho, i, \phi) = \begin{cases} m, \text{ if } \phi \text{ is either } \phi_1 \, \mathbb{S}_n \, \phi_2 \text{ or } \diamond_n \phi' \text{ and } (\rho, i) \vDash \phi, \\ 0, \text{ otherwise.} \end{cases}$$

where $m$ is as given in Lemma 2.4.1; we shall see how its value is calculated in Algorithm 3. The following theorem follows from Lemma 2.4.1.

**Theorem 2.4.1** (Recursive forms)**.** *For every model* $\rho$*, every* $n \geq 1$*,* $\phi$*,* $\phi_1$ *and* $\phi_2$*, and every* $1 < i \leq |\rho|$*, the following hold:*

1. $(\rho, i) \vDash \phi_1 \, \mathbb{S}_n \, \phi_2$ *iff* $(\rho, i) \vDash \phi_2$*, or* $(\rho, i) \vDash \phi_1$ *and* $(\rho, i-1) \vDash \phi_1 \, \mathbb{S}_n \, \phi_2$ *and* $n - (\tau_i - \tau_{i-1}) \geq \mathfrak{m}(\rho, i-1, \phi_1 \, \mathbb{S}_n \, \phi_2)$.
2. $(\rho, i) \vDash \diamond_n \phi$ *iff* $(\rho, i-1) \vDash \phi$ *and* $\tau_i - \tau_{i-1} < n$*, or* $(\rho, i-1) \vDash \diamond_n \phi$ *and* $n - (\tau_i - \tau_{i-1}) \geq \mathfrak{m}(\rho, i-1, \diamond_n \phi)$.

*Proof.* We show the case for $\mathbb{S}_n$; the other case is similar.

Suppose $(\rho, i) \vDash \phi_1 \, \mathbb{S}_n \, \phi_2$. By definition, there exists $j \leq i$ such that

$$(\rho, j) \vDash \phi_2, \tag{2.4}$$

$$\tau_i - \tau_j \leq n, \tag{2.5}$$

and for every $k$ s.t. $j < k \leq i$ we have

$$(\rho, k) \vDash \phi_1, \tag{2.6}$$

Suppose that the right-hand side of iff doesn't hold, i.e., we have $(\rho, i) \nvDash \phi_2$, and that one of the following hold:

- $(\rho, i) \nvDash \phi_1$,
- $(\rho, i-1) \nvDash \phi_1 \, \mathbb{S}_n \, \phi_2$, or
- $n - (\tau_i - \tau_{i-1}) < \mathfrak{m}(\rho, i-1, \phi_1 \, \mathbb{S}_n \, \phi_2)$.

The first case contradicts our assumption in (2.6), so it cannot hold. Note that since $(\rho, i) \nvDash \phi_2$, it must be the case that $j \leq i-1$, so (2.4), (2.5), and (2.6) above entail that $(\rho, i-1) \vDash \phi_1 \, \mathbb{S}_n \, \phi_2$, so the second case can't hold either. For the third case: from (2.5) we have $\tau_i - \tau_j = (\tau_i - \tau_{i-1}) + (\tau_{i-1} - \tau_j) \leq n$ so

$$\tau_{i-1} - \tau_j \leq n - (\tau_i - \tau_{i-1}).$$

**Algorithm 1** *Monitor*$(\rho, i, \phi)$

```
Init(ρ, φ, prev, cur, mprev, mcur)
for j = 1 to i do
  Iter(ρ, j, φ, prev, cur, mprev, mcur);
return cur[idx(φ)];
```

This, together with (2.4) and (2.5), implies that $(\rho, i-1) \vDash \phi_1 \, \mathbb{S}_{n-(\tau_i-\tau_{i-1})} \, \phi_2$. If $n - (\tau_i - \tau_{i-1}) < \mathfrak{m}(\rho, i-1, \phi_1 \, \mathbb{S}_n \, \phi_2)$ holds, then it would contradict the fact that $\mathfrak{m}(\rho, i-1, \phi_1 \, \mathbb{S}_n \, \phi_2)$ is the minimal index as guaranteed by Lemma 2.4.1. Hence the third case cannot hold either.

For the other direction, suppose that either of the following holds:

- $(\rho, i) \vDash \phi_2$, or
- $(\rho, i) \vDash \phi_1$ and $(\rho, i-1) \vDash \phi_1 \, \mathbb{S}_n \, \phi_2$ and $n - (\tau_i - \tau_{i-1}) \geq \mathfrak{m}(\rho, i-1, \phi_1 \, \mathbb{S}_n \, \phi_2)$.

If it is the first case, i.e., $(\rho, i) \vDash \phi_2$, then trivially $(\rho, i) \vDash \phi_1 \, \mathbb{S}_n \, \phi_2$. So suppose it is the second case. By Lemma 2.4.1, we have $(\rho, i-1) \vDash \phi_1 \, \mathbb{S}_m \, \phi_2$, where $m = \mathfrak{m}(\rho, i-1, \phi_1 \, \mathbb{S}_n \, \phi_2)$. Since $n - (\tau_i - \tau_{i-1}) \geq m$, by Lemma 2.4.2, we have $(\rho, i-1) \vDash \phi_1 \, \mathbb{S}_{n-(\tau_i-\tau_{i-1})} \, \phi_2$. This means there exists $j \leq i-1$ such that

- $(\rho, j) \vDash \phi_2$,
- $(\tau_{i-1} - \tau_j) \leq n - (\tau_i - \tau_{i-1})$, and
- $(\rho, k) \vDash \phi_1$, for every $k$ s.t. $j < k \leq i-1$.

The second item implies that $(\tau_i - \tau_j) \leq n$. These and the fact that $(\rho, i) \vDash \phi_1$ entail that $(\rho, i) \vDash \phi_1 \, \mathbb{S}_n \, \phi_2$. □ □

Given Theorem 2.4.1, we can adapt the monitoring algorithm for PTLTL in Havelund and Rosu [2002], but with an added data structure to keep track of the function $\mathfrak{m}$. In the following, given a formula $\phi$, we assume that $\exists$, $\blacklozenge$ and $\diamond$ have been replaced with its equivalent form as mentioned in Section 2.3.

Given a formula $\phi$, let $Sub(\phi)$ be the set of subformulas of $\phi$. We define a closure set $S^*(\phi)$ of $\phi$ as follows: Let $Sub^0(\phi) = Sub(\phi)$, and let

$$Sub^{n+1}(\phi) = Sub_n(\phi) \cup \{Sub(\phi_P(\vec{c})) \mid P(\vec{c}) \in Sub_n(\phi), \text{ and } P(\vec{x}) := \phi_P(\vec{x})\}$$

and define $Sub^*(\phi) = \bigcup_{n \geq 0} Sub^n(\phi)$. Since $\mathcal{D}_\alpha$ is finite, $Sub^*(\phi)$ is finite, although its size is exponential in the arities of recursive predicates. For our specific applications, the predicates used in our sample policies have at most arity of two (for tracking transitive calls), so this is still tractable. In future work, we plan to investigate ways of avoiding this explicit expansion of recursive predicates.

We now describe how monitoring can be done for $\phi$, given $\rho$ and $1 \leq i \leq |\rho|$. We assume implicitly a preprocessing step where we compute $Sub^*(\phi)$; we do not describe this step here, but it is quite obvious from the definition. Let $\phi_1, \phi_2, \ldots, \phi_m$ be

**Algorithm 2** $Init(\rho, \phi, prev, cur, mprev, mcur)$

**for** $k = 1, \ldots, m$ **do**
  $prev[k] := false$, $mprev[k] := 0$ and $mcur[k] := 0$;
**for** $k = 1, \ldots, m$ **do**
  **switch** $(\phi_k)$
  **case** $(\bot)$**:** $cur[k] := false$;
  **case** $(p(\vec{c}))$**:** $cur[k] := p(\vec{c}) \in \pi_1$;
  **case** $(P(\vec{c}))$**:** $cur[k] := cur[idx(\phi_P(\vec{c}))]$; {Suppose $P(\vec{x}) := \phi_P(\vec{x})$.}
  **case** $(\neg\psi)$**:** $cur[k] := \neg cur[idx(\psi)]$;
  **case** $(\psi_1 \vee \psi_2)$**:** $cur[k] := cur[idx(\psi_1)] \vee cur[idx(\psi_2)]$;
  **case** $(\bullet\psi)$**:** $cur[k] := false$;
  **case** $(\diamondsuit\psi)$**:** $cur[k] := false$;
  **case** $(\psi_1 \, \mathsf{S} \, \psi_2)$**:** $cur[k] := cur[idx(\psi_2)]$;
  **case** $(\bullet_n\psi)$**:** $cur[k] := false$;
  **case** $(\diamondsuit_n\psi)$**:** $cur[k] := false; mcur[k] := 0$;
  **case** $(\psi_1 \, \mathsf{S}_n \, \psi_2)$**:**
    $cur[k] := cur[idx(\psi_2)]$;
    **if** $cur[k] = true$ **then** $mcur[k] := 1$;
    **else** $mcur[k] := 0$;
    **end if**
  **end switch**
**return** $cur[idx(\phi)]$;

an enumeration of $Sub^*(\phi)$ respecting the partial order $\prec$, i.e., if $\phi_i \prec \phi_j$ then $i \leq j$. Then we can assign to each $\psi \in Sub^*(\phi)$ an index $i$, s.t., $\psi = \phi_i$ in this enumeration. We refer to this index as $idx(\psi)$. We maintain two boolean arrays $prev[1, \ldots, m]$ and $cur[1, \ldots, m]$. The intention is that given $\rho$ and $i > 1$, the value of $prev[k]$ corresponds to the truth value of the judgment $(\rho, i-1) \vDash \phi_k$ and the truth value of $cur[k]$ corresponds to the truth value of the judgment $(\rho, i) \vDash \phi_k$. We also maintain two integer arrays $mprev[1, \ldots, m]$ and $mcur[1, \ldots, m]$ to store the value of the function $\mathfrak{m}$. The value of $mprev[k]$ corresponds to $\mathfrak{m}(\rho, i-1, \phi_k)$, and $mcur[k]$ corresponds to $\mathfrak{m}(\rho, i, \phi_k)$. Note that this preprocessing step only needs to be done once, i.e., when generating the monitor codes for a particular policy, which is done offline, prior to inserting the monitor into the operating system kernel.

The main monitoring algorithm is divided into two subprocedures: the initialization procedure (Algorithm 2) and the iterative procedure (Algorithm 3). The monitoring procedure (Algorithm 1) is then a simple combination of these two. We overload some logical symbols to denote operators on boolean values. In the actual implementation, we do not actually implement the loop in Algorithm 1; rather it is implemented as an event-triggered procedure, to process each event as they arrive using *Iter*.

**Theorem 2.4.2.** $(\rho, i) \vDash \phi$ *iff* $Monitor(\rho, i, \phi)$ *returns true.*

*Proof.* The proof is quite straightforward because each step in the calculation of the

truth values of subformulas of $\phi$ corresponds to their (recursive-form) semantics reading and Theorem 2.4.1. Moreover, by the well-foundedness of $\prec$, this algorithm terminates, and at each step, the calculation of a subformula $\phi$ uses only values of other subformulas smaller than $\phi$ in the ordering $\prec$, or truth values of subformulas in the previous world (which are already defined). In the case of $\mathbb{S}_n$, the updates of the value *mcur* and *mprev* correspond exactly to the construction shown in the proof of Lemma 2.4.1. □

The *Iter* function only depends on two worlds: $\rho_i$ and $\rho_{i-1}$, so the algorithm is trace-length independent. In principle, there is no upper bound to its space complexity, as the timestamp $\tau_i$ can grow arbitrarily large, as is the case in Basin et al. [2012]. Practically, however, the timestamps in Android are stored in a fixed size data structure, so in such a case, when we fix the policy, the space complexity is constant (i.e., independent of the length of history $\rho$).

## 2.5 Examples

We provide some basic policies as an example of how we can use this logic to specify security policies. From now on, we shall only quantify over the domain $app$, so in the following, we shall omit the sort annotation in the existential quantifier. The predicate *trans* is the recursive predicate defined in Equation (2.1) in the introduction, i.e.,

$$trans(x,y) := call(x,y) \vee \exists z.\, \diamondsuit_n\, trans(x,z) \wedge call(z,y).$$

The constant *sink* denotes a service or resource that an unprivileged application tries to access via privilege escalation, e.g., send SMS, or access to the Internet. The constant *contact* denotes the Contact provider app in Android. We also assume the following "static" predicates (i.e., their truth values do not vary over time):

- $system(x)$: $x$ is a system app or process. By system app here we mean any app that is provided and certified by Google (such as Google Maps, Google Play, etc.) or an app that comes preinstalled on the phone.
- $hasPermissionToSink(y)$: $y$ has permission to access the sink.
- $trusted(x)$: $x$ is an app that the user trusts. This is not a feature of Android, rather, it is a specific feature of our implementation. We build into our implementation a 'trust' management app to allow the user a limited control over apps that he/she trusts.

The following policies refer to access patterns that are forbidden. So given a policy $\phi$, the monitor at each state $i$ make sure that $(\rho,i) \not\models \phi$ holds. Assuming that $(\rho,i) \not\models \phi$, where $i = |\rho|$, holds, then whenever a new event (i.e., the ICC call) $e$ is registered at time $t$, the monitor checks that $(([\pi;e],[\tau;t]),i+1) \not\models \phi$ holds. If it does, then the call is allowed to proceed. Otherwise, it will be terminated.

1. $\exists x.(call(x, sink) \wedge \neg system(x) \wedge \neg trusted(x))$.

   This is a simple policy where we block a direct call from an untrusted application to the sink. This policy can serve as a privilege manager where we dynamically revoke permission for the application to access the sink regardless of the static permission it asked during installation.

2. $\exists_x(trans(x, sink) \wedge \neg system(x) \wedge \neg hasPermissionToSink(x))$.

   This policy says that transitive calls to a sink from non-system apps are forbidden unless the source of the calls already has permission to the sink. This is then a simple privilege escalation detection (for non-system apps).

3. $\exists_x(trans(x, sink) \wedge \neg system(x) \wedge \neg trusted(x))$.

   This is a further refinement to the policy in that we also give the user privilege to decide for themselves dynamically whether or not to trust an application. Untrusted apps cannot make a transitive call to the sink, but trusted apps are allowed, regardless of their permissions.

4. $\exists_x(trans(x, internet) \wedge \neg system(x) \wedge \neg trusted(x) \wedge \diamond(call(x, contact)))$.

   This policy allows privilege escalation by non-trusted apps as long as there is no potential for data leakage through the sink. That is, as soon as a non-system and untrusted app accesses contact, it will be barred from accessing the internet. Note that the use of non-metric operator $\diamond$ ensures that the information that a particular app has accessed contact is persistent. This policy resembles the well-known Chinese Wall policy Brewer and Nash [1989] that is often used to manage conflict of interests. Here accessing contacts and connecting to the internet are considered as different conflict-of-interests classes.

## 2.6 Implementation

We have implemented the monitoring algorithm presented in the previous section in Android 4.1. Some modifications to the application framework and the underlying Linux kernel are necessary to ensure our monitor can effectively monitor and stop unwanted behaviors. We have tested our implementation in both Android emulator and an actual device (Samsung Galaxy Nexus phone).

Our implementation consists of two parts: the code that generates a monitor given a policy specification, and the modifications of Android framework and its Linux kernel to hook our monitor and to intercept ICCs and access to Android resources. The modification of the Android framework mainly revolves around Activity Manager Service, a system component which deals with processing intent. We add a hook in the framework to redirect permission checking (either starting activity, service, or broadcasting intent) to pass through our monitor first before going to the usual Android permission checking. The modification to the kernel consists mainly of additional system calls to interact with the framework and a monitor stub which

is activated when the monitor module is loaded. To improve runtime performance, the monitor generation is done outside Android; it produces C code that is then compiled into a kernel module, and inserted into the Android boot image.

### 2.6.1 Monitor Generator

The monitor generator takes an input policy, encoded in an XML format extending that of RuleML. The monitor generator works by following the logic of the monitoring algorithm presented in Section 2.4. It takes a policy formula $\phi$, and generates the required data structures and determines an ordering between elements of $Sub^*(\phi)$ as described earlier, and produces the code illustrated in Algorithm 2, 3 and 1. The main body of our monitor lies in the Linux kernel space as a kernel module. The reason for this is that there are some cases where Android leaves the permission checking to the Linux kernel layer, e.g., for opening network socket. However, to monitor the ICC events between Android components and apps, we need to place a hook inside the application framework. The ICC between apps is done by passing a data structure called *Intent*, which gets broken down into *parcels* before they are passed down to the kernel level to be delivered. So intercepting these parcels and reconstructing the original Intent object in the kernel space would be more difficult and error-prone. The events generated by apps or components will be passed down to the monitor in the kernel, along with the application's user id. If the event is a call to the sink, then depending on the policy that is implemented in the monitor, it will decide whether to block or allow the call to proceed. We do this through custom additional system calls to the Linux kernel which go to this monitor.

### 2.6.2 LogicDroid Architecture

The LogicDroid architecture is given in Figure 2.1. The detection of events is done via various "hooks" into the Android OS; these include hooks in the framework layer, system library and the Linux kernel itself. Events intercepted, such as requests to resources by an app, are forwarded to the reference monitor for further evaluation. We will discuss some implementations of the hooks below. The reference monitor resides in the Linux kernel underlying the Android OS. It processes events as they come, and for each new event, it checks whether the new event, together with the history of events processed so far, violates the current security policy. Note that we implement our security checking mechanisms on top of the Android permission mechanism, so in the case where our reference monitor does nothing, the default Android permission mechanism would still be enforced.

The current security policy to be enforced is hardcoded in the monitor for efficiency reasons. This, however, makes it slightly complicated to update the security policy in the monitor. We need to provide a facility to update security policies since this might be required to counter new forms of attacks that may not be handled by the current policy. Our current approach is to design the monitor as a kernel module; this allows it to be removed and reinstalled on a live system. We then devise an

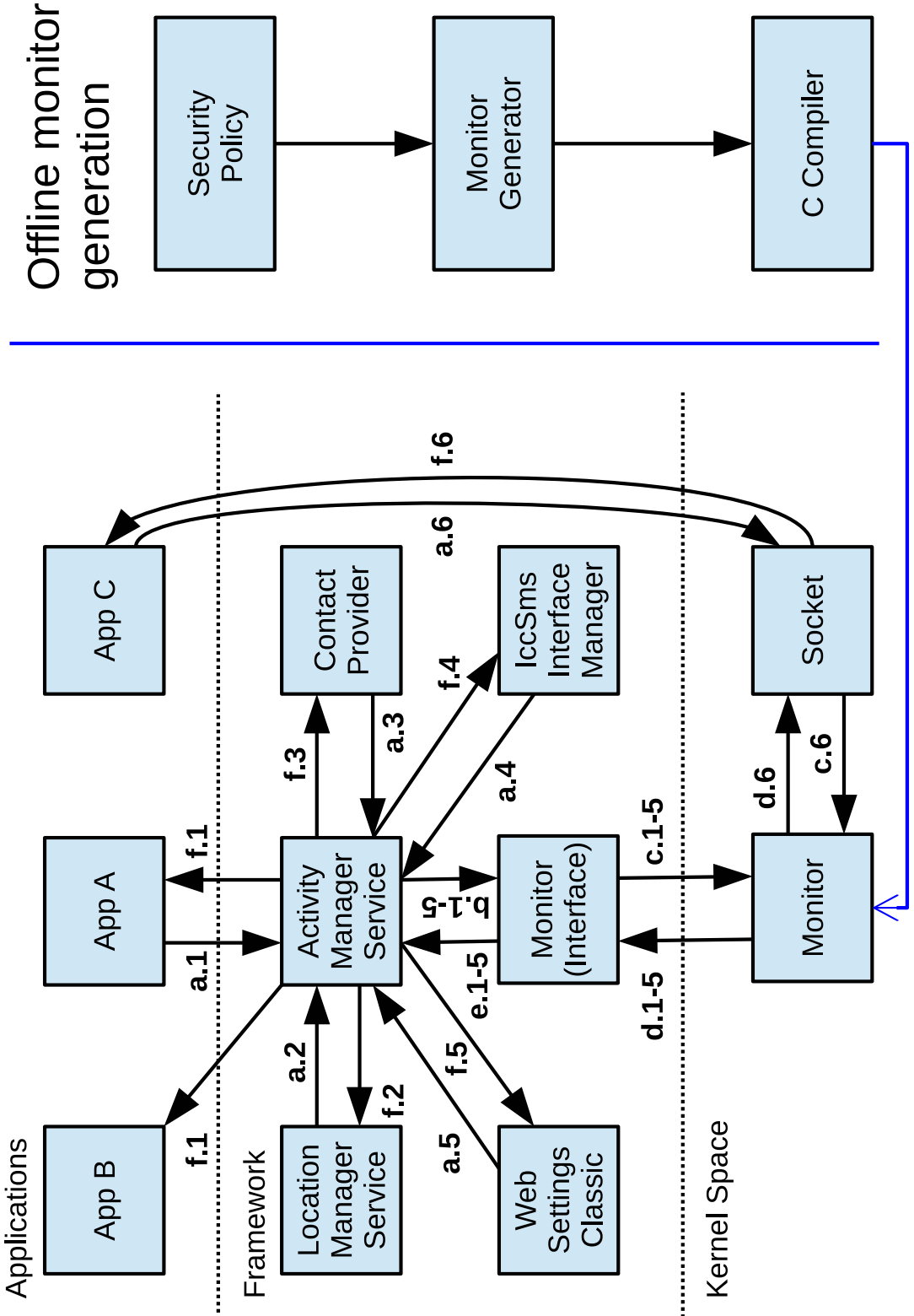



Figure 2.1: The architecture of LogicDroid

offline monitor generation process. The monitor generator takes as input a security policy (specified in RMTL, using XML as the representation language) and generates C code that corresponds to the kernel module of the policy. This appears to be a good compromise between performance and security.

Our implementation places hooks in four services, namely accessing the Internet (opening network sockets), sending SMS, accessing location, and accessing contact database. For each of these sinks, we add a virtual UID in the monitor and treat it as a component of Android. We currently track only ICC calls through the Intent passing mechanism. Android sandboxing restricts how apps can communicate. Communication between apps is typically done through the so-called *Intent* objects in the Android framework. It is essentially a form of inter-process communication (IPC) mechanism specific to Android. At the application level, we can often track leakage of permissions tracking the passing of *Intent* objects, so our framework tracks this kind of intents objects to infer privilege escalation. The most time-consuming part of our implementation was actually in tracing how *Intent* objects are processed, and in figuring out where to put the hooks to be most effective in intercepting requests to resources.

**The phone call hook** In general, resources in Android that are protected by the Android permission mechanism are located on the framework level. One such resource is the ability to make phone calls. In the source code of Android, this functionality is located in the `OutgoingCallBroadcaster.java` file. This file is compiled into a component, and when an app wants to make a phone call, it will need to send an Intent object to this component. We can intercept the phone call request in this component. Figure 2.2 shows a snippet of the code where the permission checked is done. We place the hook just before the Android permission checks take place.

**The socket hook** This hook is used to detect an attempt to connect to a network (or internet) by opening a socket. Intercepting the attempts to connect to the internet or a local network proves to be quite tricky. Although there are components in the Android framework that provide internet related services, such as HTTP protocol, an app can bypass these components entirely and simply make a system call to open sockets directly. Thus to intercept this connection, we need to place the hook in the Linux kernel. In Android, any app that wants to connect to a network needs to have the INTERNET permission. This permission is checked in the kernel when an app tries to open a socket, and the permission checking is enforced through Linux access control, i.e., the network socket connection is associated with a particular group called INTERNET. Only applications (users) belonging to this group will be able to open a socket. Practically, when an application asks for the INTERNET permission during installation, it will be added as a member of the "INTERNET" group. The hook for the socket is placed in the `socket.c` file (from the Linux kernel source). The extra security checking done by LogicDroid is the same in this case, i.e., it will consult our custom monitor before returning to the normal execution flow. The snippet of code and where to place the hook is shown in Figure 2.3

This is obviously not enough to detect all possible communications between apps, e.g., those that are done through the file systems, or side channels, such as vibration

```
private void processIntent(Intent intent) {
  …

  /* Change CALL_PRIVILEGED into CALL or CALL_EMERGENCY as needed. */
  // TODO: This code is redundant with some code in InCallScreen: refactor.
  if (Intent.ACTION_CALL_PRIVILEGED.equals(action)) {
      // We're handling a CALL_PRIVILEGED intent, so we know this request came
      // from a trusted source (like the built-in dialer.)  So even a number
      // that's *potentially* an emergency number can safely be promoted to
      // CALL_EMERGENCY (since we *should* allow you to dial "91112345" from
      // the dialer if you really want to.)

      Redirect the checking to our monitor here

      if (isPotentialEmergencyNumber) {
          Log.i(TAG, "ACTION_CALL_PRIVILEGED is used while the number is a potential"
                  + " emergency number. Use ACTION_CALL_EMERGENCY as an action instead.");
          action = Intent.ACTION_CALL_EMERGENCY;
      } else {
          action = Intent.ACTION_CALL;
      }
      if (DBG) Log.v(TAG, " - updating action from CALL_PRIVILEGED to " + action);
      intent.setAction(action);
  }
  ...
}
```

Figure 2.2: A hook in the Android framework to intercept phone calls

```
SYSCALL_DEFINE3(bind, int, fd, struct sockaddr __user *, umyaddr, int, addrlen)
{
    struct socket *sock;
    struct sockaddr_storage address;
    int err, fput_needed;

    sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (sock) {
        err = move_addr_to_kernel(umyaddr, addrlen, (struct sockaddr *)&address);
        if (err >= 0) {
            err = security_socket_bind(sock,
                                       (struct sockaddr *)&address,
                                       addrlen);
```

Redirect the checking to our monitor here

```
            if (!err)
                err = sock->ops->bind(sock,
                                      (struct sockaddr *)
                                      &address, addrlen);
        }
        fput_light(sock->file, fput_needed);
    }
    return err;
}
```

Figure 2.3: A hook placed in the Linux kernel to intercept calls to network sockets

setting (e.g., as implemented in SoundComber by Schlegel et al. [2011]), so our implementation is currently more of a proof of concept. In the case of SoundComber, our monitor can actually intercept the calls between colluding apps, due to the fact that the sender app broadcasts an intent to signal receiver app to start listening for messages from the covert channels.

### 2.6.3 Performance

We have implemented some apps to test the policies we mentioned in Section 2.5. There are two apps with a different subset of privileges, one app has the privilege to access the sensitive information such as GPS location but does not have the privilege to access the sink, and the other app does not have the privilege to access sensitive information but has the privilege to access the sink. The apps will then collude to exfiltrate the sensitive information to the sink. We decided to implement some test apps as opposed to using real applications from Google play store because there are not many known applications that exhibit the behavior we are interested in testing. Most malicious applications already have sufficient privileges and abuse them as opposed to collude or using a confused deputy attack. As a side note, there is final year report by Dnyaneshwar [2017], and undergraduate student at Nanyang Technological University, which tests LogicDroid on some actual malware applications.

In Table 2.1 and Figure 2.4, we provide some measurement of the timing of the calls between applications. The policy numbers in Table 2.1 refer to the policies in

Table 2.1: Performance Table (ms)

| Policy | Uncached | Cached |
|---|---|---|
| 1 | 76.64 | 14.36 |
| 2 | 93.65 | 42.36 |
| 3 | 94.68 | 41.83 |
| 4 | 92.43 | 42.75 |
| No Monitor | 75.8 | 16.9 |

Table 2.2: Memory Overhead Table

| Policy | Size(kB) | Overhead(%) |
|---|---|---|
| 1 | 372 | 0.05 |
| 2 | 916 | 0.11 |
| 3 | 916 | 0.11 |
| 4 | 916 | 0.11 |

Note: on emulator with 49 apps and overall memory of around 800 mB

Section 2.5. To measure the average time for each ICC call, we construct a chain of ten apps, making successive calls between them, and measure the time needed for one end to reach the other. We measure two different average timings in milliseconds (ms) for different scenarios, based on whether the apps are in the background cache (i.e., suspended) or not. We can see this effect in the case where there is no monitoring involved, in that there is a big difference between the cached or not. This discrepancy in time is due to the fact that there is an overhead involved in starting up the 10 apps.

We also measure the time spent on the monitor actually processing the event, which is around 1 ms for policy 1, and around 10 ms for the other three policies. This shows that the time spent in processing the event is quite low, but more overhead comes from the space required to process the event (there is a big jump in overall timing from simple rules of policy 1 with at most 2 free variables to policy 2 and 3 with 3 free variables). Figure 2.4 shows that the timing of calls over time for each policy is roughly the same. This backs our claim that even though our monitor implements history-based access control, its performance does not depend on the size of the history. Table 2.2 shows the memory footprints of the security monitors. The first column in the table shows the actual size of the memory required by each monitor, and the second column shows the percentage of the memory of each monitor relative to the overall available memory. As can be seen from the table, the memory overhead of the monitors is negligible.

### 2.6.4 Vulnerabilities in com.android.phone component

We now discuss how the added security mechanism of LogicDroid allows us to prevent apps from exploiting recently discovered vulnerabilities in the com.android.phone component in the Android framework. We would like to emphasize that the LogicDroid security mechanism was designed before the discovery of the vulnerabilities, and it was not designed specifically to counter those vulnerabilities. The fact that it works shows the benefit of runtime verification in general and our security extension in particular.

The vulnerabilities in question were first made public by CureSec.com on 4th July 2014, although the vulnerabilities themselves were known to them in late 2013. The fragment of code that contains these vulnerabilities is shown in Figure 2.5. These vulnerabilities are also known as "CVE-2013-6272 com.android.phone" in the database of vulnerabilities maintained by the MITRE Corporation.

The bugs in the codes in Figure 2.5 are essentially a privilege escalation bug in Android official component that enables a user to do three privileged actions without having the privilege to do so, namely, sending SMS, making unrestricted phone calls, and terminating on-going calls. The second bug allows a user to essentially gains the CALL_PRIVILEGED permission through the system component `com.android.phone`. Android documentation states that the CALL_PRIVILEGED permission "allows an application to call any phone number, including emergency numbers, without going through the Dialer user interface for the user to confirm the call being placed." Thus the bug would allow a malicious app to call any number in a stealth mode without notifying the user nor requiring the user's actions. This is the more interesting bug of the three mentioned above, so we used this as a case study to test the effectiveness of LogicDroid.

The bug is caused by a *broadcast receiver* component in `PhoneApp.java` in the Android framework (different versions of Android may have this in a different file). To be precise, the name of the component is NotificationBroadcastReceiver. The component listens to the intents broadcast by apps, and upon receiving an intent, if the action of the intent is ACTION_CALL_BACK_FROM_NOTIFICATION, then this component will start the activity that can handle CALL_PRIVILEGED. See the highlighted parts in Figure 2.5. Privilege escalation occurs in this setting because the component that broadcast the intent in the first place does not necessarily have the permission to do CALL_PRIVILEGED, and the receiving component in `Phone.java` does not enforce that the caller must have the right permission. This is a typical confused deputy attack we mentioned earlier. An interesting fact about this bug is that the component was not meant to be exported, and the developer already noted so in the comment in the class definition. Nevertheless, as opposed to writing "android:exported=false" in the manifest, they wrote "exported=false", which rendered the property to be ineffective hence the bug surfaces.

Thus a general security policy that prevents privilege escalation would stop the

exploit. We specify such a policy in LogicDroid as the following RMTL formula:

$$\exists_x. \quad (trans(x, CALL\_PRIVILEGED) \wedge \neg system(x) \wedge \ hasCallPrivilegedPermission(x))$$

where the *trans* predicate is as defined earlier in Section 2.3. In general, LogicDroid supports such policies for detecting indirect access to a *sink*, i.e., a particular permission we would like to protect, such as INTERNET, READ_CONTACT, SEND_SMS, etc. Note that policy in LogicDroid specifies the patterns that are not allowed in the system calls. In this case, the policy indicates that a forbidden access pattern is one in which there is an indirect call from a non-system application $x$ with no CALL_PRIVILEGED permission to the sink that handles the CALL_PRIVILEGED.

We have tested our runtime monitor against this particular exploit. CureSec provides a prove-of-concept app (called "Kolme") that demonstrates this exploit We ran this app on our modified Android, which is still at version 4.1.1, and so still contains the vulnerable code. LogicDroid successfully detected the attempt by Kolme app to gain CALL_PRIVILEGED permission and stops the attempt.

**Algorithm 3** $Iter(\rho, i, \phi, prev, cur, mprev, mcur)$

**Require:** $i > 1$.
  $prev := cur$; $mprev := mcur$;
  **for** $k = 1$ to $m$ **do** $mcur[k] := 0$; **end for**
  **for** $k = 1$ to $m$ **do**
    **switch** $(\phi_k)$
    **case** $(\bot)$**:** $cur[k] := false$;
    **case** $(p(\vec{c}))$**:** $cur[k] := p(\vec{c}) \in \pi_i$;
    **case** $(\neg\psi)$**:** $cur[k] := \neg cur[idx(\psi)]$;
    **case** $(P(\vec{c}))$**:** $cur[k] := cur[idx(\phi_P(\vec{c}))]$; {Suppose $P(\vec{x}) := \phi_P(\vec{x})$.}
    **case** $(\psi_1 \vee \psi_2)$**:** $cur[k] := cur[idx(\psi_1)] \vee cur[idx(\psi_2)]$;
    **case** $(\bullet\psi)$**:** $cur[k] := prev[idx(\psi)]$;
    **case** $(\diamondsuit\psi)$**:** $curr[k] := prev[idx(\psi)] \vee prev[\diamondsuit\psi]$;
    **case** $(\psi_1 \mathbin{\mathbb{S}} \psi_2)$**:** $cur[k] := cur[idx(\psi_2)] \vee (cur[idx(\psi_1)] \wedge prev[k])$;
    **case** $(\bullet_n\psi)$**:** $cur[k] := prev[\psi] \wedge (\tau_i - \tau_{i-1} < n)$;
    **case** $(\diamondsuit_n\psi)$**:**
      $l := prev[idx(\psi)] \wedge (\tau_i - \tau_{i-1} < n)$;
      $r := prev[idx(\diamondsuit_n\psi)] \wedge (n - (\tau_i - \tau_{i-1}) \geq mprev[k]))$;
      $cur[k] := l \vee r$;
      **if** $l$ **then** $mcur[k] := \tau_i - \tau_{i-1} + 1$;
      **else if** $r$ **then** $mcur[k] := mprev[k] + \tau_i - \tau_{i-1}$;
      **else** $mcur[k] := 0$;
      **end if**
    **case** $(\psi_1 \mathbin{\mathbb{S}_n} \psi_2)$**:**
      $l := cur[idx(\psi_2)]$;
      $r := cur[idx(\psi_1)] \wedge prev[k] \wedge (n - (\tau_i - \tau_{i-1}) \geq mprev[k])$;
      $cur[k] := l \vee r$;
      **if** $l$ **then** $mcur[k] := 1$;
      **else if** $r$ **then** $mcur[k] := mprev[k] + \tau_i - \tau_{i-1}$;
      **else** $mcur[k] := 0$;
      **end if**
    **end switch**
  **return** $cur[idx(\phi)]$;

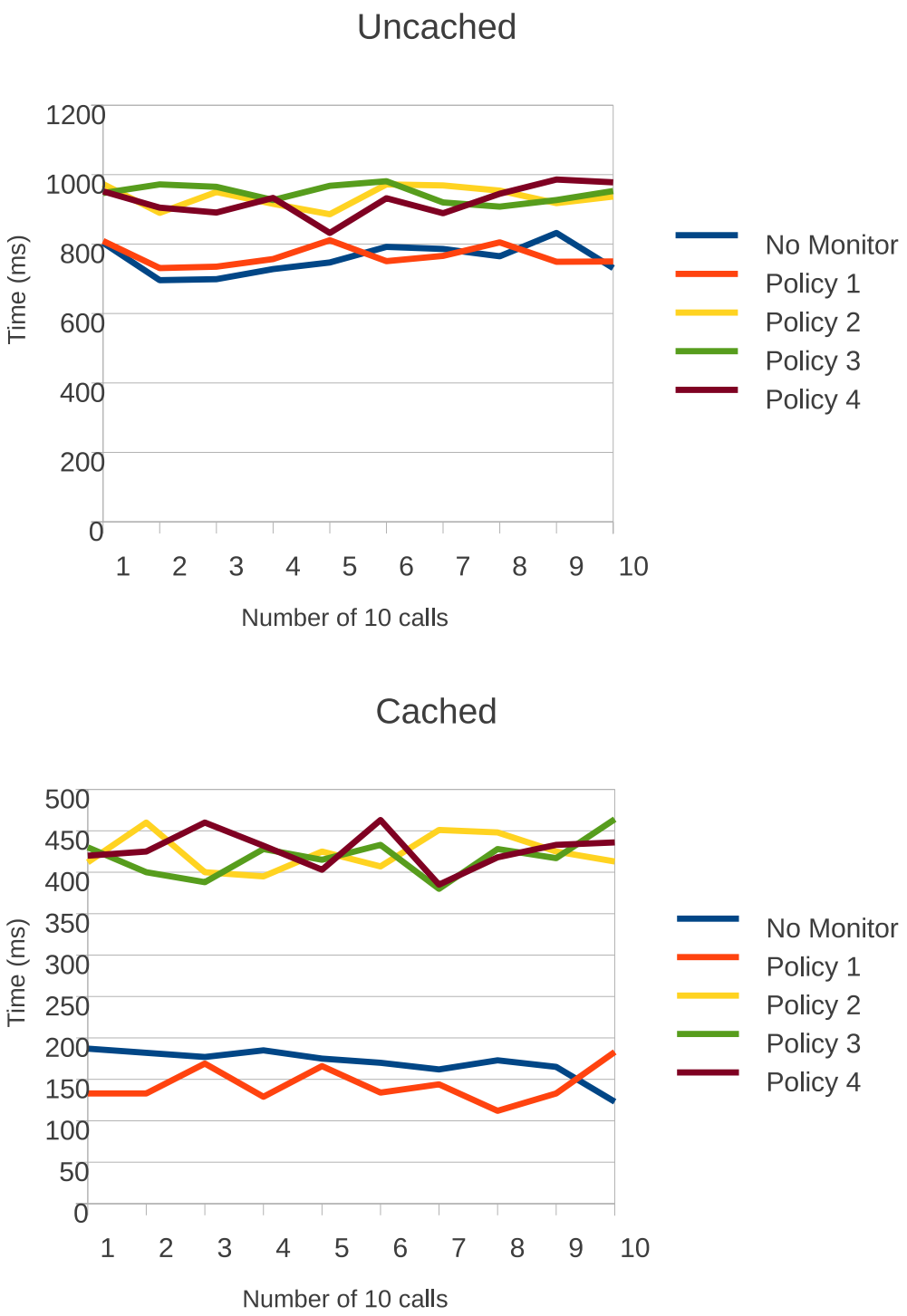



Figure 2.4: Timing of Calls

```
public static class NotificationBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        // TODO: use "if (VDBG)" here.
        Log.d(LOG_TAG, "Broadcast from Notification: " + action);

        if (action.equals(ACTION_HANG_UP_ONGOING_CALL)) {
            PhoneUtils.hangup(PhoneApp.getInstance().mCM);
        } else if (action.equals(ACTION_CALL_BACK_FROM_NOTIFICATION)) {
            // Collapse the expanded notification and the notification item itself.
            closeSystemDialogs(context);
            clearMissedCallNotification(context);

            Intent callIntent = new Intent(Intent.ACTION_CALL_PRIVILEGED,
                intent.getData());
            callIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK
                    | Intent.FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS);
            context.startActivity(callIntent);
        } else if (action.equals(ACTION_SEND_SMS_FROM_NOTIFICATION)) {
            // Collapse the expanded notification and the notification item itself.
            closeSystemDialogs(context);
            clearMissedCallNotification(context);

            Intent smsIntent = new Intent(Intent.ACTION_SENDTO, intent.getData());
            smsIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            context.startActivity(smsIntent);
        } else {
            Log.w(LOG_TAG, "Received hang-up request from notification,"
                    + " but there's no call the system can hang up.");
        }
    }
```

Figure 2.5: Vulnerabilities in the android.com.phone component.

Chapter 3

# Design of Android Bytecode Certification

Android applications are downloaded in the form of an APK. Contained in each of these APKs is a DEX file containing specific instructions (see DEX [2016]) to be executed by the Dalvik VM, so from here on we will refer to these bytecode instructions as DEX instructions. The Dalvik VM is a register-based VM, unlike the Java Virtual Machine (JVM) which is a stack-based VM. Dalvik is now superseded by A new runtime framework called ART, which compiles a DEX bytecode into device specific executable, now supersede Dalvik. This move does not affect our analysis since both Dalvik and ART use the same DEX instructions.

We aim at providing a framework for constructing trustworthy apps, where developers of apps can provide guarantees that the apps will not leak the (sensitive) information the apps used outside the device without the user's consent. The framework should also provide a means for the end-user to verify that apps constructed using the framework adhere to their advertised security policies. This is, of course, not a new concept, and it is essentially a rehash of the (foundational) proof-carrying code (PCC) (Necula [1997]; Appel [2001]), applied to the Android setting. We follow a type-based approach for restricting information flow by Sabelfeld and Myers [2003] in Android apps. Semantically, information flow properties of apps are specified via a notion of non-interference coined by Goguen and Meseguer [1982]. In this setting, typeable programs are guaranteed to be non-interferent, with respect to a given policy, and typing derivations serve as certificates of non-interference. Our eventual goal is to produce a compiler toolchain that can help developers to develop Android applications that comply with a given policy, and automate the process of generating the final non-interference certificates for DEX bytecode.

An Android application is typically written in Java and compiled to Java classes (JVM bytecode). Then using tools provided in the Android Software Development Kit (SDK), these Java classes are further compiled into an Android application in the form of an APK. One important tool in this compilation chain is the dx tool, which aggregates the Java classes and produces a DEX file to be bundled together with other resource files in the APK. Non-interference type systems exist for Java source code (Banerjee and Naumann [2005]), JVM (Barthe et al. [2013]) and (abstracted)

DEX bytecode without exception handling mechanism (Lortz et al. [2014a]). To build a framework that allows end-to-end certificate production, one needs to study certificate translation between these different type systems. The connection between Java and JVM type systems for non-interference has been studied by Barthe et al. [2006a]. In this work, we fill the gap by showing that the connection between JVM and DEX type systems. Our contributions are the following:

- We provide proof that our type system for DEX is non-interferent. On the surface, it may look like that the proof of our type system soundness is taken directly from Barthe et al. [2006b] and Lortz et al. [2014b] but it is not the case. The proof of our type system soundness is substantially different from their proof. Section 4.2 details the soundness proof of our type system, and also outline how our proof is different from Barthe et al. [2006b] and Lortz et al. [2014b]. We also provided a Coq formalization of the non-interference proof for a subset of DEX that concerns arithmetic, control flow, and object manipulation operations.
- We give a formal account of the compilation process from JVM bytecode to DEX bytecode as implemented in the official dx tool in Android SDK. Section 6.1 details some of the translation processes.
- We provide proof that the translation from JVM to DEX preserves typability. That is, JVM programs typable in the non-interference type system for JVM translates into typable programs in the non-interference type system for DEX.

## 3.1 Related Work

Our work is heavily influenced by the work of Barthe et al. [2006b, 2013] on enforcing non-interference via type systems. We discuss other related work in the following.

The closest to our work is the Cassandra project Lortz et al. [2014a,b], that aims at developing certified app stores, where apps can be certified, using an information-flow type system similar to ours, for the absence of specific information flow. Specifically, the authors of Lortz et al. [2014a,b] have developed an abstract Dalvik language (ADL), similar to Dalvik bytecode, and a type system for enforcing non-interference properties for ADL. In Cassandra, they use a client / server infrastructure where the client can submit an app to check whether it satisfies the client's requirements. Our type system for Dalvik has many similarities with that of Cassandra, but one main difference is that we consider a larger fragment of Dalvik, which includes exception handling, something that is not present in Cassandra. We choose to deal directly with Dalvik rather than an abstracted language since we aim to eventually integrate our certificate compilation into existing compiler toolchains for Android apps, without having to modify those toolchains. Our proposed infrastructure is also substantially different from client / server model of Cassandra, where we design a framework for developers to provide non-interference guarantee for their application in the form of a certificate which user can check in their own phone. See Section 6.3 for more detail.

Bian et al. [2007] target the JVM bytecode to check whether a program has the non-interference property. Differently from Barthe et al. their approach uses the idea of the compilation technique where they analyze a variable in the bytecode for its definition and usage. Using this dependence analysis, their tool can detect whether a program leaks confidential information. This is an interesting technique in itself, and it is possible to adopt their approach to analyze DEX bytecode. Nevertheless, we are more interested in the transferability of properties instead of the technique in itself. In particular, if we were to use their approach instead of a type system, the question we are trying to answer would become "if the JVM bytecode is non-interferent according to their approach, is the compiled DEX bytecode also non-interferent?".

In the case of preservation of properties itself, the idea that a non-optimizing compiler preserves a property is not something new. The work by Barthe et al. [2006b] shows that with a non-optimizing compiler, the proof obligation from a source language to a simple stack-based language will be the same, thus allowing the reuse of the proof for the proof obligation in the source language. In showing the preservation of a property, they introduce the source imperative language and target language for a stack-based abstract machine. This is the main difference with our work where we are analyzing the actual dx tool from Android which compiles the bytecode language for the stack-based virtual machine (JVM bytecode) to the actual language for the register-based machine (DEX bytecode). There are also works that address this non-interference preservation from Java source code to JVM bytecode (Barthe et al. [2006a]). Our work can then be seen as a complement to their work in that we are extending the type preservation to include the compilation from JVM bytecode to DEX bytecode.

To deal with information flow properties in Android, there are several works addressing the problem (see Fuchs et al. [2009]; Bugliesi et al. [2013]; Enck et al. [2014]; Zhao and Osono [2012]; Kim et al. [2012]; Fragkaki et al. [2012]; Jia et al. [2013]; Felt et al. [2011]; Enck et al. [2009a,b]) although some of them are geared towards the privilege escalation problem. These works base their context of Android security studied by Enck et al. [2009b]. The tool in the study, which is called Kirin, is also of great interest for us since they deal with the certification of Android applications. Kirin is a lightweight tool which certifies an Android application at install time based on permissions requested. Some of these works are similar to ours in a sense working on the static analysis for Android. The closest one to mention is ScanDroid by Fuchs et al. [2009], with the underlying type system and static analysis tool for security specification in Android by Chaudhuri [2009]. Then along the line of type systems, there is also work by Bugliesi et al. called Lintent that tries to address non-interference on the source code level (Bugliesi et al. [2013]). The main difference with what we do lies in that the analysis itself relies on the existence of the source (the JVM bytecode for ScanDroid and Java source code for Lintent) from which the DEX program is translated.

There are some other static analysis tools for Android which do not stem from the idea of type system, e.g. FlowDroid by Arzt et al. [2014], TrustDroid by Zhao

and Osono [2012] and ScanDal by Kim et al. [2012]. FlowDroid and TrustDroid are another static analysis tool on Android bytecode, trying to prevent information leaking, based on taint analysis on the program. Different from TaintDroid (Enck et al. [2014]) in that they are doing taint analysis statically from decompiled DEX bytecode whereas TaintDroid is enforcing run time taint analysis. ScanDal is also a static analysis for Android applications targetting the DEX instructions directly, aggregating the instructions in a language they call Dalvik Core. They enumerate all possible states and note when any value from any predefined information source is flowing through a predefined information sink. Their work assumed that predefined sources and sinks are given, whereas we are more interested in a flexible policy to define them. For more comprehensive studies on static analysis tools, the interested reader can refer to Reaves et al. [2016]

A work by Backes et al. [2016] called ARTist and a work by Octeau et al. [2012] called Dare are also of great interest. ARTist is a compiler-based app instrumentation. Our current work analyzes the translation from JVM bytecode into DEX bytecode, while ARTist extends *dex2oat*, which produces bytecode for ART from DEX bytecode, with static analysis applicable to the intermediate representation. Their current implementation shows that it is possible to leverage existing approach for information flow (in particular Enck et al. [2014]) by instrumenting this compiler.

Heading to another direction, Dare retarget Android bytecode to Java bytecode via an intermediate representation called Tyde with a high success rate of 99.99%. Although it has the similar motivation of leveraging program analysis tools in Java, essentially what they are doing is different to ours. They are more interested in the semantic preservation of the program as opposed to maintaining the structure of the program. In the case where program analysis of Java bytecode complains about the retargeted Android bytecode, developers have no way to know how the problem comes about in the first place, i.e., the link between the original source code and retargeted Android bytecode is lost. As a side note, there is another static analysis tool called EPICC (Octeau et al. [2013]) which complements Dare nicely. EPICC analyzes application's entry points and exit points and then analyzes possible interactions (thus forming flows of ICC).

Since the property that we are interested in is non-interference, it is also worth mentioning Sorbet, a runtime enforcement of the property by modifying the Android operating system (Fragkaki et al. [2012]; Jia et al. [2013]). Their approach is different from our ultimate goal which motivates this work in that we are aiming for no modification in the Android operating system.

## 3.2 Proof-Carrying Code

Before we proceed to the design of our proposed solution, it is good to step back and have a look at the underlying infrastructure, namely Proof-Carrying Code (PCC) (Necula [1997]). In essence, PCC bundles along proof / certificate about a property which can be easily checked by code consumer, thus moving the burden of proof to

the code producer side. This infrastructure works because it is much easier to check a proof rather than generating one.

With this infrastructure, the trusted computing base becomes really small. In the PCC setting, we do not even need to trust the compiler to produce a correct certificate; we only need to be able to trust the type checker. In the case where the compiler produces an invalid certificate, the type checker will reject the code. Thus, compiler correctness is an independent issue from the perspective of PCC. Quoting from Necula himself "The code receiver does not need to trust the code producer or the proof producer. In other words, the receiver does not have to know the identity of the producer, nor does it have to know anything about the process by which the agent code was produced. All of the information needed for determining the safety of the code is included in the annotated agent code and its proof."

The way the PCC framework is designed is to take the viewpoint of the developer: how can the developer write an app that does the job and convince the client (i.e., the type checker) that the program is safe? If the client insists that programs only use safe features, then the developer will have to comply or risk getting his / her program rejected. So the developer has control to modify the bytecode. This is different from a verification scenario where the code is usually given and can not be changed, i.e., the kind of usage scenarios for anti-virus or malware scanner products.

As a side note, Barthe et al.'s type system is devised in the spirit of PCC. Essentially, the proof is the typing annotations for a particular JVM bytecode. Coupled with the result that their type system is sound, this certificate essentially becomes the proof that the JVM bytecode is safe. In the client side, then, we only need to check whether it is a proper annotation of the program and whether it satisfies the typability relation.

## 3.3 Non-Interferent Type System for JVM

In this section, we give an overview of Barthe et al's type system for JVM. The reader is referred to Barthe et al. [2013] for a more detailed explanation and intuitions behind the design of the type system. Readers who are already familiar with the work of Barthe et al may skip this section.

### 3.3.1 Overview of JVM Bytecode

A program $P$ is given by its list of instructions given in Figure 3.2. The set $\mathcal{X}$ is the set of local variables, $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{null\}$ is the set of values, where $\mathcal{L}$ is an (infinite) set of locations, and $null$ denotes the null pointer, and $\mathcal{PP}$ is the set of program points. For any set $X$, the notation $X^*$ stands for a stack of elements of $X$. Programs are also implicitly parameterized by a set $\mathcal{C}$ of class names, a set $\mathcal{F}$ of field identifiers, a set $\mathcal{M}$ of method names, and a set $\mathcal{T}_J$ of Java types. The instructions listing can be seen in Figure 3.2.

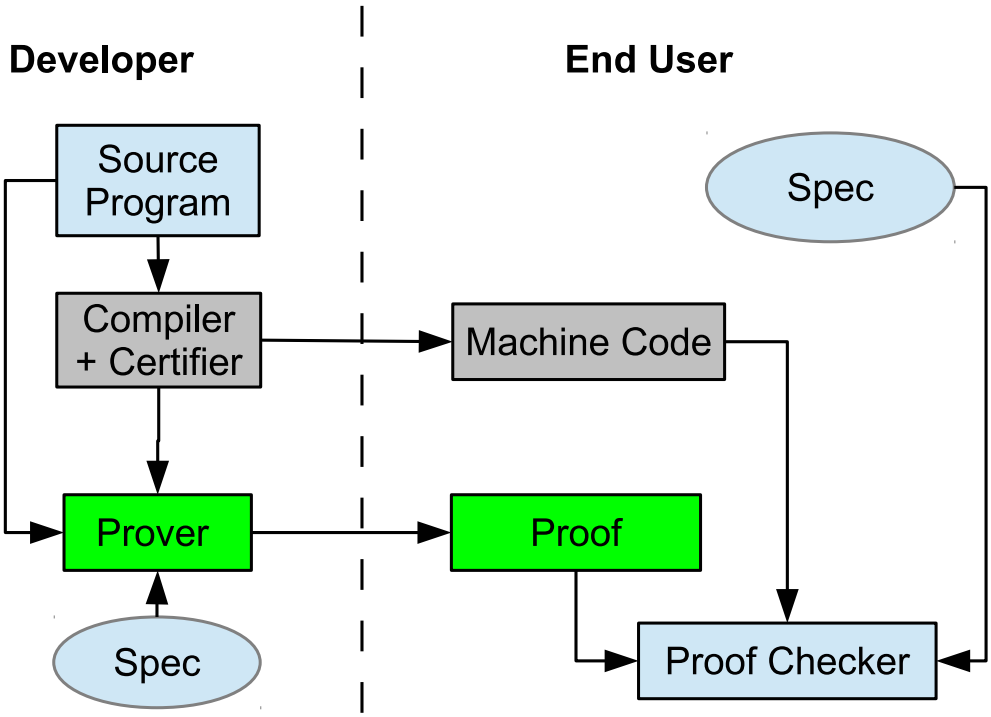



Figure 3.1: PCC structure

### 3.3.2 Operational Semantics

The operational semantics is given as a relation $\rightsquigarrow_{m,\tau} \subseteq \text{State} \times (\text{State} + (\mathcal{V}, \textbf{heap}))$ where $m$ indicates the method under which the relation is considered, and $\tau$ indicates whether the instruction is executing normally (indicated by Norm) or throwing an exception. (sometimes we omit $m$ whenever it is clear which $m$ we are referring to, we may also remove $\tau$ when it is clear from the context whether or not the instruction is executing normally). State here represents a set of JVM states, which is a tuple $\langle i, \rho, os, h \rangle$ where $i \in \mathcal{PP}$ is the program counter that points to the next instruction to be executed; $\rho \in \mathcal{X} \rightharpoonup \mathcal{V}$ is a partial function from local variables to values; $os \in \mathcal{V}^*$ is an operand stack; and $h \in \textbf{heap}$ is the heap for that particular state. Heaps are modeled as partial functions $\mathbf{h} : \mathcal{L} \rightharpoonup \mathcal{O} + \mathcal{A}$, where the set $\mathcal{O}$ of objects is modeled as $\mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V})$, i.e., each object $o \in \mathcal{O}$ possess a class $\textbf{class}(o) \in \mathcal{C}$ and a partial function $o.f \in \mathcal{F} \to \mathcal{V}$ to access the value of field $f$ of object $o$. $\mathcal{A}$ is the set of arrays modeled as $\mathbb{N} \times (\mathbb{N} \rightharpoonup \mathcal{V}) \times \mathcal{PP}$, i.e., each array has a length, partial function from index to value, and a creation point. The creation point will be used to define the notion of array indistinguishability. **Heap** is the set of heaps.

For method invocation, each program comes equipped with a set $\mathcal{M}$ of method names, and for each method $m$, there is associated list $P_m$ of instructions. Each method is identified by method identifier $m_{\text{ID}}$. Therefore we also need to know the class from which this method is invoked; it can be identified by auxiliary function

| | | |
|---|---|---|
| **binop** $op$ | : | do binary operation $op$ on the top two of the stack item and put the resulting value on top of the stack |
| **push** $c$ | : | push a constant value $c$ on top of the stack |
| **pop** | : | pop value from the top of the stack |
| **swap** | : | swap the top two operand stack values |
| **load** $x$ | : | load value of $x$and put it on top of the stack |
| **store** $x$ | : | store the value of the top of the stack in variable $x$ |
| **ifeq** $j$ | : | conditional jump to $j$ if the top of the stack is 0 |
| **goto** $j$ | : | unconditional jump to program point $j$ |
| **return** | : | return from the method with the top value of the stack |
| **new** $C$ | : | create a new object of class $c$ in the heap |
| **getfield** $f$ | : | load value of field $f$and put it on top of the stack |
| **putfield** $f$ | : | store the value of the top of stack in field $f$ |
| **newarray** $t$ | : | create a new array of type $t$ in the heap |
| **arraylength** | : | get the length of an array |
| **arrayload** | : | load value from an array |
| **arraystore** | : | store value in array |
| **invoke** $m_{\mathrm{ID}}$ | : | Invoke method indicated by its method identifier $m_{\mathrm{ID}}$ using the values from the top of the stack as the arguments |
| **throw** | : | Throw an exception object stored at the top of a stack |

where $op \in \{+, -, \times, /\}, c \in \mathbb{Z}, x \in \mathcal{X}, j \in \mathcal{PP}, C \in \mathcal{C}$, $f \in \mathcal{F}, t \in \mathcal{T}_J$, and $m_{\mathrm{ID}} \in \mathcal{M}$.

Figure 3.2: JVM Instruction List

$\mathbf{lookup_P}$ which returns the precise method to be executed based on the method identifier and class.

The program also comes equipped with a partial function $\mathbf{Handler_m} : \mathcal{PP} \times \mathcal{C} \rightharpoonup \mathcal{PP}$. We write $\mathbf{Handler_m}(i, C) = t$ for an exception of class $C \in \mathcal{C}$ thrown at program point $i$, which will be caught by a handler with its starting program point $t$. In the case where the exception is uncaught, we write $\mathbf{Handler_m}(i, C) \uparrow$ instead. The final states will be $(\mathcal{V} + \mathcal{L}) \times \mathbf{Heap}$ to differentiate between normal termination $(v, h) \in \mathcal{V} \times \mathbf{Heap}$, and an uncaught exception $(\langle l \rangle, h) \in \mathcal{L} + \mathbf{Heap}$ which contains the location $l$ for the uncaught exception thrown in the heap $h$. The symbol $\langle\rangle$ is used to identify that the return value is the result of an uncaught exception as opposed to the normal return value.

The notation $\underline{op}$ denotes here the standard interpretation of arithmetic operation of $op$ in the $\overline{\text{d}}$omain $\mathcal{V}$ of values (although there is no arithmetic operation on locations). The operator $\oplus$ denotes the function where $\rho \oplus \{r \mapsto v\}$ means a new function $\rho'$ such that $\forall i \in \mathbf{dom}(\rho) \backslash \{r\}. \rho'(i) = \rho(i)$ and $\rho'(r) = v$. The operator $\oplus$ is overloaded to also mean the update of a field on an object or update on a heap. $\mathbf{nbArguments} : \mathcal{M} \to \mathcal{Z}$ is a function from a method id to its required number of arguments.

The function $\mathbf{fresh} : \text{Heap} \to \mathcal{L}$ is an allocator function that given a heap returns the location for that object. The function $\mathbf{default} : \mathcal{C} \to \mathcal{O}$ returns for each class a default object of that class. For every field of that default object, the value will be 0 if the field is of numeric type, and *null* if the field is of an object type. Similarly for $\mathbf{defaultArray} : \mathbb{N} \times \mathcal{T}_J \to (\mathbb{N} \rightharpoonup \mathcal{V})$. The $\rightsquigarrow$ relation which defines transitions between state is $\rightsquigarrow \subseteq \mathbf{State} \times (\mathbf{State} + \mathcal{V} \times \mathbf{Heap})$.

To handle exception, a program will also come equipped with two parameters **classAnalysis** and **excAnalysis**. **classAnalysis** contains information on possible classes of exception of a program point, and **excAnalysis** contains possible escaping exception of a method.

The instructions that may throw an exception primarily are method invocation and the object / array manipulation instructions. The transitions are also parameterized by a tag $\tau \in \{\text{Norm}\} + \mathcal{C}$ to describe whether the transition occurs normally or some exception is thrown. The notation np is used as the class for null pointer exceptions, with the associated exception handler being **RuntimeExceptionHandling**. **RuntimeExceptionHandling** models the runtime exception handling mechanism in the JVM, where it will continue the execution from the exception handler if such handler is defined, or break the execution and throw an uncaught exception in the case where no handler for its particular exception is defined. There are other possible runtime exceptions for example array index out of bound exception, out of memory exception, etc., but we only focus on null pointer exception instead as it is easy to extend the class and the operational semantics with the other exceptions. As a side note, some readers may realize that in the case of object / array instructions it looks like it is possible to have an instruction where the object pointer is not valid (or the object is not in the heap). Fortunately for us, this case is already handled by the JVM bytecode verifier, which we assume is already set in place.

$$\frac{P_m[i] = \textbf{push}\ n}{\langle i, \rho, os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, n :: os, h\rangle} \qquad \frac{P_m[i] = \textbf{load}\ x}{\langle i, \rho, os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, \rho(x) :: os, h\rangle}$$

$$\frac{P_m[i] = \textbf{ifeq}\ j \quad n \neq 0}{\langle i, \rho, n :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, os, h\rangle} \qquad \frac{P_m[i] = \textbf{ifeq}\ j \quad n = 0}{\langle i, \rho, n :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle j, \rho, os, h\rangle}$$

$$\frac{P_m[i] = \textbf{binop}\ op \quad n_2\ \underline{op}\ n_1 = n}{\langle i, \rho, n_1 :: n_2 :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, n :: os, h\rangle} \qquad \frac{P_m[i] = \textbf{return}}{\langle i, \rho, v :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} v, h}$$

$$\frac{P_m[i] = \textbf{swap}}{\langle i, \rho, v_1 :: v_2 :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, v_2 :: v_1 :: os, h\rangle} \qquad \frac{P_m[i] = \textbf{goto}\ j}{\langle i, \rho, os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle j, \rho, os, h\rangle}$$

$$\frac{P_m[i] = \textbf{store}\ x \quad x \in \textbf{dom}(\rho)}{\langle i, \rho, v :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{x \mapsto v\}, os, h\rangle} \qquad \frac{P_m[i] = \textbf{pop}}{\langle i, \rho, v :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, os, h\rangle}$$

$$\frac{P_m[i] = \textbf{getfield}\ f \quad l \in \textbf{dom}(h) \quad f \in \textbf{dom}(h(l))}{\langle i, \rho, l :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, h(l).f :: os, h\rangle}$$

$$\frac{P_m[i] = \textbf{new}\ C \quad l = \textbf{fresh}(h)}{\langle i, \rho, os, h\rangle \rightsquigarrow \langle i+1, \rho, l :: os, h \oplus \{l \mapsto \textbf{default}(C)\}\rangle}$$

$$\frac{P_m[i] = \textbf{getfield}\ f \quad l' = \textbf{fresh}(h)}{\langle i, \rho, null :: os, h\rangle \rightsquigarrow_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{P_m[i] = \textbf{putfield}\ f \quad l \in \textbf{dom}(h) \quad f \in \textbf{dom}(h(l))}{\langle i, \rho, v :: l :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, os, h \oplus \{l \mapsto h(l) \oplus \{f \mapsto v\}\}\rangle}$$

$$\frac{P_m[i] = \textbf{putfield}\ f \quad l' = \textbf{fresh}(h)}{\langle i, \rho, v :: null :: os, h\rangle \rightsquigarrow_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{P_m[i] = \textbf{newarray}\ t \quad l = \textbf{fresh}(h) \quad n \geq 0}{\langle i, \rho, n :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, l :: os, h \oplus \{l \mapsto (n, \textbf{defaultArray}(n, t), i)\}\rangle}$$

$$\frac{P_m[i] = \textbf{arraylength} \quad l \in \textbf{dom}(h)}{\langle i, \rho, l :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, h(l).\textbf{length} :: os, h\rangle}$$

$$\frac{P_m[i] = \textbf{arraylength} \quad l' = \textbf{fresh}(h)}{\langle i, \rho, null :: os, h\rangle \rightsquigarrow_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{P_m[i] = \textbf{arrayload} \quad l \in \textbf{dom}(h) \quad 0 \leq j < h(l).\textbf{length}}{\langle i, \rho, j :: l :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, h(l)[j] :: os, h\rangle}$$

$$\frac{P_m[i] = \textbf{arrayload} \quad l' = \textbf{fresh}(h)}{\langle i, \rho, j :: null :: os, h\rangle \rightsquigarrow_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{P_m[i] = \textbf{arraystore} \quad l \in \textbf{dom}(h) \quad 0 \leq j < h(l).\textbf{length}}{\langle i, \rho, v :: j :: l :: os, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, os, h \oplus \{l \mapsto h(l) \oplus \{j \mapsto v\}\}\rangle}$$

$$\frac{P_m[i] = \textbf{arraystore} \quad l' = \textbf{fresh}(h)}{\langle i, \rho, v :: j :: null :: os, h\rangle \rightsquigarrow_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{\begin{array}{c} Pm[i] = \textbf{invoke}\ m_{\text{ID}} \quad m' = \textbf{lookup}_{\mathbf{P}}(m_{\text{ID}}, \textbf{class}(h(l))) \quad l \in \textbf{dom}(h) \\ \textbf{length}(os_1) = \textbf{nbArguments}(m_{\text{ID}}) \quad \langle 1, \{this \mapsto l, \vec{x} \mapsto os_1\}, \epsilon, h\rangle \rightsquigarrow^{+}_{m'} v, h' \end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, v :: os_2, h'\rangle}$$

$$\frac{\begin{array}{c} P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad m' = \textbf{lookup}_{\mathbf{P}}(m_{\text{ID}}, \textbf{class}(h(l))) \\ \langle 1, \{this \mapsto l, \vec{x} \mapsto os_1\}, \epsilon, h\rangle \rightsquigarrow^{+}_{m'} \langle l'\rangle, h' \quad l \in \textbf{dom}(h) \\ \textbf{Handler}_{\mathbf{m}}(i, e) = t \quad e = \textbf{class}(h'(l')) \quad e \in \textbf{excAnalysis}(m_{\text{ID}}) \end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \rightsquigarrow_{m,e} \langle t, \rho, l' :: \epsilon, h'\rangle}$$

$$\frac{P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad l' = \textbf{fresh}(h)}{\langle i, \rho, os_1 :: null :: os_2, h\rangle \rightsquigarrow_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{\begin{array}{c} P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad m' = \textbf{lookup}_{\mathbf{P}}(m_{\text{ID}}, \textbf{class}(h(l))) \\ \langle 1, \{this \mapsto l, \vec{x} \mapsto os_1\}, \epsilon, h\rangle \rightsquigarrow^{+}_{m'} \langle l'\rangle, h' \quad l \in \textbf{dom}(h) \\ e = \textbf{class}(h'(l')) \quad \textbf{Handler}_{\mathbf{m}}(i, e) \uparrow \quad e \in \textbf{excAnalysis}(m_{\text{ID}}) \end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \rightsquigarrow_{m,e} \langle l'\rangle, h'}$$

$$\frac{P_m[i] = \textbf{throw} \quad l' = \textbf{fresh}(h)}{\langle i, \rho, null :: os, h\rangle \rightsquigarrow_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{\begin{array}{c} P_m[i] = \textbf{throw} \quad l \in \textbf{dom}(h) \quad e = \textbf{class}(h(l)) \\ \textbf{Handler}_{\mathbf{m}}(i, e) = t \quad e \in \textbf{classAnalysis}(m, i) \end{array}}{\langle i, \rho, l :: os, h\rangle \rightsquigarrow_{m,e} \langle t, \rho, l :: \epsilon, h\rangle}$$

$$\frac{\begin{array}{c} P_m[i] = \textbf{throw} \quad l \in \textbf{dom}(h) \quad e = \textbf{class}(h(l)) \\ \textbf{Handler}_{\mathbf{m}}(i, e) \uparrow \quad e \in \textbf{classAnalysis}(m, i) \end{array}}{\langle i, \rho, l :: os, h\rangle \rightsquigarrow_{m,e} \langle l\rangle, h}$$

$\textbf{RuntimeExcHandling} : \textbf{Heap} \times \mathcal{L} \times \mathcal{C} \times \mathcal{PP} \times (\mathcal{X} \rightharpoonup \mathcal{V}) \to \textbf{State} + (\mathcal{L} \times \textbf{Heap})$
defined as

$$\textbf{RuntimeExcHandling}(h, l', C, i, \rho) = \begin{cases} \langle t, \rho, l' :: \epsilon, h \oplus \{l' \mapsto \textbf{default}(C)\}\rangle & \text{if } \textbf{Handler}_{\mathbf{m}}(i, C) = t \\ \langle l'\rangle, h \oplus \{l' \mapsto \textbf{default}(C)\} & \text{if } \textbf{Handler}_{\mathbf{m}}(i, C) \uparrow \end{cases}$$

Figure 3.3: Full JVM Operational Semantic

Some last remarks: firstly, because of method invocation, the operational semantics will also be mixed with a big step semantics style $\rightsquigarrow_m^+$ from method invocations of method $m$ and its associated result. To be more precise, $\rightsquigarrow_m^+$ is the transitive closure of $\rightsquigarrow_m$. Then, for instructions that may not throw an exception, we remove the subscript $\{m, \text{Norm}\}$ from $\rightsquigarrow$ because it is clear that they have no exception throwing operational semantic counterpart. A list of operational semantics is contained in Figure 3.3.

**Successor Relation** The successor relations $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ of a program $P$ are tagged with whether the execution is normal or throwing an exception. Whenever it is clear from the context, we will drop the tag to reduce clutter. According to the types of instructions at program point $i$, there are several possibilities:

$P_m[i]$ = **goto** $t$: The successor relation is $i \mapsto^{\text{Norm}} t$

$P_m[i]$ = **ifeq** $t$: In this case, there are 2 successor relations $i \mapsto^{\text{Norm}} i+1$ and $i \mapsto^{\text{Norm}} t$.

$P_m[i]$ = **return**: In this case, it is a return point denoted by $i \mapsto^{\text{Norm}}$

$P_m[i]$ is an instruction throwing a null pointer exception, and there is a handler for it ($\textbf{Handler}(i, \text{np}) = t$). In this case, the successor of $i$ is $t$, denoted by $i \mapsto^{\text{np}} t$.

$P_m[i]$ is an instruction throwing a null pointer exception, and there is no handler for it ($\textbf{Handler}(i, \text{np}) \uparrow$). In this case, it is a return point denoted by $i \mapsto^{\text{np}}$.

$P_m[i]$ = **throw**, throwing an exception $C \in \textbf{classAnalysis}(m, i)$, and the handler is $\textbf{Handler}(i, C) = t$. The successor relation is $i \mapsto^C t$.

$P_m[i]$ = **throw**, throwing an exception $C \in \textbf{classAnalysis}(m, i)$, and the handler is $\textbf{Handler}(i, C) = \uparrow$. It is a return point, and the successor relation is $i \mapsto^C$.

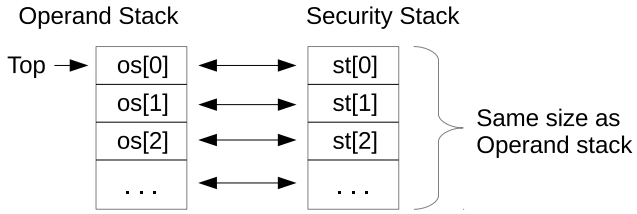
$P_m[i]$ = **invoke** $m_{\text{ID}}$, throwing an exception $C \in \textbf{excAnalysis}(m_{\text{ID}})$, and the handler is $\textbf{Handler}(i, C) = t$. The successor relation is $i \mapsto^C t$.

$P_m[i]$ = **invoke** $m_{\text{ID}}$, throwing an exception $C \in \textbf{excAnalysis}(m_{\text{ID}})$, and the handler is $\textbf{Handler}(i, C) \uparrow$. It is a return point, and the successor relation is $i \mapsto^C$.

$P_m[i]$ is any other case. The successor is the next instruction in the program, denoted by $i \mapsto^{norm} i+1$

### 3.3.3 Type System

Security levels are given by a lattice $(\mathcal{S}, \leq)$ where $\sqcup$ denotes the lub of two security levels, and for every $k \in \mathcal{S}$, $\textbf{lift}_\textbf{k}$ is a point-wise extension to stack types of $\lambda l.k \sqcup l$. The policy of a method is also defined relative to a security level $k_{\text{obs}}$ which denotes the capability of an observer to observe values from local variables, fields, and return values whose security levels are below $k_{\text{obs}}$.The typing rules are defined in terms of stack types; that is a stack that associates a value in the operand stack to the set $\mathcal{S}$ of security levels. The stack type itself takes the form of a stack with corresponding indices from the operand stack, as shown below.




We assume that a method comes with its security policy of the form $\vec{k_a} \xrightarrow{k_h} \vec{k_r}$ where $\vec{k_a}$ represents a list $\{x_1 : k_1, y_2 : k_2, \ldots, x_m : k_m\}$ with $k_i \in \mathcal{S}$ being the security level of local variables $x_i \in \mathcal{X}$ and $k_h$ is the effect of the method on the heap and $\vec{k_r}$ is the return signature, i.e., the security level of the return value. The return signature is in the form of a list to cater for the possibility of an uncaught exception on top of the normal return value. The $\vec{k_r}$ is a list of the form $\{\text{Norm} : k_{\text{Norm}}, e_1 : k_{e_1}, \ldots, e_n : k_{e_n}\}$ where $k_n$ is the security level for the normal return value and $e_i$ is the class of the uncaught exception thrown by the method and $k_{e_i}$ is the associated security level. In the sequel, we write $\vec{k_r}(\text{Norm})$ to stand for $k_{\text{Norm}}$ and $\vec{k_r}(e_i)$ to stand for $k_{e_i}$. An example of this policy can be $\{1 : L, 2 : H\} \xrightarrow{H} \{\text{Norm} : L\}$ where $L, H \in \mathcal{S}, L \leq k_{\text{obs}}, H \not\leq k_{\text{obs}}$ indicating that the method will return a low value and that throughout the execution of the method, the security level of local variable 1 will be low while the security level of local variable 2 will be high.

Compared to the usual object or value, arrays have an extended security level to cater for the security level of the contents. The security level of an array is of the form $k[k_c]$ where $k$ represents the security level of the array, and $k_c$ represents the security level of its content (this implies that all array elements have the same security level $k_c$). Let $\mathcal{S}^{\text{ext}}$ be the extension of security levels $\mathcal{S}$ to define the array's security level. The partial order on $\mathcal{S}$ will also be extended with $\leq^{\text{ext}}$ :

$$\frac{k \leq k' \quad k, k' \in \mathcal{S}}{k \leq^{\text{ext}} k'} \qquad \frac{k \leq k' \quad k, k' \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}}}{k[k_c] \leq^{\text{ext}} k'[k_c]}$$

Generally, in the case of a comparison between extended level $k[k_c] \in \mathcal{S}^{\text{ext}}$ and a standard level $k' \in \mathcal{S}$, we only compare $k$ and $k'$ w.r.t. the partial order on $\mathcal{S}$. In the case of comparison with $k_{\text{obs}}$, since $k_{\text{obs}} \in \mathcal{S}$, an extended security $k[k_c]$ is considered low (written $k[k_c] \leq k_{\text{obs}}$) if $k \leq k_{\text{obs}}$.

The transfer rules come equipped with a security policy for fields $\mathbf{ft} : \mathcal{F} \to \mathcal{S}^{\text{ext}}$ and $\text{at} : \mathcal{PP} \to \mathcal{S}^{\text{ext}}$ that maps the creation point of an array with the security level of its content. $\mathbf{at}(a)$ will also be used to denote the security level of the content of array $a$ at its creation point.

The notation $\Gamma$ is used to define the table of method signatures which will associate a method identifier $m_{\text{ID}}$ and a security level $k \in \mathcal{S}$ (of the object invoked) to a security signature $\Gamma_{m_{\text{ID}}}[k]$. The collection of security signatures of a method $m$ is defined as $\mathbf{Policies}_\Gamma(m_{\text{ID}}) = \{\Gamma_{m_{\text{ID}}}[k] \mid k \in \mathcal{S}\}$.

A method is also parameterized by a control dependence region (CDR) which is defined in terms of two functions: **region** and **jun**. The function $\mathbf{region} : \mathcal{PP} \times \{\text{Norm} \cup \mathcal{C}\} \to \wp(\mathcal{PP})$ can be seen as all the program points executing under the

guard of the instruction at the specified program point, i.e., in the case of $\mathbf{region}(i,\tau)$ the guard will be the program point $i$. The function $\mathbf{jun}(i,\tau)$ itself can be seen as the nearest program point which all instructions in $\mathbf{region}(i,\tau)$ have to execute (junction point). A CDR is safe if it satisfies the following SOAP (Safe Over APproximation) properties.

**Definition 3.3.1.** *A CDR structure (region, jun) satisfies the SOAP properties if the following properties hold :*

**SOAP1.** $\forall i,j,k \in \mathcal{PP}$ *and tag* $\tau$ *if* $i \mapsto j$ *and* $i \mapsto^{\tau} k$ *and* $j \neq k$ *(i is hence a branching point) then* $k \in \mathbf{region}(i,\tau)$ *or* $k = \mathbf{jun}(i,\tau)$.

**SOAP2.** $\forall i,j,k \in \mathcal{PP}$ *and tag* $\tau$*, if* $j \in \mathbf{region}(i,\tau)$ *and* $j \mapsto k$*, then either* $k \in \mathbf{region}(i,\tau)$ *or* $k = \mathbf{jun}(i,\tau)$.

**SOAP3.** $\forall i,j \in \mathcal{PP}$ *and tag* $\tau$*, if* $j \in \mathbf{region}(i,\tau)$ *and* $j$ *is a return point then* $\mathbf{jun}(i,\tau)$ *is undefined.*

**SOAP4.** $\forall i \in \mathcal{PP}$ *and tags* $\tau_1,\tau_2$ *if* $\mathbf{jun}(i,\tau_1)$ *and* $\mathbf{jun}(i,\tau_2)$ *are defined and* $\mathbf{jun}(i,\tau_1) \neq \mathbf{jun}(i,\tau_2)$ *then* $\mathbf{jun}(i,\tau_1) \in \mathbf{region}(i,\tau_2)$ *or* $\mathbf{jun}(i,\tau_2) \in \mathbf{region}(i,\tau_1)$.

**SOAP5.** $\forall i,j \in \mathcal{PP}$ *and tag* $\tau$*, if* $j \in \mathbf{region}(i,\tau)$ *and* $j$ *is a return point then for all tags* $\tau'$ *if* $\mathbf{jun}(i,\tau')$ *is defined then* $\mathbf{jun}(i,\tau') \in \mathbf{region}(i,\tau)$.

**SOAP6.** $\forall i \in \mathcal{PP}$ *and tag* $\tau_1$*, if* $i \mapsto^{\tau_1}$ *then for all tags* $\tau_2, \mathbf{region}(i,\tau_2) \subseteq \mathbf{region}(i,\tau_1)$ *and if* $\mathbf{jun}(i,\tau_2)$ *is defined then* $\mathbf{jun}(i,\tau_2) \in \mathbf{region}(i,\tau_1)$.

The security environment function $se : \mathcal{PP} \to \mathcal{S}$ is a map from a program point to a security level. The notation $\Rightarrow$ represents a relation between the stack type before execution and the stack type after execution of an instruction.

The typing system is formally parameterized by :

**$\Gamma$:** a table of method signatures, needed to define the transfer rules for method invocation;

**ft:** a map from fields to their global policy level;

**CDR:** a structure consisting of (**region**, **jun**).

*se*: a security environment

*sgn*: the method signature of the current method

$S_i$: stack type annotation at program point $i$

*st*: stack typing after the instruction is executed

thus the complete form of a judgment parameterized by a tag $\tau \in \{\mathrm{Norm} + \mathcal{C}\}$ is

$$\Gamma, \mathbf{ft}, \mathbf{region}, se, sgn, i \vdash^{\tau} S_i \Rightarrow st.$$

$$\frac{P_m[i] = \textbf{load}\ x}{se, i \vdash st \Rightarrow \left(\vec{k_a}(x) \sqcup se(i)\right) :: st} \qquad \frac{P_m[i] = \textbf{store}\ x \quad se(i) \sqcup k \leq \vec{k_a}(x)}{se, i \vdash k :: st \Rightarrow st}$$

$$\frac{P_m[i] = \textbf{swap}}{i \vdash k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st} \qquad \frac{P_m[i] = \textbf{push}\ n}{se, i \vdash st \Rightarrow se(i) :: st} \qquad \frac{P_m[i] = \textbf{pop}}{i \vdash k :: st \Rightarrow st}$$

$$\frac{P_m[i] = \textbf{ifeq}\ j \quad \forall j' \in \textbf{region}(i, \mathrm{Norm}), k \leq se(j')}{\textbf{region}, se, i \vdash k :: st \Rightarrow \textbf{lift}_{\mathbf{k}}(st)} \qquad \frac{P_m[i] = \textbf{goto}\ j}{i \vdash st \Rightarrow st}$$

$$\frac{P_m[i] = \textbf{binop}\ op}{se, i \vdash k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2 \sqcup se(i)) :: st} \qquad \frac{P_m[i] = \textbf{return} \quad se(i) \sqcup k \leq \vec{k_r}(n)}{\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash k :: st \Rightarrow}$$

$$\frac{P_m[i] = \textbf{new}\ C}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \textbf{region}, se, i \vdash st \Rightarrow se(i) :: st}$$

$$\frac{P_m[i] = \textbf{newarray}\ t \quad k \in \mathcal{S}}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \textbf{region}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow k[\textbf{at}(i)] :: st}$$

$$\frac{P_m[i] = \textbf{getfield}\ f \quad k \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \mathrm{Norm}), k \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \textbf{region}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow \textbf{lift}_{\mathbf{k}}\left(\left((k \sqcup se(i)) \sqcup^{\mathrm{ext}} \textbf{ft}(f)\right) :: st\right)}$$

$$\frac{P_m[i] = \textbf{getfield}\ f \quad k \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \mathrm{np}), k \leq se(j) \quad \textbf{Handler}(i, \mathrm{np}) = t}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \textbf{region}, se, i \vdash^{\mathrm{np}} k :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \textbf{getfield}\ f \quad k \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \mathrm{np}), k \leq se(j) \\ \textbf{Handler}(i, \mathrm{np}) \uparrow \quad k \leq \vec{k_r}(\mathrm{np}) \end{array}}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \textbf{region}, se, i \vdash^{\mathrm{np}} k :: st \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \textbf{putfield}\ f \quad (se(i) \sqcup k_2) \sqcup^{\mathrm{ext}} k_1 \leq \textbf{ft}(f) \quad k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \\ k_h \leq \textbf{ft}(f) \quad \forall j \in \textbf{region}(i, \mathrm{Norm}), k_2 \leq se(j) \end{array}}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \textbf{region}, se, i \vdash^{\mathrm{Norm}} k_1 :: k_2 :: st \Rightarrow \textbf{lift}_{\mathbf{k_2}}(st)}$$

$$\frac{\begin{array}{c} P_m[i] = \textbf{putfield}\ f \quad (se(i) \sqcup k_2) \sqcup^{\mathrm{ext}} k_1 \leq \textbf{ft}(f) \quad k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \\ \forall j \in \textbf{region}(i, \mathrm{np}), k_2 \leq se(j) \quad \textbf{Handler}(i, \mathrm{np}) = t \end{array}}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \textbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$$

$$\frac{P_m[i] = \textbf{putfield}\ f \quad (se(i) \sqcup k_2) \sqcup^{\text{ext}} k_1 \leq \textbf{ft}(f) \quad k_1 \in \mathcal{S}^{\text{ext}} \quad k_2 \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \text{np}), k \leq se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow \quad k_2 \leq \vec{k_r}(\text{np})}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2 :: st \Rightarrow}$$

$$\frac{P_m[i] = \textbf{arraylength} \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{Norm}} k[k_c] :: st \Rightarrow \textbf{lift}_{\textbf{k}}(k :: st)}$$

$$\frac{P_m[i] = \textbf{arraylength} \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k \leq se(j) \quad \textbf{Handler}(i, \text{np}) = t}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{np}} k[k_c] :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$$

$$\frac{P_m[i] = \textbf{arraylength} \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k \leq se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow \quad k \leq \vec{k_r}(\text{np})}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{np}} k[k_c] :: st \Rightarrow}$$

$$\frac{P_m[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{Norm}), k_2 \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{Norm}} k_1 :: k_2[k_c] :: st \Rightarrow \textbf{lift}_{\textbf{k}_2}(\big((k_1 \sqcup k_2) \sqcup^{\text{ext}} k_c\big) :: st)}$$

$$\frac{P_m[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k_2 \leq se(j) \quad \textbf{Handler}(i, \text{np}) = t}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2[k_c] :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$$

$$\frac{P_m[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k_2 \leq se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow \quad k_2 \leq \vec{k_r}(\text{np})}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2[k_c] :: st \Rightarrow}$$

$$\frac{P_m[i] = \textbf{arraystore} \quad ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \leq^{\text{ext}} k_c \quad k_2, k_3 \in \mathcal{S} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{Norm}), k_2 \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{Norm}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow \textbf{lift}_{\textbf{k}_2}(st)}$$

$$\frac{P_m[i] = \textbf{arraystore} \quad ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \leq^{\text{ext}} k_c \quad k_2, k_3 \in \mathcal{S} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k_2 \leq se(j) \quad \textbf{Handler}(i, \text{np}) = t}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$$

$$\frac{P_m[i] = \textbf{arraystore} \quad ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \leq^{\text{ext}} k_c \quad k_2, k_3 \in \mathcal{S} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k_2 \leq se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow \quad k_2 \leq \vec{k_r}(\text{np})}{\Gamma, \textbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow}$$

$$\frac{P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad \textbf{length}(st_1) = \textbf{nbArguments}(m_{\text{ID}}) \quad \Gamma_{m_{\text{ID}}}[k] = \vec{k'_a} \xrightarrow{k'_h} \vec{k'_r} \quad \forall i \in [0, \textbf{length}(st_1) - 1].st_1[i] \leq \vec{k'_a}[i+1] \quad k \leq \vec{k'_a}[0] \quad k \sqcup k_h \sqcup se(i) \leq k'_h \quad k_e = \bigsqcup \{\vec{k'_r}(e) \mid e \in \textbf{excAnalysis}(m_{\text{ID}})\} \quad \forall j \in \textbf{region}(i, \text{Norm}), k \sqcup k_e \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{Norm}} st_1 :: k :: st_2 \Rightarrow \text{lift}_{k \sqcup k_e}\big((\vec{k'_r}(n) \sqcup se(i)) :: st_2\big)\big)}$$

$$\frac{P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad \textbf{length}(st_1) = \textbf{nbArguments}(m_{\text{ID}}) \quad \Gamma_{m_{\text{ID}}}[k] = \vec{k'_a} \xrightarrow{k'_h} \vec{k'_r} \quad \forall i \in [0, \textbf{length}(st_1) - 1].st_1[i] \leq \vec{k'_a}[i+1] \quad k \leq \vec{k'_a}[0] \quad k \sqcup k_h \sqcup se(i) \leq k'_h \quad e \in \textbf{excAnalysis}(m_{\text{ID}}) \cup \{\text{np}\} \quad \textbf{Handler}(i, e) = t \quad \forall j \in \textbf{region}(i, e), k \sqcup k'_r(e) \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \textbf{region}, se, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow (k \sqcup \vec{k'_r}(e)) :: \epsilon}$$

$$\frac{P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad \textbf{length}(st_1) = \textbf{nbArguments}(m_{\text{ID}}) \quad \Gamma_{m_{\text{ID}}}[k] = \vec{k'_a} \xrightarrow{k'_h} \vec{k'_r} \quad \forall i \in [0, \textbf{length}(st_1) - 1].st_1[i] \leq \vec{k'_a}[i+1] \quad k \leq \vec{k'_a}[0] \quad k \sqcup k_h \sqcup se(i) \leq k'_h \quad k \sqcup se(i) \sqcup \vec{k'_r}(e) \leq \vec{k_r}(e) \quad e \in \textbf{excAnalysis}(m_{\text{ID}}) \cup \{\text{np}\} \quad \textbf{Handler}(i, e) \uparrow \quad \forall j \in \textbf{region}(i, e), k \sqcup k'_r(e) \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \textbf{region}, se, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow}$$

$$\frac{P_m[i] = \textbf{throw} \quad e \in \textbf{classAnalysis}(i) \cup \{\text{np}\} \quad \forall j \in \textbf{region}(i, e), k \leq se(j) \quad \textbf{Handler}(i, e) = t}{\Gamma, \textbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \textbf{region}, se, i \vdash^{e} k :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$$

$$\frac{P_m[i] = \textbf{throw} \quad e \in \textbf{classAnalysis}(i) \cup \{\text{np}\} \quad k \leq \vec{k_r}(e) \quad \forall j \in \textbf{region}(i, e), k \leq se(j) \quad \textbf{Handler}(i, e) \uparrow}{\Gamma, \textbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \textbf{region}, se, i \vdash^{e} k :: st \Rightarrow}$$

Figure 3.4: JVM Transfer Rule

In the case where some elements are unnecessary, we may omit some of the parameters, e.g., $i \vdash S_i \Rightarrow st$.

As in the operational semantics, wherever it is clear that the instructions may not throw an exception, we remove the tag Norm to reduce clutter. The transfer rules are contained in Figure 3.4. Using these transfer rules, we can then define the notion of typability:

**Definition 3.3.2** (Typable method)**.** *A method $m$ is typable w.r.t. a method signature table $\Gamma$, a global field policy $\mathbf{ft}$, a policy $sgn$, and a CDR $\mathbf{region_m} : \mathcal{PP} \to \wp(\mathcal{PP})$ if there exists a security environment $se : \mathcal{PP} \to \mathcal{S}$ and a function $S : \mathcal{PP} \to \mathcal{S}^*$ s.t. $S_1 = \epsilon$ and for all $i, j \in \mathcal{PP}$, and exception tags $e \in \{\text{Norm} + \mathcal{C}\}$:*

*(a) $i \mapsto^e j$ implies there exists $st \in \mathcal{S}^*$ such that $\Gamma, \mathbf{ft}, \mathbf{region}, se, sgn, i \vdash^e S_i \Rightarrow st$ and $st \sqsubseteq S_j$;*

*(b) $i \mapsto^e$ implies $\Gamma, \mathbf{ft}, \mathbf{region}, se, sgn, i \vdash^e S_i \Rightarrow$,*

*where $\sqsubseteq$ denotes the point-wise partial order on type stack w.r.t. the partial order taken on security levels.*

The Non-interference definition relies on the notion of indistinguishability. Loosely speaking, a method is non-interferent if, given indistinguishable inputs, it yields indistinguishable outputs. Obviously, we have to define what it means to be indistinguishable.

To define the notions of location, object, and array indistinguishability, Barthe et al. define the notion of a $\beta$ mapping. $\beta$ is a bijection on (a partial set of ) locations in the heap. The bijection maps low objects (objects whose references might be stored in low fields or variables) allocated in the heap of a state to low objects allocated in the heap of subsequent state. Two objects might be indistinguishable, even if their locations are different during execution. The $\beta$ function is a bijection in the sense that it is a one to one correspondence on all low objects in the heap, but it is also partial in a sense that high objects are not mapped.

**Definition 3.3.3** (Value indistinguishability)**.** *Letting $v, v_1, v_2 \in \mathcal{V}$, and given a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, the relation $\sim_\beta \subseteq \mathcal{V} \times \mathcal{V}$ is defined by the clauses :*

$$null \sim_\beta null \qquad \frac{v \in \mathcal{N}}{v \sim_\beta v} \qquad \frac{v_1, v_2 \in \mathcal{L} \quad \beta(v_1) = v_2}{v_1 \sim_\beta v_2}$$

**Definition 3.3.4** (Local variables indistinguishability)**.** *For $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$, we have $\rho \sim_{k_{\text{obs}}, \vec{k_a}, \beta} \rho'$ if $\rho$ and $\rho'$ have the same domain and if $\vec{k_a}(x) \leq k_{\text{obs}}$ then $\rho(x) \sim_\beta \rho'(x)$ for all $x \in \mathbf{dom}(\rho)$.*

**Definition 3.3.5** (Object indistinguishability)**.** *Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (written as $o_1 \sim_{k_{\text{obs}}, \beta} o_2$) if and only if $o_1$ and $o_2$ are objects of the same class and $o_1.f \sim_\beta o_2.f$ for all fields $f \in \mathbf{dom}(o_1)$ s.t. $\mathbf{ft}(f) \leq k_{\text{obs}}$.*

**Definition 3.3.6** (Array indistinguishability)**.** *Two arrays $a_1, a_2 \in \mathcal{A}$ are indistinguishable w.r.t. an attacker level $k_{\text{obs}}$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (written as $a_1 \sim_{k_{\text{obs}},\beta} a_2$) if and only if $a_1.\textbf{length} = a_2.\textbf{length}$ and, moreover, if $\textbf{at}(a_1) \leq k_{\text{obs}}$, then $a_1[i] \sim_\beta a_2[i]$ for all $i$ such that $0 \leq i < a_1.\textbf{length}$.*

**Definition 3.3.7** (Heap indistinguishability)**.** *Two heaps $h_1$ and $h_2$ are indistinguishable, written $h_1 \sim_{k_{\text{obs}},\beta} h_2$, with respect to an attacker level $k_{\text{obs}}$and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ iff:*

- *$\beta$ is a bijection between $\textbf{dom}(\beta)$ and $\textbf{rng}(\beta)$;*
- *$\textbf{dom}(\beta) \subseteq \textbf{dom}(h_1)$ and $\textbf{rng}(\beta) \subseteq \textbf{dom}(h_2)$;*
- *$\forall l \in \textbf{dom}(\beta), h_1(l) \sim_{k_{\text{obs}},\beta} h_2(\beta(l))$ where $h_1(l)$ and $h_2(\beta(l))$ are either two objects or two arrays.*

**Definition 3.3.8** (Output indistinguishability)**.** *Given an attacker level $k_{\text{obs}}$, a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, an output level $\vec{k_r}$, the indistinguishability of two final states in method $m$ is defined by the clauses below where $\rightarrow$ indicates logical implication :*

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad \vec{k_r}(\text{Norm}) \leq k_{\text{obs}} \rightarrow v_1 \sim_\beta v_2}{(v_1, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad (\textbf{class}(h_1(l_1)) : k) \in \vec{k_r} \quad k \leq k_{\text{obs}} \quad l_1 \sim_\beta l_2}{(\langle l_1 \rangle, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad (\textbf{class}(h_1(l_1)) : k) \in \vec{k_r} \quad k \not\leq k_{\text{obs}}}{(\langle l_1 \rangle, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad (\textbf{class}(h_2(l_2)) : k) \in \vec{k_r} \quad k \not\leq k_{\text{obs}}}{(v_1, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad (\textbf{class}(h_1(l_1)) : k_1) \in \vec{k_r} \quad k_1 \not\leq k_{\text{obs}} \quad (\textbf{class}(h_2(l_2)) : k_2) \in \vec{k_r} \quad k_2 \not\leq k_{\text{obs}}}{(\langle l_1 \rangle, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (\langle l_2 \rangle, h_2)}$$

At this point, it is worth mentioning that whenever it is clear from the usage, we may drop some subscript from the indistinguishability relation, e.g., for two indistinguishable objects $o_1$ and $o_2$ w.r.t. a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and observer level $k_{\text{obs}}$, instead of writing $o_1 \sim_{k_{\text{obs}},\beta} o_2$ we may drop $k_{\text{obs}}$ and write $o_1 \sim_\beta o_2$ if $k_{\text{obs}}$is obvious. We may also drop $k_h$ from a policy $\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}$ and write $\vec{k_a} \rightarrow \vec{k_r}$ if $k_h$ is irrelevant to the discussion.

**Definition 3.3.9** (Non-interferent JVM method)**.** *A method $m$ is* non-interferent *w.r.t. a policy $\vec{k_a} \rightarrow \vec{k_r}$, if for every attacker level $k_{\text{obs}}$, every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every*

$\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}, h_1, h_2, h_1', h_2' \in Heap, r_1, r_2 \in \mathcal{V} + \mathcal{L}$ *s.t.*

$$\begin{array}{ll} \langle 1, \rho_1, \epsilon, h_1\rangle \rightsquigarrow_m^+ r_1, h_1' & h_1 \sim_{k_{\text{obs}},\beta} h_2 \\ \langle 1, \rho_2, \epsilon, h_2\rangle \rightsquigarrow_m^+ r_2, h_2' & \rho_1 \sim_{k_{\text{obs}},\vec{k_a},\beta} \rho_2 \end{array}$$

*there exists a partial function* $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ *s.t.* $\beta \subseteq \beta'$ *and*

$$(r_1, h_1') \sim_{k_{\text{obs}},\beta',\vec{k_a}} (r_2, h_2').$$

Because of method invocation, there will be a notion of a side effect preorder for the notion of safety.

**Definition 3.3.10** (Side effect preorder)**.** *Two heaps* $h_1, h_2 \in$ **Heap** *are* side effect pre-ordered *(written as* $h_1 \preceq_k h_2$*) with respect to a security level* $k \in \mathcal{S}$ *if and only if* $\mathbf{dom}(h_1) \subseteq \mathbf{dom}(h_2)$ *and* $h_1(l).f = h_2(l).f$ *for all location* $l \in \mathbf{dom}(h_1)$ *and all fields* $f \in \mathcal{F}$ *such that* $k \not\leq \mathbf{ft}(f)$*.*

From which we can define a *side-effect-safe* method.

**Definition 3.3.11** (Side effect safe)**.** *A method* $m$ *is* side-effect-safe *with respect to a security level* $k_h$ *if for all local variables* $\rho \in \mathcal{X} \rightharpoonup \mathcal{V}$*, all heaps* $h, h' \in$ **Heap** *and value* $v \in \mathcal{V}, \langle 1, \rho, \epsilon, h\rangle \rightsquigarrow_m^+ v, h'$ *implies* $h \preceq_{k_h} h'$*.*

**Definition 3.3.12** (Safe JVM method)**.** *A method* $m$ *is* safe *w.r.t. a policy* $\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}$ *if* $m$ *is side-effect safe w.r.t.* $k_h$ *and* $m$ *is non-interferent w.r.t.* $\vec{k_a} \rightarrow \vec{k_r}$*.*

**Definition 3.3.13** (Safe JVM program)**.** *A program is* safe *w.r.t. a table* $\Gamma$ *of method signature if every method* $m$ *is safe w.r.t. all policies in* $\mathbf{Policies}_\Gamma(m)$*.*

**Theorem 3.3.1.** *Let* $P$ *be a JVM typable program w.r.t. safe CDRs (*$\mathbf{region_m}, \mathbf{jun_m}$*) and a table* $\Gamma$ *of method signatures. Then* $P$ *is safe w.r.t.* $\Gamma$*.*

## 3.4 Infrastructure for Android Bytecode Certification

We extend Barthe et al. [2007, 2013] to be applicable to Android bytecode. Our proposed infrastructure still retains the structure of PCC, but with the extension of certificate translation from JVM type system. In particular, we develop a type system to enforce non-interference property on Android bytecode and show that the certificate from JVM bytecode can be translated as the certificate of the resulting Android bytecode.

Figure 3.5 shows how our structure still retains this PCC infrastructure. The security policy in our scenario is that of non-interference. The code producer can translate a certificate from JVM bytecode and inject the certificate into resulting Android app as a proof that the app is non-interferent. The end users then can type check this certificate on their own phone using our type checker app. This corresponds to the PCC infrastructure in that the developer provided the proof and the

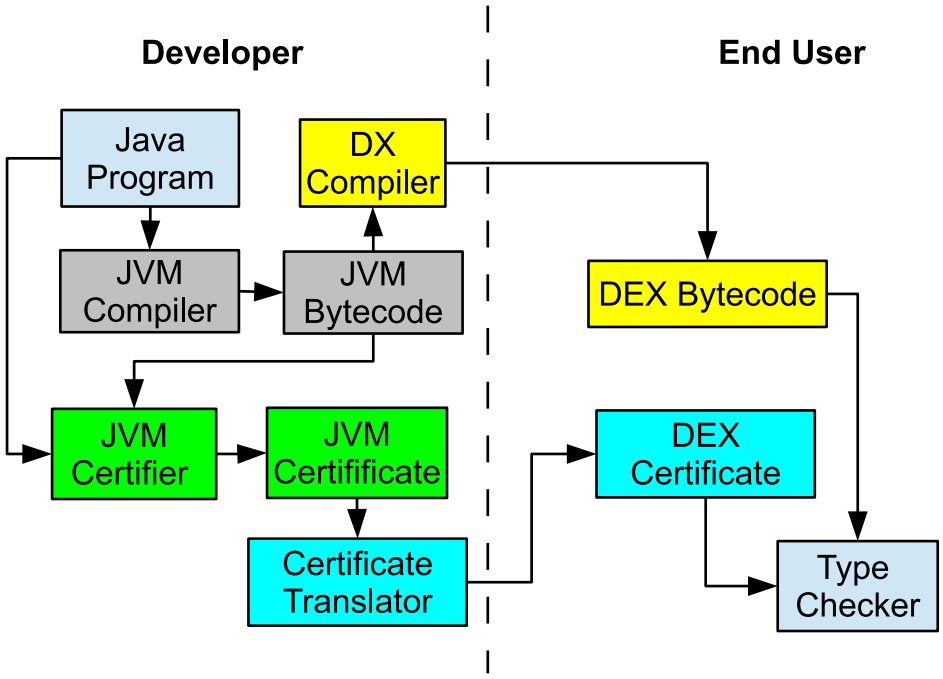



Figure 3.5: PCC structure for Android Bytecode

client just need to check whether the proof validates the bytecode's property. This means that the trusted computing base for our infrastructure is just the type checker application which can run on the user's phone. Section 6.3 gives more detail of our infrastructure.

Since our framework is essentially of PCC infrastructure, we also inherit benefits from PCC. One of the most obvious ones is that we do not have to assume that the compiler is correct. As long as there is a certificate that is produced and can be checked, it is okay even if the compiler is actually unsound in some cases. It is, of course, better if we can prove the compiler correct down to the smallest details but that is only to ensure usability, i.e., that we know a certificate for DEX bytecode can be produced from a certificate for JVM bytecode that complies with how the compiler translates the JVM bytecode to DEX bytecode. Even if the compiler occasionally produces a wrong certificate, there is no harm to security since the certificate will fail type checking.

Although the proposed type system used in our infrastructure is only applicable to certain features, the client could allow more features to be used, provided that there is a guarantee that those features are safe. That is another question about extending the type system to allow more features, which is independent of whether the existing type system is safe or not. It does mean, for example, that this framework cannot be used to certify apps that use low-level hardware acceleration for games that require native libraries. But it is perfectly useable and practical to use it to certify, e.g., password managers, contact managers, etc., that does not require complicated

language features to write.

One of the limitations of PCC is that we have to express the security policy in a very precise and concise manner. This means that due to the design of our type system, our framework is only meant to provide a guarantee against privilege misuse (apps ask for permission to access sensitive information and granted, and PCC is there to ensure it does not leak the information that it obtains). It does not prevent exploits that leverage on system vulnerabilities that violate our assumptions, in that the underlying Android system is trustworthy.

Chapter 4

# Non-Interferent Type System for Android Bytecode

The development of the operational semantics and the type systems for DEX bytecode follows closely the framework set up in Barthe et al. [2013]. Although Dalvik is a register-based machine and JVM is a stack-based machine, the translation from one instruction set to the other is the for the most part quite straightforward. The adaptation of the type system for JVM to its DEX counterpart is complicated slightly by the need to simulate JVM stacks in DEX registered-based instructions. The non-trivial parts are when we want to capture both direct (via operand stacks) and indirect information flow (e.g., those resulting from branching on high value). In Barthe et al. [2013], to deal with both direct and indirect flow, several techniques are used, among others, the introduction of operand stack types (each stack element carries a type which is a security label), a notion of safe control dependence region (CDR), which keeps track of the regions of the bytecode executing under a 'high' security level, and the notion of security environment, which attaches security levels to points in programs. Since Dalvik is a register-based machine, when translating a JVM bytecode to DEX bytecode, the dx tool simulates the operand stack using DEX registers. As the type system for JVM is parameterized by a safe CDR and a security environment, we also need to define how these are affected by the translation, e.g., whether one can construct a safe CDR for DEX given a safe CDR for JVM. This was complicated by the fact that the translation by dx, in general, is organized along blocks of sequential (non-branching) code, so one needs to relate blocks of code in the image of the translation back to the original code (see Section 6.1) .

In a sense, DEX is simpler than JVM due to the difference where JVM uses local variables and operand stack, and DEX only uses registers. For starters, the representation of a state is simpler in DEX as we collapse local variables and operand stack into registers. We also do not need the lift mechanism for DEX. In JVM, the presence of instruction **swap** means that there is a possibility of implicit information flow through stack operations hence the need for a lift mechanism. Unlike JVM, DEX does not have an instruction where the content of registers can be swapped around, so DEX does not need this mechanism. Unfortunately, since we do not have a lift mechanism anymore, the way we prove the type system soundness is also different

(see Section 4.2).

## 4.1 Syntax, Semantics, and Type System for Android Bytecode

We first propose the design of our non-interferent type system for Android bytecode. In Section 4.2 we prove that our type system is sound.

### 4.1.1 Overview of DEX Bytecode

A program $P$ is given by its list of instructions in Figure 4.1. The set $\mathcal{R}$ is the set of DEX virtual registers, $\mathcal{V}$ is the set of values, $\mathcal{PP}$ is the set of program points, and $\rho$ is the mapping from registers to values.

As in the case for JVM, we assume that the program comes equipped with the set of class names $\mathcal{C}$ and the set of fields $\mathcal{F}$. The program will also be extended with array manipulation instructions and the program will come parameterized by the set of available DEX types $\mathcal{T}_D$ analogous to Java type $\mathcal{T}_J$. The DEX language also deals with method invocation. As for JVM, DEX programs will also come with a set $m$ of method names. The method name and signatures themselves are represented explicitly in the DEX file, as such the lookup function required will be different from the JVM counterpart in that we do not need the class argument. Thus in the sequel, we will remove this lookup function and overload that method ID to refer to the code as well. DEX uses two special registers. We will use *ret* for the first one which can hold the return value of a method invocation. In the case of a **moveresult**, the instruction behaves like a **move** instruction with the special register *ret* as the source register. The second special register is *ex* which stores an exception thrown for the next instruction. Figure 4.1 contains the list of DEX instructions.

### 4.1.2 Operational Semantics

A state in DEX is just $\langle i, \rho, h \rangle$ where the $\rho$ here is a mapping from registers to values and $h$ is the heap. It is similar to that of the JVM, with several differences, e.g., the state in DEX does not have an operand stack, but its functionality is covered by the registers (local variables) $\rho$. The function $\mathbf{fresh} : \text{Heap} \to \mathcal{L}$ is an allocator function that given a heap returns the location for that object. The function $\mathbf{default} : \mathcal{C} \to \mathcal{O}$ returns for each class a default object of that class. For every field of that default object, the value will be 0 if the field is of numeric type, and *null* if the field is of object type. Similarly for $\mathbf{defaultArray} : \mathbb{N} \times \mathcal{T}_D \to (\mathbb{N} \rightharpoonup \mathcal{V})$. The $\rightsquigarrow$ relation which defines transitions between state is $\rightsquigarrow \subseteq \mathbf{State} \times (\mathbf{State} + (\mathcal{V} \times \mathbf{Heap}))$.

The notation $\underline{\text{op}}$ denotes here the standard interpretation of arithmetic operation of $op$ in the domain $\mathcal{V}$ of values (although there is no arithmetic operation on locations). The operator $\oplus$ denotes the function where $\rho \oplus \{r \mapsto v\}$ means a new function $\rho'$ such that $\forall i \in \mathbf{dom}(\rho) \backslash \{r\}. \rho'(i) = \rho(i)$ and $\rho'(r) = v$. The operator $\oplus$ is overloaded to also mean the update of a field on an object or update on a heap.

| | | |
|---|---|---|
| **binop** $op$ | $r, r_a, r_b$ | Do the binary operation $op$ on the values contained in $r_a$ and $r_b$, then store the result in $r$. |
| **const** | $r, v$ | Fill register $r$ with the constant value $v$. |
| **move** | $r, r_s$ | Copy the value stored in $r_s$ to $r$. |
| **ifeq** | $r, t$ | Conditional jump if $r$ has the value of 0. |
| **ifneq** | $r, t$ | Conditional jump if $r$ has the value of anything but 0. |
| **goto** | $t$ | Unconditional jump to the program point $t$. |
| **return** | $r_s$ | Return the value stored in $r_s$. |
| **new** | $r, c$ | Create a new object of class $c$ and put the reference in $r$ |
| **iget** | $r, r_o, f$ | Get the value contained in the field $f$ of object referenced by $\rho(r_o)$ and store it in register $r$. |
| **iput** | $r_s, r_o, f$ | Read the value contained in register $r_s$ and store it in the field $f$ of object referenced by $\rho(r_o)$. |
| **newarray** | $r, r_l, t$ | Create a new array of type $t$ which has $r_l$ number of elements and put the reference to the array in $r$. |
| **arraylength** | $r, r_a$ | Store the length of array referenced by $r_a$ in register $r$. |
| **aget** | $r, r_a, r_i$ | Get the value stored in array referenced by $\rho(r_a)$ at index $\rho(r_i)$ and put it in register $r$. |
| **aput** | $r_s, r_a, r_i$ | Put the value stored in register $r_s$ to array referenced by $\rho(r_a)$ at index $\rho(r_i)$. |
| **invoke** | $n, m, \vec{p}$ | Invoke $\rho(\vec{p}[0]).m$ with $n$ arguments stored in $\vec{p}$ |
| **moveresult** | $r$ | Store invoke's result to $r$. This instruction has to be placed directly after invoke, otherwise the result is lost. |
| **throw** | $r$ | Throw the exception object stored in $r$ |
| **moveexception** | $r$ | Store exception in $r$. This instruction has to be the first instruction in the handler region. |

where $op \in \{+, -, \times, /\}, v \in \mathbb{Z}, \{r, r_a, r_b, r_s\} \in \mathcal{R}, t \in \mathcal{PP}, c \in \mathcal{C}, f \in \mathcal{F}$ and $\rho : \mathcal{R} \to \mathbb{Z}$.

Figure 4.1: DEX Instruction List

$$\frac{P_m[i] = \mathbf{const}(r, v)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto v\}, h\rangle} \qquad \frac{P_m[i] = \mathbf{move}(r, r_s) \quad r_s \in \mathbf{dom}(\rho)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto \rho(r_s)\}, h\rangle}$$

$$\frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b) \quad r_a, r_b \in \mathbf{dom}(\rho) \quad n = \rho(r_a) \ \underline{op} \ \rho(r_b)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto n\}, h\rangle}$$

$$\frac{P_m[i] = \mathbf{goto}(t)}{\langle i, \rho, h\rangle \rightsquigarrow \langle t, \rho, h\rangle} \qquad \frac{P[i]_m = \mathbf{ifeq}(r, t) \quad \rho(r) = 0}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle t, \rho, h\rangle} \qquad \frac{P_m[i] = \mathbf{ifeq}(r, t) \quad \rho(r) \neq 0}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, h\rangle}$$

$$\frac{P[i]_m = \mathbf{return}(r_s) \quad r_s \in \mathbf{dom}(\rho)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \rho(r_s), h} \qquad \frac{P_m[i] = \mathbf{new}(r, c) \quad l = \mathbf{fresh}(h)}{\langle i, \rho, h\rangle \rightsquigarrow \langle i+1, \rho \oplus \{r \mapsto l\}, h \oplus \{l \mapsto \mathbf{default}(c)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{iget}(r, r_o, f) \quad \rho(r_o) \in \mathbf{dom}(h) \quad f \in \mathbf{dom}(h(\rho(r_o)))}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto h(\rho(r_o)).f\}, h\rangle}$$

$$\frac{P_m[i] = \mathbf{iget}(r, r_o, f) \quad \rho(r_o) = null \quad l' = \mathbf{fresh}(h)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{np}} \mathbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{P_m[i] = \mathbf{iput}(r_s, r_o, f) \quad \rho(r_o) = null \quad l' = \mathbf{fresh}(h)}{\langle i, \rho, h\rangle \rightsquigarrow_{n,\text{np}} \mathbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{P_m[i] = \mathbf{iput}(r_s, r_o, f) \quad \rho(r_o) \in \mathbf{dom}(h) \quad f \in \mathbf{dom}(h(\rho(r_o)))}{\langle i, \rho, h\rangle \rightsquigarrow_{n,\text{Norm}} \langle i+1, \rho, os, h \oplus \{\rho(r_o) \mapsto h(\rho(r_o)) \oplus \{f \mapsto \rho(r_s)\}\}\rangle}$$

$$\frac{P_m[i] = \mathbf{newarray}(r, r_l, t) \quad l = \mathbf{fresh}(h) \quad \rho(r_l) \geq 0}{\langle i, \rho, h\rangle \rightsquigarrow \langle i+1, \rho \oplus \{r \mapsto l\}, h \oplus \{l \mapsto (\rho(r_l), \mathbf{defaultArray}(\rho(r_l), t), i)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad \rho(r_a) \in \mathbf{dom}(h)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto h(\rho(r_a)).\mathbf{length}, h\}\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad \rho(r_a) = null \quad l' = \mathbf{fresh}(h)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{np}} \mathbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad \rho(r_a) \in \mathbf{dom}(h) \quad 0 \leq \rho(r_i) < h(\rho(r_a)).\mathbf{length}}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto h(\rho(r_a))[\rho(r_i)]\}, h\rangle}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad \rho(r_a) = null \quad l' = \mathbf{fresh}(h)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{np}} \mathbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

$$\frac{P_m[i] = \mathbf{aput}(r_s, r_a, r_i) \quad \rho(r_a) \in \mathbf{dom}(h) \quad 0 \leq \rho(r_i) < h(\rho(r_a)).\mathbf{length}}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, h \oplus \{\rho(r_a) \mapsto h(\rho(r_a)) \oplus \{\rho(r_i) \mapsto \rho(r_s)\}\}\rangle}$$

$$\frac{P_m[i] = \textbf{aput}(r_s, r_a, r_i) \quad \rho(r_a) = null \quad l' = \textbf{fresh}(h)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\mathrm{np}} \textbf{RuntimeExcHandling}(h, l', \mathrm{np}, i, \rho)}$$

$$\frac{P_m[i] = \textbf{moveresult}(r) \quad r \in \textbf{dom}(\rho)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\mathrm{Norm}} \langle i+1, \rho \oplus \{r \mapsto \rho(ret)\}, h\rangle}$$

$$\frac{P_m[i] = \textbf{invoke}(n, m', \vec{p}) \quad \vec{p} \in \textbf{dom}(\rho) \quad \langle 1, \{\vec{x} \mapsto \vec{p}\}, h\rangle \rightsquigarrow^{+}_{m'} v, h'}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\mathrm{Norm}} \langle i+1, \rho \oplus \{ret \mapsto v\}, h'\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \textbf{invoke}(n, m', \vec{p}) \quad \vec{p} \in \textbf{dom}(\rho) \quad \langle 1, \{\vec{x} \mapsto \vec{p}\}, h\rangle \rightsquigarrow^{+}_{m'} \langle l'\rangle, h' \\ e = \textbf{class}(h'(l')) \quad \textbf{Handler}_{\mathbf{m}}(i, e) = t \quad e \in \textbf{excAnalysis}(m')\end{array}}{\langle i, \rho, h\rangle \rightsquigarrow_{m,e} \langle t, \rho \oplus \{ex \mapsto l'\}, h'\rangle}$$

$$\frac{P_m[i] = \textbf{invoke}(n, m', \vec{p}) \quad l' = \textbf{fresh}(h) \quad \rho(\vec{p}[0]) = null}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\mathrm{np}} \textbf{RuntimeExcHandling}(h, l', \mathrm{np}, i, \rho)}$$

$$\frac{\begin{array}{c}P_m[i] = \textbf{invoke}(n, m', \vec{p}) \quad \vec{p} \in \textbf{dom}(\rho) \quad \langle 1, \{\vec{x} \mapsto \vec{p}\}, h\rangle \rightsquigarrow^{+}_{m'} \langle l'\rangle, h' \\ e = \textbf{class}(h'(l')) \quad \textbf{Handler}_{\mathbf{m}}(i, e) \uparrow \quad e \in \textbf{excAnalysis}(m')\end{array}}{\langle i, \rho, h\rangle \rightsquigarrow_{m,e} \langle l'\rangle, h'}$$

$$\frac{\begin{array}{c}P_m[i] = \textbf{throw}(r) \quad \rho(r) \in \textbf{dom}(h) \quad e = \textbf{class}(h(\rho(r))) \\ \textbf{Handler}_{\mathbf{m}}(i, e) = t \quad e \in \textbf{classAnalysis}(m, i)\end{array}}{\langle i, \rho, h\rangle \rightsquigarrow_{m,e} \langle t, \rho \oplus \{ex \mapsto \rho(r)\}, h\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \textbf{throw}(r) \quad \rho(r) \in \textbf{dom}(h) \quad e = \textbf{class}(h(\rho(r))) \\ \textbf{Handler}_{\mathbf{m}}(i, e) \uparrow \quad e \in \textbf{classAnalysis}(m, i)\end{array}}{\langle i, \rho, h\rangle \rightsquigarrow_{m,e} \langle \rho(r)\rangle, h}$$

$$\frac{P_m[i] = \textbf{throw}(r) \quad l' = \textbf{fresh}(h) \quad \rho(r) = null}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\mathrm{np}} \textbf{RuntimeExcHandling}(h, l', \mathrm{np}, i, \rho)}$$

$$\frac{P_m[i] = \textbf{moveexception}(r) \quad r \in \textbf{dom}(\rho)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\mathrm{Norm}} \langle i+1, \rho \oplus \{r \mapsto \rho(ex)\}, h\rangle}$$

$\textbf{RuntimeExcHandling} : \textbf{Heap} \times \mathcal{L} \times \mathcal{C} \times \mathcal{PP} \times (\mathcal{R} \rightharpoonup \mathcal{V}) \to \textbf{State} + (\mathcal{L} \times \textbf{Heap})$
defined as

$$\textbf{RuntimeExcHandling}(h, l', C, i, \rho) = \begin{cases} \langle t, \rho \oplus \{ex \mapsto l'\}, h \oplus \{l' \mapsto \textbf{default}(C)\}\rangle & \text{if } \textbf{Handler}_{\mathbf{m}}(i, C) = t \\ \langle l'\rangle, h \oplus \{l' \mapsto \textbf{default}(C)\} & \text{if } \textbf{Handler}_{\mathbf{m}}(i, C) \uparrow \end{cases}$$

Figure 4.2: DEX Operational Semantic

There is also an additional partial function $\mathbf{Handler_m} : \mathcal{PP} \times \mathcal{C} \rightharpoonup \mathcal{PP}$ for method $m$ which gives the handler address for a given program point and type of exception. Given a program point $i$ and a thrown exception $c$, if $\mathbf{Handler_m}(i, c) = t$ then the control will be transferred to program point $t$, if the handler is undefined (denoted by $\mathbf{Handler_m}(i, c) \uparrow$) then the exception is uncaught in method $m$.

Figure 4.2 shows the operational semantics for DEX instructions. Here we give the intuitions for each of the instruction:

**const**$(r, v)$**:** updates the mapping for register $r$ with the value $v$. This instruction is the equivalent of JVM instruction **push** $v$. Whereas **push** always put the constant value $v$ at the top of the stack; the **const** specifies where the value will be stored.

**move**$(r, r_s)$**:** takes the value contained in $r_s$ and then update the mapping for register $r$ with this value. This instruction simulates the instruction **store** $x$ and **load** $x$ in the JVM. The only difference between them is that the JVM relates the top of the stack and a particular local variable $x$ (**store** puts the value at the top of the stack to the local variable $x$, and **load** puts the value of the local variable $x$ to the top of the stack) while there is no restrictions for **move**.

**binop**$(op, r, r_a, r_b)$**:** takes the value contained in $r_a$ and $r_b$, and apply the binary operation to these values. The mapping for register $r$ is then updated with the resulting value. This instruction is the equivalent of JVM instruction **binop** $op$, but with a difference in that the operands and the place to put the result in JVM are fixed (always pop the top two values from the stack, and then place the result on top of the stack) but in DEX we have to specify the registers ($r_a$ and $r_b$ for the operands and $r$ as the target register).

**goto**$(t)$**:** transfer the next program point to execute to $j$. This instruction behaves exactly like its JVM counterpart.

**ifeq**$(r, t)$**:** transfer the next program point to execute to $t$ if the value contained in register $r$ is 0. Otherwise, the next program point to execute is the next program point. This instruction behaves really similar to its JVM counterpart. The only difference is that in JVM, the conditional jump depends on the top of the stack whereas in DEX it depends on the register specified.

**return**$(r_s)$**:** ends the execution with the value contained in register $r_s$. Just like **ifeq**, it is really similar to JVM's **return** except that JVM always return a value from the top of the stack and DEX can specify from which register it will return a value.

**new**$(r, c)$**:** updates the mapping for register $r$ with the created new object of class $c$ initialized with the default value. JVM also has **new** with the difference that in JVM, the newly instantiated object reference will be put on top of the stack while in DEX we have to specify where we will store the reference.

**iget**$(r, r_o, f)$**:** get the value contained in the field $f$ of the object whose reference is contained in $r_o$, and then update the mapping for register $r$ with this value. If there is no exception, the instruction succeeds, and the program continues with the next program point. In the case where there is an exception, the behavior of the program is modeled by the function **RuntimeExcHandling**, which will creates an exception object and then either transfer the program point to the handler of such exception if it exists, or terminates the exception with the exception object. Such behavior is exhibited by all of the exception throwing instructions, so we will not mention this behavior for the rest of the exception throwing instructions. It is the DEX version of **getfield** $f$, with more flexibility to specify where we the reference of the object is ($r_o$) and where we will put the value ($r$). In JVM, it will pop the object reference that is at the top of the stack and then put back the value to the top of the stack.

**iput**$(r_s, r_o, f)$**:** get the value contained in register $r_s$ and then copy the value to the field $f$ of the object whose reference is contained in $r_o$. **iput** is an exception throwing instruction, so the behavior mentioned in **iget** is also applicable here. In JVM, **putfield** $f$ will pop top two values from the stack for the source value and the object reference, whereas in DEX we have to specify the register containing the source value ($r_s$) and the register containing the object reference ($r_o$).

**newarray**$(r, r_l, t)$**:** create a new array of type $t$ with the length of $\rho(r_l)$ initialized with the default value, and then update the mapping for register $r$ with the reference to this newly created array. It is really similar to **newarray** $t$ in that it is creating a new array of type $t$ with a certain length. The difference lies in where do we get the length of the array from, and where will we put the reference for the newly created array. In JVM, the length of the array is the value at the top of the stack, and the reference will be put at the top of the stack, while in DEX the length of the array is contained in register $r_i$ and the reference will be put in $r$.

**arraylength**$(r, r_a)$**:** update the mapping for register $r$ with the length of the array whose reference is contained in register $r_a$. **arraylength** is an exception throwing instruction, so the behavior mentioned in **iget** is also applicable here. As usual, it behaves similarly to its JVM counterpart except that in JVM, the array reference is always located at the top of the stack, and the length of the array will be put on top of the stack. DEX **arraylength** will take the array reference that is stored in $r_a$ and then put the length in register $r$.

**aget**$(r, r_a, r_i)$**:** get the value of the array whose reference is contained in $r_a$ at index $\rho(r_i)$ and then update the mapping for register $r$ with this value. **aget** is an exception throwing instruction, so the behavior mentioned in **iget** is also applicable here. **arrayload** in JVM uses the top two values from the stack for the array reference and its index, and then put the content of the array on top of

the stack. It is different compared to DEX where the array location, array index, and the register to store the value are flexible.

**aput**$(r_s, r_a, r_i)$**:** get the value contained in register $r_s$ and then update the value contained in the array (whose reference is contained in register $r_a$) at index $\rho(r_i)$ with this value. **aput** is an exception throwing instruction, so the behavior mentioned in **iget** is also applicable here. **arraystore** in JVM uses the three two values from the stack for source value, the array reference, and its index. It is different compared to DEX where the array location, array index, and the register which contain the source value are flexible.

**moveresult**$(r)$**:** get the value contained in the pseudo register *ret* and update the mapping for register $r$ with this value. There is no JVM equivalent of this instruction.

**invoke**$(n, m, \vec{p})$**:** execute the method $m$ with the parameters contained in registers $\vec{p}$. The resulting value of executing the method will be contained in the pseudo register *ret*. **invoke** is an exception throwing instruction, so the behavior mentioned in **iget** is also applicable here. There are several differences with JVM's **invoke**. Firstly, in JVM the number of parameters $n$ is implicit in the method definition. Secondly, the parameters for the method invoked in JVM are located on top of the stack, whereas in DEX we can specify which registers are the parameters for the method.

**throw**$(r)$**:** throw the exception contained in the register $r$. If the handler for such exception exists, the program point is then transferred to the handler. Otherwise, the program will ends abruptly with the exception as the result. The difference with its JVM counterpart is the location of the exception. In JVM, the exception is always obtained from the top of the stack, whereas in DEX it is obtained from the register $r$.

**moveexception**$(r)$**:** move the value contained in the pseudo register *ex* and then update the mapping for register $r$ with this value. This instruction only ever appears as the first instruction in an exception handler. There is no equivalent of this instruction in JVM.

**Successor Relation** The successor relation closely resembles that of the JVM; instructions will have its next instruction as the successor, except jump instructions, return instructions, and instructions that throw an exception. As with JVM, whenever it is clear from the context, we will drop the tag to reduce clutter.

- $P_m[i]$ = **goto** $t$. The successor relation is $i \mapsto^{\text{Norm}} t$
- $P_m[i]$ = **ifeq** $t$ or $P_m[i]$ = **ifneq** $t$. In this case, there are 2 successor relations denoted by $i \mapsto^{\text{Norm}} i+1$ and $i \mapsto^{\text{Norm}} t$.
- $P_m[i]$ = **return**. In this case, it is a return point denoted by $i \mapsto^{\text{Norm}}$

- $P_m[i]$ is an instruction throwing a null pointer exception, and there is a handler for it ($\mathbf{Handler}(i, \mathrm{np}) = t$). In this case, the successor is $t$ denoted by $i \mapsto^{\mathrm{np}} t$.
- $P_m[i]$ is an instruction throwing a null pointer exception, and there is no handler for it ($\mathbf{Handler}(i, \mathrm{np}) \uparrow$). In this case it is a return point denoted by $i \mapsto^{\mathrm{np}}$.
- $P_m[i] = \mathbf{throw}$, throwing an exception $C \in \mathbf{classAnalysis}(m, i)$, and the handler is $\mathbf{Handler}(i, C) = t$. The successor relation is $i \mapsto^{C} t$.
- $P_m[i] = \mathbf{throw}$, throwing an exception $C \in \mathbf{classAnalysis}(m, i)$, and the handler is $\mathbf{Handler}(i, C) = \uparrow$. It is a return point, and the successor relation is $i \mapsto^{C}$.
- $P_m[i] = \mathbf{invoke}\ m_{\mathrm{ID}}$, throwing an exception $C \in \mathbf{excAnalysis}(m_{\mathrm{ID}})$, and the handler is $\mathbf{Handler}(i, C) = t$. The successor relation is $i \mapsto^{C} t$.
- $P_m[i] = \mathbf{invoke}\ m_{\mathrm{ID}}$, throwing an exception $C \in \mathbf{excAnalysis}(m_{\mathrm{ID}})$, and the handler is $\mathbf{Handler}(i, C) \uparrow$. It is a return point, and the successor relation is $i \mapsto^{C}$.
- $P_m[i]$ is any other cases. The successor is its immediate instruction denoted by $i \mapsto^{norm} i + 1$

### 4.1.3 Type System

The transfer rules of DEX are defined in terms of register typing $rt : (\mathcal{R} \to \mathcal{S})$ instead of stack typing. Note that this registers typing is total and is restricted to the number of registers used in a method, which is statically determined, so we know in advance how many registers there are. To be more concrete, if a method only uses 16 registers, then $rt$ is a map for these 16 registers to security levels, as opposed to the whole number of possible 65535 registers according to And [2017].

The typing system is formally parameterized by :

**Γ:** a table of method signatures, needed to define the transfer rules for method invocation;

**ft:** a map from fields to their global policy level;

**CDR:** a structure consisting of (**region**, **jun**).

*se*: a security environment

*sgn*: the method signature of the current method

*rt*: register typing after the instruction is executed

Thus the complete form of a judgment parameterized by a tag $\tau \in \{\mathrm{Norm} + \mathcal{C}\}$ is

$$\Gamma, \mathbf{ft}, \mathbf{region}, se, sgn, i \vdash^{\tau} RT_i \Rightarrow rt$$

although in the case where some elements are unnecessary, we may not write the full notation whenever it is clear from the context. In the table of operational semantics, we may drop the subscript $m$,Norm from $\rightsquigarrow$, e.g., we may write $\rightsquigarrow$ instead of $\rightsquigarrow_{m,\text{Norm}}$ to mean the same thing. In the table of transfer rules, we may drop the superscript tag from $\vdash^{\tau}$ and write $\vdash$ instead. The same case applies to the typing judgment, we may write $i \vdash^{\tau} rt \Rightarrow rt'$ instead of $\Gamma, \mathbf{ft}, \mathbf{region}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash^{\tau} rt \Rightarrow rt'$.

The transfer rules also come equipped with a security policy for fields $\mathbf{ft} : \mathcal{F} \rightarrow \mathcal{S}^{\text{ext}}$ and security policy for array at the point of declaration $\mathbf{at} : \mathcal{PP} \rightarrow \mathcal{S}^{\text{ext}}$. As in the type system for JVM, we overload the function $\mathbf{at}(a)$ to denote the security level of the content of array $a$ at its creation point. Some of the transfer rules for DEX instructions are contained in Figure 4.3.

The typability of the DEX closely follows that of the JVM, except that the relation between program points is defined in terms of register typing as opposed to stack typing ($i \vdash RT_i \Rightarrow rt, rt \sqsubseteq RT_j$). The definition of $\sqsubseteq$ is also defined in terms of point-wise registers. For now, we assume the existence of a safe CDR with the same definition as that of the JVM side. We shall see later how we can construct a safe CDR for DEX from a safe CDR in JVM.

**Definition 4.1.1** (Typable method)**.** *A method $m$ is typable w.r.t. a method signature table $\Gamma$, a global field policy $\mathbf{ft}$, a policy $sgn$, and a CDR $\mathbf{region_m} : \mathcal{PP} \rightarrow \wp(\mathcal{PP})$ if there exists a security environment $se : \mathcal{PP} \rightarrow \mathcal{S}$ and a function $\mathbf{RT} : \mathcal{PP} \rightarrow (\mathcal{R} \rightarrow \mathcal{S})$ s.t. $RT_1 = \vec{k_a}$ and for all $i, j \in \mathcal{PP}, e \in \{\text{Norm} + \mathcal{C}\}$:*

- *$i \mapsto^e j$ implies there exists $rt \in (\mathcal{R} \rightarrow \mathcal{S})$ such that $\Gamma$, $\mathbf{ft}$, $\mathbf{region}$, $se$, $sgn$, $i \vdash^e RT_i \Rightarrow rt$ and $rt \sqsubseteq RT_j$;*
- *$i \mapsto^e$ implies $\Gamma, \mathbf{ft}, \mathbf{region}, se, sgn, i \vdash^e RT_i \Rightarrow$*

Following that of the JVM side, what we want to establish here is not just the typability, but also that typability means non-interference. As in the JVM, the notion of non-interference relies on the definition of indistinguishability, while the notion of value indistinguishability is the same as that of JVM.

**Definition 4.1.2** (Register indistinguishability)**.** *For $\rho, \rho' : (\mathcal{R} \rightarrow \mathcal{V})$, $rt, rt' : (\mathcal{R} \rightarrow \mathcal{S})$, and a register $r \in \mathcal{R}$, two registers mapping are indistinguishable w.r.t. to a single register (denoted by $\rho \sim_{k_{\text{obs}}, rt, rt', r, \beta} \rho'$) if either:*

- *$rt(r) = rt'(r) = k$ and $k \not\leq k_{\text{obs}}$ where $k, k' \in \mathcal{S}^{\text{ext}}$; or*
- *$\rho(r) \sim_{\beta} \rho'(r)$.*

**Definition 4.1.3** (Registers indistinguishability)**.** *For $\rho, \rho' : (\mathcal{R} \rightarrow \mathcal{V})$ and $rt, rt' : (\mathcal{R} \rightarrow \mathcal{S})$, we have $\rho \sim_{k_{\text{obs}}, rt, rt', \beta} \rho'$ iff $\forall r \in \mathcal{R}, \rho \sim_{k_{\text{obs}}, rt, rt', r} \rho'$*

**Definition 4.1.4** (Object indistinguishability)**.** *Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (denoted by $o_1 \sim_{k_{\text{obs}}, \beta} o_2$) if and only if $o_1$ and $o_2$ are objects of the same class and $o_1.f \sim_{\beta} o_2.f$ for all fields $f \in \mathbf{dom}(o_1)$ s.t. $\mathbf{ft}(f) \leq k_{\text{obs}}$.*

$$\frac{P_m[i] = \mathbf{const}(r, v)}{se, i \vdash rt \Rightarrow rt \oplus \{r \mapsto se(i)\}} \qquad \frac{P_m[i] = \mathbf{move}(r, r_s)}{se, i \vdash rt \Rightarrow rt \oplus \{r \mapsto \big(rt(r_s) \sqcup se(i)\big)\}}$$

$$\frac{P_m[i] = \mathbf{ifeq}(r, t) \quad \forall j' \in \mathbf{region}(i, \mathrm{Norm}), se(i) \sqcup rt(r) \leq se(j')}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash rt \Rightarrow rt}$$

$$\frac{P_m[i] = \mathbf{ifneq}(r, t) \quad \forall j' \in \mathbf{region}(i, \mathrm{Norm}), se(i) \sqcup rt(r) \leq se(j')}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash rt \Rightarrow rt}$$

$$\frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b)}{se, i \vdash rt \Rightarrow rt \oplus \{r \mapsto \big(rt(r_a) \sqcup rt(r_b) \sqcup se(i)\big)\}}$$

$$\frac{P_m[i] = \mathbf{return}(r_s) \quad se(i) \sqcup rt(r_s) \leq \vec{k_r}(\mathrm{Norm})}{\vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash rt \Rightarrow} \qquad \frac{P_m[i] = \mathbf{new}(r, c)}{se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(i)\}}$$

$$\frac{P_m[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(r_o) \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto \big((rt(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f)\big)\}}$$

$$\frac{P_m[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} rt \Rightarrow rt \oplus \{ex \mapsto (rt(r_o) \sqcup se(i))\}}$$

$$\frac{\begin{array}{c} P_m[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \leq se(j) \\ \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad se(i) \sqcup rt(r_o) \leq \vec{k_r}(\mathrm{np}) \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} rt \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \mathbf{iput}(r, r_o, f) \quad rt(r) \in \mathcal{S}^{\mathrm{ext}} \quad rt(r_o) \in \mathcal{S} \quad (rt(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} rt(r_s) \leq \mathbf{ft}(f) \\ k_h \leq \mathbf{ft}(f) \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(r_o) \leq se(j) \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt}$$

$$\frac{\begin{array}{c} P_m[i] = \mathbf{iput}(r_s, r_o, f) \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad rt(r_o) \in \mathcal{S} \quad (rt(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} rt(r_s) \leq \mathbf{ft}(f) \\ \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} rt \Rightarrow rt \oplus \{ex \mapsto rt(r_o) \sqcup se(i)\}}$$

$$\frac{\begin{array}{c} P_m[i] = \mathbf{iput}(r_s, r_o, f) \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad rt(r_o) \in \mathcal{S} \quad (rt(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} rt(r_s) \leq \mathbf{ft}(f) \\ \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad se(i) \sqcup rt(r_o) \leq \vec{k_r}(\mathrm{np}) \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} rt \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{newarray}(r, r_l, t) \quad rt(r_l) \in \mathcal{S} \quad rt(r_l)[\mathbf{at}(i)] \leq \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto rt(r_l)[\mathbf{at}(i)]\}}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto k\}}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad k \leq \vec{k_a}(r) \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow rt \oplus \{ex \mapsto (k \sqcup se(i))\}}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad k \leq \vec{k_a}(r) \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) \uparrow \quad se(i) \sqcup k \leq \vec{k_a}[\text{np}]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto \big((se(i) \sqcup k \sqcup rt(r_i)\big) \sqcup^{\text{ext}} k_c)\}}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow rt \oplus \{ex \mapsto (k \sqcup se(i))\}}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) \uparrow \quad se(i) \sqcup k \leq \vec{k_r}(\text{np})}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt}$$

$$\frac{P_m[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow rt \oplus \{ex \mapsto (k \sqcup se(i))\}}$$

$$\frac{P_m[i] = \textbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad se(i) \sqcup k \leq \vec{k_r}(\text{np}) \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad \forall j \in \textbf{region}(i, \text{np}), k \leq se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow}$$

$$\frac{P_m[i] = \textbf{moveresult}(r)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(i) \sqcup rt(ret)\}}$$

$$\frac{P_m[i] = \textbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\to} \vec{k'_r} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall 0 \leq i < n . rt(\vec{p}[i]) \leq \vec{k'_a}[i] \quad k_e = \bigsqcup \{\vec{k'_r}(e) \mid e \in \textbf{excAnalysis}(m')\} \quad \forall j \in \textbf{region}(i, \text{Norm}), rt(\vec{p}[0]) \sqcup k_e \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow \left(rt \oplus \{ret \mapsto \vec{k'_r}(\text{Norm}) \sqcup se(i)\}\right)}$$

$$\frac{P_m[i] = \textbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\to} \vec{k'_r} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall 0 \leq i < n . rt(\vec{p}[i]) \leq \vec{k'_a}[i] \quad \textbf{Handler}(i, e) = t \quad e \in \textbf{excAnalysis}(m') \cup \{\text{np}\} \quad \forall j \in \textbf{region}(i, e), rt(\vec{p}[0]) \sqcup k'_r[e] \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{e} rt \Rightarrow rt \oplus \{ex \mapsto (rt(\vec{p}[0]) \sqcup \vec{k'_r}(e))\}}$$

$$\frac{P_m[i] = \textbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\to} \vec{k'_r} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall 0 \leq i < n . rt(\vec{p}[i]) \leq \vec{k'_a}[i] \quad rt(\vec{p}[0]) \sqcup se(i) \sqcup \vec{k'_r}(e) \leq \vec{k_r}(e) \quad \textbf{Handler}(i, e) \uparrow \quad e \in \textbf{excAnalysis}(m') \cup \{\text{np}\} \quad \forall j \in \textbf{region}(i, e), rt(\vec{p}[0]) \sqcup k'_r[e] \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{e} rt \Rightarrow}$$

$$\frac{P_m[i] = \textbf{throw}(r) \quad e \in \textbf{classAnalysis}(i) \cup \{\text{np}\} \quad \textbf{Handler}(i, e) = t \quad \forall j \in \textbf{region}(i, e), rt(r) \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{e} rt \Rightarrow rt \oplus \{ex \mapsto (rt(r) \sqcup se(i))\}}$$

$$\frac{P_m[i] = \textbf{throw}(r) \quad e \in \textbf{classAnalysis}(i) \cup \{\text{np}\} \quad se(i) \sqcup rt(r) \leq \vec{k_r}(e) \quad \textbf{Handler}(i, e) \uparrow \quad \forall j \in \textbf{region}(i, e), rt(r) \leq se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{e} rt \Rightarrow}$$

$$\frac{P_m[i] = \textbf{moveexception}(r)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto (rt(ex) \sqcup se(i))\}}$$

Figure 4.3: DEX Transfer Rule

**Definition 4.1.5** (Array indistinguishability)**.** *Two arrays $a_1, a_2 \in \mathcal{A}$ are indistinguishable w.r.t. an attacker level $k_{\text{obs}}$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (denoted by $a_1 \sim_{k_{\text{obs}},\beta} o_2$) if and only if $a_1$.**length** = $a_2$.**length** and, moreover, if $\mathbf{at}(a_1) \leq k_{\text{obs}}$, then $a_1[i] \sim_\beta a_2[i]$ for all $i$ such that $0 \leq i < a_1$.**length**.*

**Definition 4.1.6** (Heap indistinguishability)**.** *Two heaps $h_1$ and $h_2$ are indistinguishable with respect to an attacker level $k_{\text{obs}}$and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, written $h_1 \sim_{k_{\text{obs}},\beta} h_2$, if and only if :*

- *$\beta$ is a bijection between $\mathbf{dom}(\beta)$ and $\mathbf{rng}(\beta)$;*
- *$\mathbf{dom}(\beta) \subseteq \mathbf{dom}(h_1)$ and $\mathbf{rng}(\beta) \subseteq \mathbf{dom}(h_2)$;*
- *$\forall l \in \mathbf{dom}(\beta), h_1(l) \sim_{k_{\text{obs}},\beta} h_2(\beta(l))$ where $h_1(l)$ and $h_2(\beta(l))$ are either two objects or two arrays.*

**Definition 4.1.7** (Output indistinguishability)**.** *Given an attacker level $k_{\text{obs}}$, a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, an output level $\vec{k_r}$, the indistinguishability of two final states in method $m$ is defined by the clauses below where $\rightarrow$ indicates logical implication :*

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad \vec{k_r}(\text{Norm}) \leq k_{\text{obs}} \Rightarrow v_1 \sim_\beta v_2}{(v_1, h_1) \sim_{k_{\text{obs}},\vec{k_r},\beta} (v_2, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad (\mathbf{class}(h_1(l_1)) : k) \in \vec{k_r} \quad k \leq k_{\text{obs}} \quad l_1 \sim_\beta l_2}{(\langle l_1 \rangle, h_1) \sim_{k_{\text{obs}},\vec{k_r},\beta} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad (\mathbf{class}(h_1(l_1)) : k) \in \vec{k_r} \quad k \not\leq k_{\text{obs}}}{(\langle l_1 \rangle, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad (\mathbf{class}(h_2(l_2)) : k) \in \vec{k_r} \quad k \not\leq k_{\text{obs}}}{(v_1, h_1) \sim_{k_{\text{obs}},\vec{k_r},\beta} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad (\mathbf{class}(h_1(l_1)) : k_1) \in \vec{k_r} \quad k_1 \not\leq k_{\text{obs}} \quad (\mathbf{class}(h_2(l_2)) : k_2) \in \vec{k_r} \quad k_2 \not\leq k_{\text{obs}}}{(\langle l_1 \rangle, h_1) \sim_{k_{\text{obs}},\vec{k_r},\beta} (\langle l_2 \rangle, h_2)}$$

For two outputs to be deemed indistinguishable, they have to be either:

- both are normal return values, and the policy for the normal return value is high;
- both are normal return values, the policy for the normal return value is low, and the values are indistinguishable;
- both are exceptions, and the policy for that particular exception is low, and the locations are indistinguishable;

- one is an exception, and the other one is a normal return value, where the policy for that particular exception is high;
- both are exceptions, and the policy for the exceptions are high;

**Definition 4.1.8** (Non-interferent DEX method)**.** *A method $m$ is* non-interferent *w.r.t. a policy $\vec{k_a} \to \vec{k_r}$, if for every attacker level $k_{\text{obs}}$, every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in \mathcal{R} \rightharpoonup \mathcal{V}$, $rt_1, rt_2 \in \mathcal{PP} \to (\mathcal{R} \to \mathcal{S})$, $h_1, h_2, h_1', h_2' \in \mathbf{Heap}$, $v_1, v_2 \in \mathcal{V} + \mathcal{L}$ s.t.*

$$\begin{array}{ll} \langle 1, \rho_1, h_1\rangle \rightsquigarrow_m^+ v_1, h_1' & h_1 \sim_{k_{\text{obs}},\beta} h_2 \\ \langle 1, \rho_2, h_2\rangle \rightsquigarrow_m^+ v_2, h_2' & \rho_1 \sim_{k_{\text{obs}},rt_1,rt_2,\beta} \rho_2 \end{array}$$

*there exists a partial function $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ s.t. $\beta \subseteq \beta'$ and*

$$(v_1, h_1') \sim_{k_{\text{obs}},\vec{k_r},\beta'} (v_2, h_2').$$

**Definition 4.1.9** (Side effect preorder)**.** *Two heaps $h_1, h_2 \in \mathbf{Heap}$ are side effect preordered with respect to a security level $k \in \mathcal{S}$ (written as $h_1 \preceq_k h_2$) if and only if $\mathbf{dom}(h_1) \subseteq \mathbf{dom}(h_2)$ and $h_1(l).f = h_2(l).f$ for all location $l \in \mathbf{dom}(h_1)$ and all fields $f \in \mathcal{F}$ such that $k \not\leq \mathbf{ft}(f)$.*

Side effect preorder $h \preceq_k h'$ states that a method is only allowed to modify a field or array whose security value is higher than $k$. We need this definition to ensure that there is no information flow to a field that has security lower than $k$ while executing a method. With this definition, then we can define the property in the scope of a method. The proof of the theorem will be given in Section 4.2.

**Definition 4.1.10** (Side effect safe)**.** *A method $m$ is side-effect-safe with respect to a security level $k_h$ if for all local variables $\rho \in \mathcal{R} \rightharpoonup \mathcal{V}$, for all heaps $h, h' \in \mathbf{Heap}$ and value $v \in \mathcal{V}$, $\langle 1, \rho, h\rangle \rightsquigarrow_m^+ v, h'$ implies $h \preceq_{k_h} h'$.*

**Definition 4.1.11** (Safe DEX method)**.** *A method $m$ is* safe *w.r.t. a policy $\vec{k_a} \overset{k_h}{\to} \vec{k_r}$ if $m$ is side-effect safe w.r.t. $k_h$ and $m$ is non-interferent w.r.t. $\vec{k_a} \to \vec{k_r}$.*

**Definition 4.1.12** (Safe DEX program)**.** *A program is* safe *w.r.t. a table $\Gamma$ of method signatures if every method $m$ is safe w.r.t. all policies in $\text{Policies}_\Gamma(m)$.*

**Theorem 4.1.1.** *Let $P$ be a DEX typable program w.r.t. safe CDRs ($\mathbf{region_m}, \mathbf{jun_m}$) and a table $\Gamma$ of method signatures. Then $P$ is safe w.r.t. $\Gamma$.*

### 4.1.4 Examples

Throughout our examples, we will use two security levels $L$ and $H$ to indicate low and high security level respectively. As a note, we use an abstracted line number as opposed to the actual program pointer because it does not have an impact other than the different value. We start with a simple example where a high guard is used to determine the value of a low variable.

**Example 4.1.1.** Assume that local variable 1 is of level $H$ and local variable 2 is of level $L$. Assume that $\vec{k_r}(\text{Norm})$ is low. Assume that the security environment is initially L. For now also assume that $r_3$ is the start of the registers used to simulate the stack in the DEX instructions. Consider the following JVM bytecode and its translation.

| Line | Ins | SE |
|---|---|---|
| 1 : | **push** 0 | $L$ |
| 2 : | **store** 2 | $L$ |
| 3 : | **load** 1 | $L$ |
| 4 : | **ifeq** 7 | $L$ |
| 5 : | **push** 1 | $H$ |
| 6 : | **store** 2 | $H$ |
| 7 : | **load** 2 | $L$ |
| 8 : | **return** | $L$ |

| Line | Ins | SE |
|---|---|---|
| 1 : | **const**$(r_3, 0)$ | $L$ |
| 2 : | **move**$(r_2, r_3)$ | $L$ |
| 3 : | **move**$(r_3, r_1)$ | $L$ |
| 4 : | **ifeq**$(r_3, 7)$ | $L$ |
| 5 : | **const**$(r_3, 1)$ | $H$ |
| 6 : | **move**$(r_2, r_3)$ | $H$ |
| 7 : | **move**$(r_3, r_2)$ | $L$ |
| 8 : | **move**$(r_0, r_3)$ | $L$ |
| 9 : | **return**$(r_0)$ | $L$ |

In this case, the type system for the JVM bytecode will reject this example because there is a violation in the constraint of line 6. The reasoning is that $se(i)$ for **push** 1 will be $H$. Thus the constraint will be $H \sqcup H \leq L$ which cannot be satisfied. A similar thing goes for the DEX instructions. Since $r_3$ gets its value from $r_1$ which is $H$ (line 3), the typing rule for **ifeq**$(r_3, l_1)$ states that $se$ in the region will be $H$. Even though for DEX line 6 is still typable, it already carries with it the information flow which will be captured in line 9, hence the program is also rejected.

There is a lift mechanism in JVM, but there is no such mechanism in DEX. The following example shows how such program in JVM can lead to an interference while in DEX we do not have such problem.

**Example 4.1.2.** Assume that local variable 1 is of level $H$, $\vec{k_r}(\text{Norm})$ is low and security environment is initially low. For now, also assume that $r_2$ is the start of the registers used to simulate the stack in the DEX instructions. Consider the following JVM bytecode and its translation.

| Line | Ins | SE |
|---|---|---|
| 1 : | **push** 0 | $L$ |
| 2 : | **push** 1 | $L$ |
| 3 : | **load** 1 | $L$ |
| 4 : | **ifeq** 8 | $L$ |
| 5 : | **swap** | $H$ |
| 6 : | **pop** | $H$ |
| 7 : | **goto** 9 | $H$ |
| 8 : | **pop** | $H$ |
| 9 : | **return** | $L$ |

| Line | Ins | SE |
|---|---|---|
| 1 : | **const**$(r_2, 0)$ | $L$ |
| 2 : | **const**$(r_3, 1)$ | $L$ |
| 3 : | **move**$(r_4, r_1)$ | $L$ |
| 4 : | **ifeq**$(r_4, 9)$ | $L$ |
| 5 : | **move**$(r_4, r2)$ | $H$ |
| 6 : | **move**$(r_5, r_3)$ | $H$ |
| 7 : | **move**$(r_2, r5)$ | $H$ |
| 8 : | **move**$(r_3, r_4)$ | $H$ |
| 9 : | **move**$(r_0, r_2)$ | $L$ |
| 10 : | **return**$(r_0)$ | $L$ |

In JVM, if there is no lift mechanism attached to the branching instruction (line

4) then observer will be able to deduce the value of local variable 1 which should be a secret, hence an information flow. With the addition of the lift mechanism, all of the values in the stack will be high. Therefore it will be rejected at line 9 since it cannot satisfy the constraint $L \sqcup H \leq L$ ($se(i)k \leq \vec{k_r}(\text{Norm})$ where $k$ is the security level of the value). In DEX, even though we do not have lift mechanism, the type checker will reject the program. This is because in the high region, the move instruction to $r_2$ (line 7) will result in the security environment of $r_2$ becomes high. Then it will follow that $r_0$ will also be high, and it will violate the constraint $L \sqcup H \leq L$ ($se(i) \sqcup rt(r) \leq \vec{k_r}(\text{Norm})$).

The following example illustrates one of the types of the interference caused by modification of low fields of a high object aliased to a low object.

**Example 4.1.3.** Assume that $\vec{k_a} = \{r_1 \mapsto H, r_2 \mapsto H, r_3 \mapsto L\}$ (which means local variable 1 has security level high, local variable 2 has security level high and local variable 3 has security level low), and that the security environment is initially low. Assume that $r_4$ is the start of the registers used to simulate the stack in the DEX instructions. Also the field $f$ is low ($\mathbf{ft}(f) = L$).

| **Line** | **Ins** | **SE** | **Line** | **Ins** | **SE** |
|---|---|---|---|---|---|
| 1 : | **new** $C$ | $L$ | 1 : | **new**$(r_4, C)$ | $L$ |
| 2 : | **store** 3 | $L$ | 2 : | **move**$(r_3, r_4)$ | $L$ |
| 3 : | **load** 2 | $L$ | 3 : | **move**$(r_4, r_2)$ | $L$ |
| 4 : | **ifeq** 7 | $L$ | 4 : | **ifeq**$(r_4, 7)$ | $L$ |
| 5 : | **new** $C$ | $H$ | 5 : | **new**$(r_4, C)$ | $L$ |
| 6 : | **goto** 8 | $H$ | 6 : | **goto**$(8)$ | $L$ |
| 7 : | **load** 3 | $H$ | 7 : | **move**$(r_4, r_3)$ | $L$ |
| 8 : | **store** 1 | $L$ | 8 : | **move**$(r_1, r_4)$ | $L$ |
| 9 : | **load** 1 | $L$ | 9 : | **move**$(r_4, r_1)$ | $L$ |
| 10 : | **push** 1 | $L$ | 10 : | **const**$(r_5, 1)$ | $L$ |
| 11 : | **putfield** $f$ | $L$ | 11 : | **iput**$(r_5, r_4, f)$ | $L$ |

The above JVM bytecode and its translation will be rejected by the type system for the JVM bytecode because there is a constraint with the security level of the object for **putfield** $f$ at line 11. In this case, the **load** 1 instruction (line 9) will push a reference of the object with high security level. Therefore, the constraint that $L \sqcup H \sqcup L \leq L$ ($k_1 \sqcup k_2 \sqcup se(i) \leq \mathbf{ft}(f)$ where $k_1$ is the security level of the value and $k_2$ is the security level of the object) cannot be satisfied. The same goes for the DEX type system; it will also reject the translated program. The reasoning is that the **move**$(r_4, r_1)$ instruction (line 9) will copy a reference to the object stored in $r_1$ which has a high security level, therefore $rt(r_4) = H$. Then, at the **iput**$(r_5, r_4, f)$ we won't be able to satisfy $L \sqcup H \sqcup L \leq L$ ($rt(r_s) \sqcup rt(r_o) \sqcup se(i) \leq \mathbf{ft}(f)$).

This next example shows that the type system also handles information flow through exceptions.

**Example 4.1.4.** Assume that $\vec{k_a} = \{r_1 \mapsto H, r_2 \mapsto L, r_3 \mapsto H\}$, security environment is initially low, and $\vec{k_r} = \{\text{Norm} \mapsto L\}$. **Handler**$(l_2, \text{np}) = 9$, and for any $e$ other than np,**Handler**$(l_2, e) \uparrow$. The following JVM bytecode and its translation will be rejected by the typing system for the JVM bytecode.

| Line | Ins | SE | Line | Ins | SE |
|---|---|---|---|---|---|
| 1 : | **load** 1 | $L$ | 1 : | **move**$(r_4, r_1)$ | $L$ |
| 2 : | **ifeq** 6 | $L$ | 2 : | **ifeq**$(r_4, l_2)$ | $L$ |
| 3 : | **new** $C$ | $H$ | 3 : | **new**$(r_4, C)$ | $H$ |
| 4 : | **store** 3 | $H$ | 4 : | **move**$(r_3, r_4)$ | $H$ |
| 5 : | **load** 3 | $H$ | 5 : | **move**$(r_4, r_3)$ | $H$ |
| 6 :: | **invokevirtual** $m$ | $L$ | 6 : | **invoke**$(1, m, r_4)$ | $L$ |
| 7 : | **push** 0 | $L$ | 7 : | **const**$(r_4, 0)$ | $L$ |
| 8 : | **return** | $L$ | 8 : | **return**$(r_4)$ | $L$ |
| 9 : | **push** 1 | $H$ | 9 : | **const**$(r_4, r_1)$ | $H$ |
| 10 : | **return** | $H$ | 10 : | **return**$(r_4)$ | $H$ |

The reason is that the typing constraint for the **invokevirtual** will be separated into several tags, and on each tag of execution we will have *se* as high (because the local variable 3 is high). Therefore, when the program reaches **return** (line 8 and 10) the constraint $H \leq L$ is violated since we have $\vec{k_r}(n) = L$, thus the program is rejected. Similar reasoning holds for the DEX type system as well, in that the **invoke** will have *se* high because the object on which the method is invoked upon is high, therefore the typing rule will reject the program because it can not satisfy the constraint when the program is about to return value from $r_4$ (constraint $H \leq L$ is violated, where $H$ comes from lub with *se*).

**Example 4.1.5.** Let local variable 1 be of high security level, the security environment initially is low, and register $r_2$ is the first register used to simulate the stack.

| Line | Ins | SE | Line | Ins | SE |
|---|---|---|---|---|---|
| 1 : | **new** $C$ | $L$ | 1 : | **new**$(r_2, C)$ | $L$ |
| 2 : | **push** 0 | $L$ | 2 : | **const**$(r_3, 0)$ | $L$ |
| 3 : | **load** 1 | $L$ | 3 : | **move**$(r_4, r_1)$ | $L$ |
| 4 : | **binop** *add* | $L$ | 4 : | **binop**$(add, r_3, r_3, r_4)$ | $L$ |
| 5 : | **ifeq** 8 | $L$ | 5 : | **ifeq**$(r_3, 8)$ | $L$ |
| 6 : | **new** $C$ | $H$ | 6 : | **new**$(r_3, C)$ | $H$ |
| 7 : | **getfield** $f$ | $H$ | 7 : | **iget**$(r_3, r_3, f)$ | $H$ |
| 8 : | **putfield** $f$ | $L$ | 8 : | **iput**$(r_3, r_2, f)$ | $L$ |
| 9 : | **push** 5 | $L$ | 9 : | **const**$(r_2, 5)$ | $L$ |
| 10 : | **newarray** $t$ | $L$ | 10 : | **newarray**$(r_2, r_2, t)$ | $L$ |
| 11 : | **arraylength** | $L$ | 11 : | **arraylength**$(r_2, r_2)$ | $L$ |
| 12 : | **return** | $L$ | 12 : | **move**$(r_0, r_2)$ | $L$ |
| | | | 13 : | **return**$(r_0)$ | $L$ |

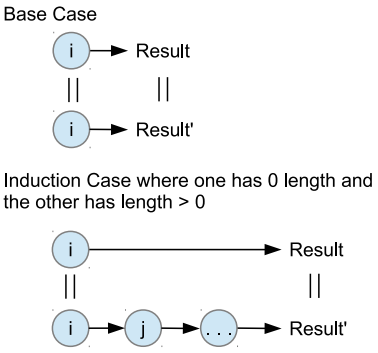



Figure 4.4: Base Case for Type System Soundness

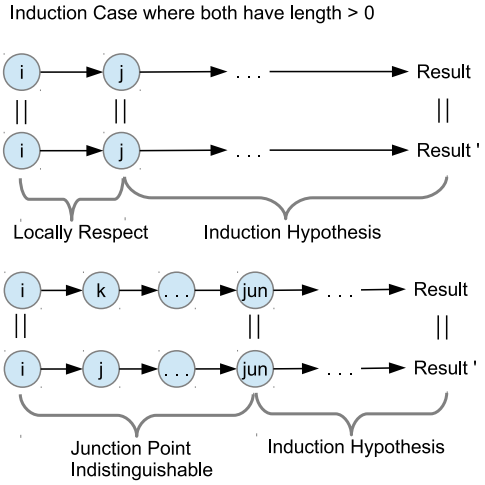



Figure 4.5: Induction Case for Type System Soundness

On the other hand, if the program does not have any information flow problem, it will pass the type checker easily. This last example type checks because even though it uses a variety of instructions, there is no interference.

## 4.2 Typable DEX Program Implies Non-Interference

In this section, we present the soundness of our type system for DEX programs, i.e., typable DEX program implies that the program is safe. Following Barthe et al., we structure the presentation of the proof based on the fragments of DEX in increasing complexity. In the paper, we present the type system for the aggregate of the submachines. In the proof construction, we will have four submachines: standard instruction without modifying the heap ($\text{DEX}_{\mathcal{I}}$), object and array instructions ($\text{DEX}_{\mathcal{O}}$), method invocation ($\text{DEX}_{\mathcal{C}}$), and exception mechanism ($\text{DEX}_{\mathcal{G}}$).

Even though we base our proof based on the work of Barthe et al., our proof is substantially different. We first outline how the type soundness proof in their work,

and then detail how it is different from ours. The type system soundness proof for JVM relies on two main lemmas:

**Locally respect:** a lemma which states that executing an instruction on two indistinguishable stack types with the same program point will yield two indistinguishable stack types.

**Step Consistent:** a lemma which states that executing an instruction with a high stack type in a high region will yield a high stack type which will be indistinguishable from the stack type of the source instruction.

And two other additional lemmas:

**High Branching:** a lemma which states that the security environment in the region of branching instruction with a high guard will be high.

**High Step:** a lemma which states executing an instruction in a high region on a high stack type will yield a high stack type (a stack type where all elements of the stack are high).

They prove non-interference by induction on the length of two executions starting from the same program point and two indistinguishable states. In the base case (See Figure 4.4) where both are return points or one of the execution trace is a return point, the non-interference is proven by locally respect lemma. In the induction case (See Figure 4.5), if the successors are of the same program point, then we can just use locally respect to prove indistinguishability and then use the induction hypothesis as the length of the execution is shorter. If the successors are of different program points, then there are two cases:

- there is a junction point for both: in this case, we use high branching, step consistent, symmetry, and transitivity of the stack type to show that the stack type is indistinguishable on the junction point. We can then use the induction hypothesis to conclude the proof;
- no junction point exists: in which case we can show that the stack type will always be high in the region (using high step) and the return value will have high security level, hence are indistinguishable.

At first glance, it looks like that we can adopt the proof directly to the DEX type system. We also prove the soundness of the DEX type system using induction on the execution trace. However, in the JVM type system, we have a lift mechanism (See Section 3.3) to maintain the high stack type and its indistinguishability, whereas the DEX type system does not have such mechanism. Without such a mechanism, even in the high region, the modified register might cause the indistinguishability to break. In particular, if initially the register has low security level and then it is updated with high security level. Clearly in the DEX we see that they are not indistinguishable since at that particular register we are comparing a low security level and a high security level. This means that to be indistinguishable, the register necessarily has

to hold the same value, which obviously we can not guarantee. In the JVM, we can claim indistinguishability since both of the stack types are high; therefore they are by definition indistinguishable. In the DEX, with a possibility of such modification, we do not have step consistent to claim the indistinguishability on the junction point anymore.

To be more explicit about this, we present a naive formulation of the step consistent lemma (Lemma 4.2.1) following the one in the JVM. We omit the part where the registers typing is high because we do not have lift mechanism. If we put this restriction on, practically the lemma becomes unusable. Now in this formulation, assume that all the conditions of the lemma are fulfilled, the instruction at $pc(s_1)$ is $\mathbf{const}(r, v)$ for arbitrary $v$ and any $r$ where $rt_1(r)$ is low and $s_1 \sim_{k_{\text{obs}}, rt_1, rt_2, \beta} s_2$, which implies that that $\rho_1(r) = \rho_2(r) = v'$ for arbitrary v' which may not be the same as $v$. Now the moment we execute this instruction, we have that $rt'_1(r)$ is high and $\rho'_1(r) = v$ which may give us $\rho'_1(r) \neq \rho_2(r)$, hence we have $s_1 \not\sim_{k_{\text{obs}}, rt_1, rt_2, \beta} s_2$.

**Lemma 4.2.1** (Naive Step Consistent in DEX)**.** *if we have* $s_1 \sim_{k_{\text{obs}}, rt_1, rt_2, \beta} s_2$ *and* $s_1 \rightsquigarrow s'_1$ *and* $pc(s_1) \vdash rt_1 \Rightarrow rt'_1$*, and security environment at program point* $pc(s_1)$ *is high, then* $s'_1 \sim_{k_{\text{obs}}, rt'_1, rt_2, \beta} s_2$.

The locally respect lemma is also broken for exception throwing instruction, so we only have a specialized version of locally respect where the successors also have the same program point. If the instruction is non-throwing instruction, we can prove the step consistent lemma based on the transfer rules (see Lemma 4.2.11, Lemma 4.2.14, and Lemma 4.2.20). However, when we deal with exception throwing instruction, we hit the same problem of not having a lift mechanism. In the JVM, the case where the two execution traces branch to a normal execution and a caught exception we still have indistinguishability since both of the stack types are high due to lift mechanism. In the DEX, caught exception will update the *ex* register, and normal execution may update a particular register. Clearly, these updates may cause the register typing to be distinguishable.

Here we give again the naive formulation of the locally respect lemma in JVM (Lemma 4.2.2) and show an example of how it breaks without the lift mechanism. Let us assume that the conditions for the lemma are fulfilled. The instruction at $i$ is $\mathbf{iget}(r, r_o, f)$, and $rt_1(r) = rt_2(r) = low$, $\rho_1(r_o)$ is null and $\rho_2(r_o)$ is not null. The difference in the value of $r_o$ implies that $rt_1(r_o) = rt_2(r_o) = high$. Based on the transfer rules, $s_1$ will continue to the next execution where $rt'_1(r) = high$, but $s_2$ will be transferred to the null pointer exception handler with $rt'_2(r) = low$, hence we already have $s'_1 \sim_{k_{\text{obs}}, rt'_1, rt'_2, \beta} s'_2$.

**Lemma 4.2.2** (Naive Locally Respect in DEX)**.** *If* $s_1 \sim_{k_{\text{obs}}, rt_1, rt_2, \beta} s_2$ *and* $pc(s_1) = pc(s_2) = i$*, and* $s_1 \rightsquigarrow s'_1$*,* $s_2 \rightsquigarrow s'_2$*,* $i \vdash rt_1 \Rightarrow rt'_1$*, and* $i \vdash rt_2 \Rightarrow rt'_2$*, then* $s'_1 \sim_{k_{\text{obs}}, rt'_1, rt'_2, \beta} s'_2$.

To prove the indistinguishability of register type on junction point, we appeal (as opposed to locally respect and step consistent ) to the following observations on the branching instruction which has indistinguishable register type:

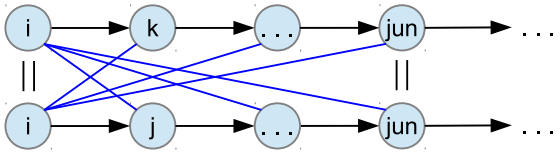



Figure 4.6: Junction Point Indistinguishability in JVM

- any change to the register in the high region can only upgrade the register's security level;
- it is also the case that if the register is not changed, then they will not affect indistinguishability. If the security level is high, then it trivially holds. Otherwise, if the security level is low, then both of $\rho$ will have the same value.

In the case of there is a change to the register in the high region, we know that they will have a high security level. But, in order to claim the indistinguishability, we have to know the other register typing (or to which program point we are comparing it). Fortunately since what we need is indistinguishability of the junction point we can claim the indistinguishability. This is because we know that there can only be one junction point defined, so we are two register type on a junction point, which will be the same. Since for all registers it does not matter whether they are changed or not, we know that at the junction point they will have indistinguishable register type; thus the proof can proceed as Barthe et al.'s proof.

In short, the difference in our proofs lies in how we prove junction point indistinguishability. In JVM, they prove the junction point indistinguishability by the step consistent lemma, symmetry, and transitivity of the indistinguishability relationship (See Figure 4.6). In DEX, we rely on the properties of both changed and unchanged registers throughout the execution to the junction point.

The type soundness for Cassandra also uses very similar proof to Barthe et al. (hence their proof is essentially different compared to ours), with the locally respect and step consistent. Interestingly they also do not have lift mechanism in their type system. However, they have a different definition of register indistinguishability compared to our definition, therefore they can claim those two lemmas. In particular, for a program point $i$ and $j$ where $i \mapsto j$, their indistinguishability is defined as $s_i \sim_{RT_j} s_j$ (only depending on the register type of $j$) whereas we define our indistinguishability as $s_i \sim_{RT_i,RT_j} s_j$ (depending on the register types of both $i$ and $j$).

We now proceed detail the proof of our type system soundness.

### 4.2.1 Auxilliary Lemmas

These lemmas are useful to prove the soundness of the DEX type system.

**Lemma 4.2.3.** *Let $k \in S$ be a security level. Then for all heap $h \in$ **Heap** and object / array $o \in \mathcal{O}$ (or $o \in \mathcal{A}$), $h \preceq_k h \oplus \{\mathbf{fresh}(h) \mapsto o\}$.*

**Lemma 4.2.4.** *For all heaps $h, h_0 \in \mathbf{Heap}$, object $o \in \mathcal{O}$ and $l = \mathbf{fresh}(h)$, $h \sim_{k_{\text{obs}},\beta} h_0$ implies $h \oplus \{l \mapsto o\} \sim_{k_{\text{obs}},\beta} h_0$.*

**Lemma 4.2.5.** *For all heaps $h, h_0 \in \mathbf{Heap}$ and $\mathbf{ft}(f) \not\leq k_{\text{obs}}$, $h \sim_\beta h_0$ implies $h \oplus \{l \mapsto h(l) \oplus \{f \mapsto v\}\} \sim_{k_{\text{obs}},\beta} h_0$.*

**Lemma 4.2.6.** *For all heaps $h, h_0 \in \mathbf{Heap}$, $r \in R$, $\rho \in (R \rightharpoonup V)$, $rt \in (\mathcal{R} \to \mathcal{S})$, $\rho(r) \in \mathbf{dom}(h)$, $\rho(r)$ is an array, integer $0 \leq i < h(\rho(r)).\mathbf{length}$, $rt(\rho(r)) = k[k_c]$ and $k_c \not\leq^{\text{ext}} k_{\text{obs}}$, $h \sim_{k_{\text{obs}},\beta} h_0$ implies $h \oplus \{\rho(r) \mapsto h(\rho(r)) \oplus \{i \mapsto v\}\} \sim_{k_{\text{obs}},\beta} h_0$.*

**Lemma 4.2.7.** *For all heaps $h, h', h_0 \in \mathbf{Heap}$, $k \not\leq k_{\text{obs}}$, and $h \preceq_k h'$, $h \sim_{k_{\text{obs}},\beta} h_0$ implies $h' \sim_{k_{\text{obs}},\beta} h_0$.*

**Lemma 4.2.8.** *If $h_1 \sim_\beta h_2$, if $l_1 = \mathbf{fresh}(h_1)$ and $l_2 = \mathbf{fresh}(h_2)$ then the following properties hold*

- $\forall C, h_1 \oplus \{l_1 \mapsto \mathbf{default}_C\} \sim_{k_{\text{obs}},\beta} h_2$
- $\forall C, h_1 \sim_{k_{\text{obs}},\beta} h_2 \oplus \{l_2 \mapsto \mathbf{default}_C\}$
- $\forall C, h_1 \oplus \{l_1 \mapsto \mathbf{default}_C\} \sim_{k_{\text{obs}},\beta} h_2 \oplus \{l_2 \mapsto \mathbf{default}_C\}$
- $\forall l, t, i, h_1 \oplus \{l_1 \mapsto (l, \mathbf{defaultArray}(l,t), i)\} \sim_{k_{\text{obs}},\beta} h_2$
- $\forall l, t, i, h_1 \sim_{k_{\text{obs}},\beta} h_2 \oplus \{l_2 \mapsto (l, \mathbf{defaultArray}(l,t), i)\}$
- $\forall l, t, i, l', t', i' \; h_1 \oplus \{l_1 \mapsto (l, \mathbf{defaultArray}(l,t), i)\} \sim_{k_{\text{obs}},\beta} h_2 \oplus \{l_2 \mapsto (l',$ $\mathbf{defaultArray}(l',t'), i')\}$.

**Lemma 4.2.9.** *$\rho_1 \sim_{k_{\text{obs}},rt_1,rt_2,\beta} \rho_2$ implies for any register $r \in \rho_1$ :*

- *either $rt_1(r) = rt_2(r), rt_1(r) \leq k_{\text{obs}}$ and $\rho_1(r) \sim_{k_{\text{obs}},rt_1,rt_2,\beta} \rho_2(r)$*
- *or $rt_1(r) \not\leq k_{\text{obs}}$ and $rt_2(r) \not\leq k_{\text{obs}}$.*

**Lemma 4.2.10.** *Let $\beta$ be a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in$ State be two states at the same program point $i$ and let $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ be two registers types such that $s_1 \sim_{k_{\text{obs}},rt_1,rt_2,\beta} s_2$. Let $s_1', s_2' \in$ State and $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ such that $s_1 \rightsquigarrow s_1', s_2 \rightsquigarrow s_2'$, $i \vdash rt_1 \Rightarrow rt_1'$, and $i \vdash rt_2 \Rightarrow rt_2'$, then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\forall r \notin U, \rho_1' \sim_{k_{\text{obs}},rt_1',rt_2',r,\beta} \rho_2'$ where $U$ is the set of registers updated by the instruction in $i$.*

Lemma 4.2.3 states that creating new objects or arrays does not affect side-effect preorder. Lemma 4.2.4 states that creating new objects or arrays does not affect heap indistinguishability. Lemma 4.2.5 states that if any two heaps are indistinguishable and the policy for the field is high, then any update to the field will preserve heap indistinguishability. Lemma 4.2.6 is Lemma 4.2.5 equivalent for the update on arrays. Lemma 4.2.7 states that if any two heaps $h, h'$ after executions are side effect preordered w.r.t. a high security level, the executions also preserve heap indistinguishability. Lemma 4.2.8 states that adding new default objects or arrays preserve

heap indistinguishability. Lemma 4.2.9 is the inversion of registers indistinguishability. It states that if any two registers mappings are indistinguishable, then for each register pair they are either have high security level, or they have indistinguishable values. Lemma 4.2.10 states that if two registers at the same program point are indistinguishable, then for any registers which are not updated by the operation, it will maintain indistinguishability.

### 4.2.2 Typable DEX$_\mathcal{I}$ Implies Non-interference

There are actually more definitions on indistinguishability that would be required to establish that typability implies non-interference. We have to make several notes here in this fragment of DEX. The execution is always expected to return normally. Therefore, the form of the policy for return value only takes the form of $k_r$ instead of $\vec{k_r}$. There is also no need to involve the heap and $\beta$ mapping. Hence we will drop them from the proofs.

**Definition 4.2.1** (State indistinguishability)**.** *Two states $\langle i, \rho\rangle$ and $\langle i', \rho'\rangle$ are indistinguishable w.r.t. $rt, rt' \in (\mathcal{R} \to \mathcal{S})$, denoted $\langle i, \rho\rangle \sim_{k_{\text{obs}}, rt, rt'} \langle i', \rho'\rangle$, iff $\rho \sim_{k_{\text{obs}}, rt, rt'} \rho'$*

**Lemma 4.2.11** (Locally Respects)**.** *Let $(i, \rho_1), (i, \rho_2) \in \text{State}_I$ be two DEX$_\mathcal{I}$ states at the same program point $i$ and let $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ be two register types such that $s_1 \sim_{k_{\text{obs}}, rt_1, rt_2} s_2$.*

- *Let $s_1', s_2' \in \text{State}_I$ and $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ such that $s_1 \rightsquigarrow s_1'$, $s_2 \rightsquigarrow s_2'$, $i \vdash rt_1 \Rightarrow rt_1'$, and $i \vdash rt_2' \Rightarrow rt_2'$, then $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2'} s_2'$.*
- *Let $v_1, v_2 \in \mathcal{V}$ such that $s_1 \rightsquigarrow v_1$, $s_2 \rightsquigarrow v_2$, $i \vdash rt_1 \Rightarrow$, and $i \vdash rt_2' \Rightarrow$, then $k_r \leq k_{\text{obs}}$ implies $v_1 \sim v_2$.*

*Proof.* We prove the lemma by structural induction on the instruction:

**move**$(r, r_s)$**:** this case will fall into the first case of the lemma as a program point with this instruction will not be a return point. We then need to prove that for all registers in the registers mapping, they are indistinguishable. If the register is any register $r' \neq r$, we can use Lemma 4.2.10, and we have $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', r'} \rho_2'$. If the register is $r$, we then do a case analysis on the security level on $r_s$. If the security level is high, we know that $rt_1'(r) = rt_2'(r) = high$ hence they are indistinguishable. If the security level is low, then we know by Lemma 4.2.9 that $\rho_1(r_s) = \rho_2(r_s)$ hence we will have $\rho_1'(r) = \rho_2'(r)$, which means that we still have $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2'} s_2'$.

**binop**$(r, r_a, r_b)$**:** this case will also fall into the first case of the lemma. We also prove the indistinguishability of the registers. If the register is not $r$, we can use Lemma 4.2.10. If the register is $r$, we then do a case analysis on the security level of $r_a$ and $r_b$. If either the security level of $r_a$ or $r_b$ is high (or both are high), then we know that $rt_1'(r) = rt_2'(r) = high$ hence they are indistinguishable. If the security level is low, then we know by Lemma 4.2.9 that $\rho_1(r_a) = \rho_2(r_a)$ and

$\rho_1(r_b) = \rho_2(r_b)$ hence we will have $\rho_1'(r) = \rho_2'(r)$ (because a binary operation on the same operands will always result in the same value), which means that we still have $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2'} s_2'$.

**const**$(r, v)$**:** this case will also fall into the first case of the lemma. We again prove the lemma by proving the indistinguishability on the registers. If the register is not $r$, we can use Lemma 4.2.10. If the register is $r$, we know that we still have $\rho_1'(r) = \rho_2'(r)$, which means that we still have $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2'} s_2'$.

**goto**$(t)$**:** this case will fall into the first case of the lemma, and it trivially holds since this instruction does not modify any of the registers. Hence the indistinguishability is preserved.

**ifeq**$(r, t)$**:** this case will fall into the first case of the lemma, and it trivially holds since this instruction does not modify any of the registers. Hence the indistinguishability is preserved.

**return**$(r)$**:** this is the only case in $\text{DEX}_{\mathcal{I}}$ which falls into the second case. If $k_r \not\leq k_{\text{obs}}$ there is nothing to prove as the assumption is not satisfied. If $k_r \leq k_{\text{obs}}$, then we know by definition that $rt_1(r) = rt_2(r) = low$, hence we know that $\rho_1(r) = \rho_2(r) = v$ for arbitrary value $v$. This will further give us $v \sim v$ by the definition of value indistinguishability.

Since for all of the possible instructions the lemma is satisfied, we have proven that the lemma holds. □

**Lemma 4.2.12** (High Branching)**.** *Let $s_1, s_2 \in \text{State}_I$ be two $\text{DEX}_{\mathcal{I}}$ states at the same program point $i$ and let $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ be two register types such that $s_1 \sim_{k_{\text{obs}}, rt_1, rt_2} s_2$. If two states $\langle i_1, \rho_1' \rangle, \langle i_2, \rho_2' \rangle \in \text{State}_I$ and two register type $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ s.t. $i_1 \neq i_2$ (hence the instruction at program point $i$ is a branching instruction), $s_1 \rightsquigarrow \langle i_1, \rho_1' \rangle$, $s_2 \rightsquigarrow \langle i_2, \rho_2' \rangle$, $i \vdash rt_1 \Rightarrow rt_1'$, $i \vdash rt_2 \Rightarrow rt_2'$ then $\forall j \in \textbf{region}(i), se(j) \not\leq k_{\text{obs}}$.*

*Proof.* This holds by definition of the branching instruction (**ifeq** and **ifneq**). $se(i)$ will be high because register $r$ (the register which is involved in the branching instruction) will by definition be high. This level cannot be low, because if the level is low, then the register $r$ is low and by the definition of indistinguishability will have to have the same values, and therefore will take the same program counter. Since $se$ is high in scope of the region, we have $\forall j \in \textbf{region}(i), se(j) \not\leq k_{\text{obs}}$ and the lemma holds. □

**Lemma 4.2.13** (indistinguishablility single monotony)**.** *if $s \sim_{k_{\text{obs}}, S, T} t, S \sqsubseteq S'$ and $S$ is high then $s \sim_{k_{\text{obs}}, S', T} t$.*

### 4.2.3 Typable $\text{DEX}_{\mathcal{O}}$ Implies Non-interference

Indistinguishability between states can be defined with the additional definition of heap indistinguishability, so we do not need additional indistinguishability defini-

tions. In the $\text{DEX}_{\mathcal{O}}$ part, we only need to appropriate the lemmas used to establish the proof.

**Definition 4.2.2** (State indistinguishability)**.** *Two states $\langle i, \rho, h\rangle$ and $\langle i', \rho', h'\rangle$ are indistinguishable w.r.t. a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, and two register typing $rt, rt' \in (\mathcal{R} \to \mathcal{S})$, denoted $\langle i, \rho, h\rangle \sim_{k_{\text{obs}}, rt, rt', \beta} \langle i', \rho', h'\rangle$, iff $\rho \sim_{k_{\text{obs}}, rt, rt', \beta} \rho'$ and $h \sim_{k_{\text{obs}}, \beta} h'$ hold.*

**Lemma 4.2.14** (Locally Respects)**.** *Let $\beta$ be a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \text{State}_O$ be two $\text{DEX}_{\mathcal{O}}$ states at the same program point $i$ and let $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ be two registers types such that $s_1 \sim_{k_{\text{obs}}, rt_1, rt_2, \beta} s_2$.*

- *Let $s_1', s_2' \in \text{State}_O$ and $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ such that $s_1 \rightsquigarrow s_1', s_2 \rightsquigarrow s_2'$, $i \vdash rt_1 \Rightarrow rt_1'$, and $i \vdash rt_2 \Rightarrow rt_2'$, then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta'} s_2'$ and $\beta \subseteq \beta'$.*
- *Let $v_1, v_2 \in \mathcal{V}$ such that $s_1 \rightsquigarrow v_1$, $s_2 \rightsquigarrow v_2$, $i \vdash rt_1 \Rightarrow$, and $i \vdash rt_2' \Rightarrow$, then $k_r \leq k_{\text{obs}}$ implies $v_1 \sim_\beta v_2$.*

*Proof.* Just like in $\text{DEX}_{\mathcal{I}}$, we do a structural induction on the possible instruction. If the instruction is a $\text{DEX}_{\mathcal{I}}$ instruction, then we know that registers wise they preserve indistinguishability. Since $\text{DEX}_{\mathcal{I}}$ instruction does not modify the heap, then this lemma is proved trivially. If the instruction is not in $\text{DEX}_{\mathcal{I}}$:

**new**$(r, c)$**:** the proof for the registers indistinguishability follows that of **const**. This is one of the instructions that may add a mapping to $\beta$, therefore we know that $\beta \subseteq \beta'$ is also satisfied. Now we also know from Lemma 4.2.8 that we have $h_1' \sim_{k_{\text{obs}}, \beta'} h_2'$. Combined with the fact that we have $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta'} \rho_2'$, we have $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta'} s_2'$ (Case 1 of the lemma).

**iget**$(r, r_o, f)$**:** there is no change to the heaps, and there is no change to the $\beta$ mapping either, therefore we have $\beta \subseteq \beta'$ and $h_1' \sim_{k_{\text{obs}}, \beta'} h_2'$. We just need to prove the registers indistinguishability to prove state indistinguishability. If the register is not $r$, then we can use Lemma 4.2.10. If the register is $r$, we first make the distinction whether $r_o$ is of high security level. If $r_o$ is of high security levels, $r$ will also be updated with high security level. Hence the registers indistinguishability is preserved. In the case where $r_o$ has low security level, we make further cases on whether $\mathbf{ft}(f)$ is of high security level. If $\mathbf{ft}(f)$ is of high security level, then $r$ will be updated with high security level, which also means that the registers indistinguishability is preserved. If $\mathbf{ft}(f)$ is of low security level, we know that the field contains the same value, hence we will have $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', r, \beta} \rho_2'$. With this registers indistinguishability, we will have $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta'} s_2'$ (Case 1 of the lemma)

**iput**$(r_s, r_o, f)$**:** there is no change to the registers, hence registers indistinguishability still holds ($\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta'} \rho_2'$). Even though this instruction modifies the heap, it does not affect the $\beta$ mapping. Therefore we have $\beta \subseteq \beta'$. Now for the heaps indistinguishability, we distinguish between the reference to the object that is pointed by $r_o$. If it is not the object pointed by $r_o$, we know that they are

indistinguishable, since initially, the heaps are indistinguishable. If the object is actually the object pointed by $r_o$, we have two possibilities, either both $r_o$ contains the same object ($\rho_1(r_o) = \rho_2(r_o)$), or they point to different objects.

- In the case where they point to the same objects, we also have further two possibilities depending on the value of $se(i) \sqcup rt(r_o) \sqcup rt(r_s)$ ($rt$ is either $rt_1$ or $rt_2$). If they are of high security level, we know that to be typable $\mathbf{ft}(f)$ is necessarily of high security level. Therefore it does not affect indistinguishability. If they are of low security level, we know that $\rho_1(r_s) = \rho_2(r_s)$ and $\beta(\rho_1(r_o)) = \rho_2(r_o)$. Since the operation will update the field with the same value, it will preserve the heap indistinguishability.
- In the case where they point to different objects, we know that $rt_1(r_o)$ and $rt_2(r_o)$ will be of high security levels. Since the program is typable, it implies that $\mathbf{ft}(f)$ is of high security level. Therefore we can conclude that the update is applied to a high field which does not affect indistinguishability.

We have $h_1' \sim_{k_{\text{obs}},\beta'} h_2'$ and we have $\rho_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta'} \rho_2'$i, therefore, we can conclude that we have $s_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta'} s_2'$ (Case 1 of the lemma).

**newarray**$(r, r_l, t)$**:** This instruction possibly adds a mapping to $\beta$, so we know that we have $\beta \subseteq \beta'$. Using Lemma 4.2.8, we get $h_1' \sim_{k_{\text{obs}},\beta'} h_2'$. For the registers indistinguishability, if the register is not $r$, then we can use Lemma 4.2.10. If the register is $r$, depending on the security level of $r_l$, we have two possibilities. If the security level of $r_l$ is high, then based on the typing rule we have the security level of $r$ also to be high, thus completing the proof for $\rho_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta} \rho_2'$. If $r_l$ is low, we will have $\rho_1'(r) \sim_\beta \rho_2'(r)$ since the type and the length of the array are the same. With the registers and heap indistinguishability, we have $s_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta'} s_2'$ (Case 1 of the lemma).

**arraylength**$(r, r_a)$**:** This instruction does not modify the $\beta$ mapping and the heap, therefore we have $\beta \subseteq \beta'$ and $h_1' \sim_{k_{\text{obs}},\beta'} h_2'$ for granted. As usual, for the registers indistinguishability, we distinguish the case whether the register is $r$. If the register is not $r$, then we can use Lemma 4.2.10. If the register is $r$, we will distinguish the case further depending on whether $r_a$ has high security level or not. If $r_a$ is of high security level, we know that $r$ will also be of high security level, thus we have $\rho_1'(r) \sim_{k_{\text{obs}},rt_1',rt_2',r,\beta} \rho_2'$ thus completing the proof for $\rho_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta} \rho_2'$. If $r_a$ is of low security level, then we know that the array is the same thus we will have $\rho_1'(r) \sim_\beta \rho_2'(r)$. With the registers and heap indistinguishability, we have $s_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta'} s_2'$ (Case 1 of the lemma).

**aget**$(r, r_a, r_i)$**:** The argument for this instruction is pretty much the same as the argument for **iget**. We already have $\beta \subseteq \beta'$ and $h_1' \sim_{k_{\text{obs}},\beta'} h_2'$ since the instruction does not modify $\beta$ mapping and the heap. For registers indistinguishability, we first distinguish the case whether the register we are interested in is $r$ or not. If the register is not $r$, then we can use Lemma 4.2.10. If the register is $r$, then we distinguish the case further based on the lub of the security level of $r_a$ and

$r_i$. If the least upper bound is of high security level, then we know that $r$ will also be updated with high security level ($\rho_1' \sim_{k_{\text{obs}},rt_1',rt_2',r,\beta} \rho_2'$). If the least upper bound is low, then we know that the index and the array are indistinguishable, hence the value will also be indistinguishable ($\rho_1'(r) \sim_\beta \rho_2'(r)$). This concludes the proof of $s_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta'} s_2'$ (Case 1 of the lemma).

**aput**$(r_s, r_a, r_i)$**:** The argument for this instruction is pretty much the same as the argument for **iput**. This instruction does not modify $\beta$ mapping and the registers, therefore we have $\beta \subseteq \beta'$ and $\rho_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta} \rho_2'$ for granted. Now for the heaps indistinguishability, we distinguish between the reference to the array that is pointed by $r_a$. If it is not the object pointed by $r_a$, we know that they are indistinguishable, since initially, the heaps are indistinguishable. If the array is actually the array pointed by $r_a$, we have two possibilities, either both $r_a$ contains the same arrays ($\rho_1(r_a) = \rho_2(r_a)$), or they point to different arrays.

- In the case where they point to the same arrays, we also have further two possibilities depending on the value of $(se(i) \sqcup rt(r_a)) \sqcup^{\text{ext}} rt(r_s)$ ($rt$ is either $rt_1$ or $rt_2$). If they are of high security level, we know that to be typable the security level of the array content is necessarily of high security level, therefore it does not affect indistinguishability. If they are of low security level, we know that $\rho_1(r_s) = \rho_2(r_s)$ and $\rho_1(r_a) \sim_\beta \rho_2(r_a)$. Since the operation will update the array content with the same value, it will preserve the heap indistinguishability.
- In the case where they point to different arrays, we know that $rt_1(r_a)$ and $rt_2(r_a)$ will be of high security levels. Since the program is typable, it implies that the security level of the content is also high which does not affect indistinguishability.

We have $h_1' \sim_{k_{\text{obs}},\beta'} h_2'$ and we have $\rho_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta'} \rho_2'$, therefore, we can conclude that we have $s_1' \sim_{k_{\text{obs}},rt_1',rt_2',\beta'} s_2'$ (Case 1 of the lemma).

All of the possible instructions maintain state indistinguishability, therefore the lemma holds. □

**Lemma 4.2.15** (High Branching)**.** *Let $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial function, $s_1, s_2 \in \text{State}_O$ be two $DEX_{\mathcal{O}}$ states at the same program point $i$ and let two registers types $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ such that $s_1 \sim_{k_{\text{obs}},rt_1,rt_2,\beta} s_2$. Let two states $\langle i_1, \rho_1', h_1' \rangle, \langle i_2, \rho_2', h_2' \rangle \in \text{State}_O$ and two register type $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ s.t. $i_1 \neq i_2, s_1 \rightsquigarrow \langle i_1, \rho_1', h_1' \rangle$, $s_2 \rightsquigarrow \langle i_2, \rho_2', h_2' \rangle$. If $i \vdash rt_1 \Rightarrow rt_1', i \vdash rt_2 \Rightarrow rt_2'$ then $\forall j \in \textbf{region}(i), se(j) \not\leq k_{\text{obs}}$.*

*Proof.* This lemma is trivially true since there is no branching instruction in $\text{DEX}_{\mathcal{O}}$. □

### 4.2.4 Typable $\text{DEX}_{\mathcal{C}}$ Implies Security

The notion of a secure program is now also defined with *side-effect-safety* due to a method invocation. Therefore we also need to establish that typable program im-

plies it is *side-effect-safe*. We show this by showing the property that all instruction transforms a heap $h$ into a heap $h'$ s.t. $h \preceq_{k_h} h'$.

**Lemma 4.2.16.** *Let* $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle \in \text{State}_C$ *be two states s.t.* $\langle i, \rho, h\rangle\rangle \rightsquigarrow_m \langle i', \rho', h'\rangle$. *Let two register types* $rt, rt' \in (\mathcal{R} \to \mathcal{S})$ *s.t.* $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash^{\text{Norm}} rt \Rightarrow rt'$ *and* $P[i] \neq \mathbf{invoke}$, *then* $h \preceq_{k_h} h'$

*Proof.* The only instruction that can cause this difference is **newarray**, **new**, **iput**, and **aput**. For creating new objects or arrays, Lemma 4.2.3 shows that they still preserve the side-effect-safety. For **iput**, the transfer rule implies $k_h \leq \mathbf{ft}(f)$. Since there will be no update such that $k_h \not\leq \mathbf{ft}(f)$, $h \preceq_{k_h} h'$ holds. □

**Lemma 4.2.17.** *Let* $\langle i, \rho, h\rangle \in \text{State}_C$ *be a state,* $h' \in \mathbf{Heap}$, *and* $v \in V$ *s.t.* $\langle i, \rho, h\rangle \rightsquigarrow v, h'$. *Let* $rt \in (\mathcal{R} \to \mathcal{S})$ *s.t.* $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash^{\text{Norm}} rt \Rightarrow$, *then* $h \preceq_{k_h} h'$.

*Proof.* This only concerns the **return** instruction at the moment. And it's clear that the return instruction will not modify the heap, therefore $h \preceq_{k_h} h'$ holds. □

**Lemma 4.2.18.** *For all methods m in P, let* $(\mathbf{region_m}, \mathbf{jun_m})$ *be a safe CDR for m. Suppose all methods m in P are typable with respect to* $\mathbf{region_m}$ *and all signatures in* $\mathbf{Policies_\Gamma}(m)$. *Let* $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle \in \text{State}_C$ *be two states s.t.* $\langle i, \rho, h\rangle \rightsquigarrow_m \langle i', \rho', h'\rangle$. *Let two register types* $rt, rt' \in (\mathcal{R} \to \mathcal{S})$ *s.t.* $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash^{\text{Norm}} rt \Rightarrow rt'$ *and* $P[i] = \mathbf{invoke}$, *then* $h \preceq_{k_h} h'$

*Proof.* Assume that the method called by **invoke** is $m_0$. The instructions contained in $m'$ can be any of the instructions in DEX, including another **invoke** to another method. Since we are not dealing with termination / non-termination, we can assume that for any instruction **invoke** called, it will either return normally or throw an exception. Therefore, for any method $m_0$ called by **invoke**, there can be a chain of one or more calls

$$m_0 \rightsquigarrow m_1 \rightsquigarrow ... \rightsquigarrow m_n$$

where $m \rightsquigarrow m'$ signifies that an instruction in method $m$ calls $m'$. Since the existence of such a call chain is assumed, we can use induction on the length of the longest call chain. The base case would be the length of the chain is 0, which means we can just invoke Lemma 4.2.16 and Lemma 4.2.17 because all the instructions contained in this method $m_0$ will fall to either one of the two above case.

The induction step is when we have a chain with length 1 or more and we want to establish that assuming the property holds when the length of call chain is $n$, then the property also holds when the length of call chain is $n+1$. In this case, we just examine possible instructions in $m_0$, and proceed like the base case except that there is also a possibility that the instruction is **invoke** on $m_1$. Since the call chain is necessarily shorter now $m_0 \rightsquigarrow m_1$ is dropped from the call chain, we know that **invoke** on $m_1$ will fulfill *side-effect-safety*. Since all possible instructions maintain *side-effect-safety*, we know that this lemma holds. □

Since all instructions in a typable program maintain *side-effect-safety*, then we can state the lemma saying that typable program will be *side-effect-safe*.

**Lemma 4.2.19.** *For all methods m in P, let (*$\mathbf{region_m}, \mathbf{jun_m}$*) be a safe CDR for m. Suppose all methods m in P are typable with respect to* $\mathbf{region_m}$ *and to all signatures in* $\mathbf{Policies_\Gamma}(m)$*. Then all methods m are* side-effect-safe *w.r.t. the heap effect level of all the policies in* $\mathbf{Policies_\Gamma}(m)$*.*

Then, like the previous machine, we need to appropriate the unwinding lemmas. The unwinding lemmas for $\text{DEX}_\mathcal{O}$ stay the same, and the one for instruction **moveresult** is straightforward. Fortunately, **Invoke** is not a branching source, so we don't need to appropriate the high branching lemma for this instruction (it will be needed for exception throwing one in the subsequent machine).

**Lemma 4.2.20** (Locally Respect Lemma)**.** *Let P be a program and* $\Gamma$ *be a table of signature s.t. all of its methods* $m'$ *are* non-interferent *w.r.t. all the policies in* $\mathbf{Policies_\Gamma}(m')$ *and* side-effect-safe *w.r.t. the heap effect level of all the policies in* $\mathbf{Policies_\Gamma}(m')$*. Let m be a method in P,* $\beta \in L \rightharpoonup L$ *a partial function,* $s_1, s_2 \in \text{State}_C$ *two* $DEX_\mathcal{C}$ *states at the same program point i and two registers types* $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ *s.t.* $s_1 \sim_{k_{\text{obs}}, rt_1, rt_2, \beta} s_2$*. If there exist two states* $s_1'.s_2' \in \text{State}_C$ *and two registers types* $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ *s.t.*

$$s_1 \rightsquigarrow_m s_1' \quad \textit{and} \quad \Gamma, \text{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash rt_1 \Rightarrow rt_1'$$

*and*

$$s_2 \rightsquigarrow_{m'} s_2' \quad \textit{and} \quad \Gamma, \text{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash rt_2 \Rightarrow rt_2'$$

*then there exists* $\beta' \in L \rightharpoonup L$ *s.t.* $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta'} s_2'$ *and* $\beta \subseteq \beta'$*.*

*Proof.* We are only interested in **invoke** and **moveresult**. We first start by proving that **invoke** maintain state indistinguishability. This instruction may modify the registers and the heap, therefore we have to prove each indistinguishability to claim state indistinguishability. **invoke** only modifies the pseudo-register *ret* with the values that will be dependent on the security of the return value. Because we know that the method invoked is non-interferent and the arguments are indistinguishable, therefore we can conclude that the result will be indistinguishable as well. This will give us $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', ret, \beta} \rho_2'$, which we can combine with Lemma 4.2.10 for the rest of the registers to obtain $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta} \rho_2$ (**invoke** only updates register *ret*). Output indistinguishability also implies that we have $h_1' \sim_{k_{\text{obs}}, \beta} h_2'$. Since we have both $h_1' \sim_{k_{\text{obs}}, \beta} h_2'$ and $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta} \rho_2'$, we have $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta} s_2'$.

**Moveresult**$(r)$ only updates register $r$, therefore we already have $\beta \subseteq \beta'$ and $h_1' \sim_{k_{\text{obs}}, \beta} h_2'$ by definition. Now if the register is not $r$, we can use Lemma 4.2.10 to show indistinguishability. If the register is $r$, then we distinguish the case based on whether the security level of *ret*. If it is high, then we know that the security level of $r$ will be updated to high as well, hence we have $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', r, \beta} \rho_2'$. If the security level of $r$ is low, we know that $\rho_1'(ret) \sim_\beta \rho_2'(ret)$, therefore we will also have $\rho_1'(r) \sim_\beta \rho_2'(r)$, which in turn will give us $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', r, \beta} \rho_2'$. Since we have both $h_1' \sim_{k_{\text{obs}}, \beta} h_2'$ and $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta} \rho_2'$, we have $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta} s_2'$

□

### 4.2.5 Typable DEX$_{\mathcal{G}}$ Implies Security

Like the one in DEX$_{\mathcal{C}}$, we also need to firstly prove the *side-effect-safety* of a program if it's typable. Fortunately, this proof extends almost directly from the one in DEX$_{\mathcal{C}}$. The only difference is that there is a possibility of invoking a function which throws an exception and the addition of **throw** instruction. The proof for invoking a function which throws an exception is the same as the usual **invoke** because we are not concerned about whether the returned value $r$ is in $L$ or in $V$. The one case for **throws** we use the same Lemma 4.2.3 as it differs only in the allocation of exception in the heap. The complete definition :

**Lemma 4.2.21.** *Let* $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle \in \text{State}_G$ *be two states s.t.* $\langle i, \rho, h\rangle \rightsquigarrow_m \langle i', \rho', h'\rangle$. *Let two registers types* $rt, rt' \in (\mathcal{R} \to \mathcal{S})$ *s.t.* $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash rt \Rightarrow rt'$ *and* $P[i] \neq \mathbf{invoke}$, *then* $h \preceq_{k_h} h'$.

*Proof.* The only instructions that can cause this difference are array / object manipulation instructions that throw a null pointer exception. For creating new objects or arrays and allocating the space for an exception, Lemma 4.2.3 shows that they still preserve the *side-effect-safety*. **throw** instruction itself does not allocate space for an exception, so there is no modification to the heap. □

**Lemma 4.2.22.** *Let* $\langle i, \rho, h\rangle \in \text{State}_G$ *be a state,* $h' \in \mathbf{Heap}$, *and* $v \in V$ *s.t.* $\langle i, \rho, h\rangle \rightsquigarrow v, h'$. *Let* $rt \in (\mathcal{R} \to \mathcal{S})$ *s.t.* $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash rt \Rightarrow$, *then* $h \preceq_{k_h} h'$.

*Proof.* This can only be one of two cases, either it is **return** instruction or an uncaught exception. For return instruction, it's clear that it will not modify the heap, therefore, $h \preceq_{k_h} h'$ holds. For uncaught exception, the only difference is that we first need to allocate the space on the heap for the exception, and again we use Lemma 4.2.3 to conclude that it will still make $h \preceq_{k_h} h'$ hold □

**Lemma 4.2.23.** *For all method* $m \in P$, *let* ($\mathbf{region}_m$, $\mathbf{jun}_m$) *be a safe CDR for* $m$. *Suppose all methods* $m \in P$ *are typable w.r.t.* $\mathbf{region}_m$ *and all signatures in* $\mathbf{Policies}_{\mathbf{\Gamma}}(m)$. *Let* $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle \in \text{State}_G$ *be two states s.t.* $\langle i, \rho, h\rangle \rightsquigarrow_m \langle i', \rho', h'\rangle$. *Let two register types* $rt, rt' \in (\mathcal{R} \to \mathcal{S})$ *s.t.* $\mathbf{region}$, $se$, $\vec{k_a} \overset{k_h}{\to} k_r$, $i \vdash rt \Rightarrow rt'$ *and* $P[i] = \mathbf{invoke}$, *then* $h \preceq_{k_h} h'$.

*Proof.* In the case of **invoke** instruction executing normally, we can refer to the proof in Lemma 4.2.18. In the case of an exception, if it is caught, then we can follow the same reasoning in Lemma 4.2.21. In the case of an uncaught exception, it will fall to Lemma 4.2.22. □

**Lemma 4.2.24.** *For all method* $m \in P$, *let* ($\mathbf{region}_m$, $\mathbf{jun}_m$) *be a safe CDR for* $m$. *Suppose all methods* $m \in P$ *are typable w.r.t.* $\mathbf{region}_m$ *and all signatures in* $\mathbf{Policies}_{\mathbf{\Gamma}}(m)$. *Then all methods* $m$ *are* side-effect-safe *w.r.t. the heap effect level of all the policies in* $\mathbf{Policies}_{\mathbf{\Gamma}}(m)$.

*Proof.* We use the definition of typable method and Lemma 4.2.21, Lemma 4.2.22, and Lemma 4.2.23. Given a typable method, for a derivation

$$\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \langle i_1, \rho_1, h_1\rangle \ldots \rightsquigarrow_{m,\tau_n} (r, h)$$

there exists $RT \in \mathcal{PP} \to (\mathcal{R} \to \mathcal{S})$ and $rt_1, \ldots rt_n \in (\mathcal{R} \to \mathcal{S})$ s.t.

$$i_0 \vdash^{\tau_0} RT_{i_0} \Rightarrow rt_1 \;\; i_1 \vdash^{\tau_1} RT_{i_1} \Rightarrow rt_2, \ldots i_n \vdash^{\tau_n} RT_{i_n} \Rightarrow$$

Using the lemmas, then we will get

$$h_0 \preceq_{k_h} h_1 \preceq_{k_h} \cdots \preceq h_n \preceq h$$

Now we can use the transitivity of $\preceq_{k_h}$ to conclude that $h_0 \preceq_{k_h} h$ (the definition of *side-effect-safety*). □

**Definition 4.2.3** (High Result)**.** *Given* $(r,h) \in (V+L) \times \mathbf{Heap}$ *and an output level* $\vec{k_r}$*, the predicate* $\mathbf{highResult}_{\mathbf{k_r}}(r,h)$ *is defined as :*

$$\frac{\vec{k_r}(n) \not\leq k_{\text{obs}} \quad v \in V}{\mathbf{highResult}_{\mathbf{k_r}}(v,h)} \qquad \frac{\vec{k_r}(\mathbf{class}(h(l))) \not\leq k_{\text{obs}} \quad l \in \mathbf{dom}(h)}{\mathbf{highResult}_{\mathbf{k_r}}(\langle l \rangle, h)}$$

**Definition 4.2.4** (Typable Execution)**.**

- *An execution step* $\langle i,\rho,h\rangle \rightsquigarrow_{m,\tau} \langle i',\rho',h'\rangle$ *is typable w.r.t.* $RT \in PP \to (\mathcal{R} \to \mathcal{S})$ *if there exists* $rt'$ *s.t.* $i \vdash^{\tau} RT_i \Rightarrow rt'$ *and* $rt' \sqsubseteq RT_{i'}$
- *An execution step* $\langle i,\rho,h\rangle \rightsquigarrow_{m,\tau} (r,h')$ *is typable w.r.t.* $RT \in PP \to (\mathcal{R} \to \mathcal{S})$ *if* $i \vdash^{\tau} RT_i \Rightarrow$
- *An execution sequence* $s_0 \rightsquigarrow_{m,\tau_0} s_1 \rightsquigarrow_{m,\tau_1} \ldots s_k \rightsquigarrow_{m,\tau_k} (r,h')$ *is typable w.r.t.* $RT \in PP \to (\mathcal{R} \to \mathcal{S})$ *if :*
  - $\forall i, 0 \leq i < k, s_i \rightsquigarrow_{m,\tau_i} s_{i+1}$ *is typable w.r.t. RT;*
  - $s_n \rightsquigarrow_{m,\tau_n} (r,h')$ *is typable w.r.t. RT.*

**Lemma 4.2.25** (High Security Environment High Result)**.** *Let* $\langle i,\rho,h\rangle, \langle i',\rho',h'\rangle \in State_G$*,* $se(i)$ *is high,* $\langle i,\rho,h\rangle \sim_{k_{\text{obs}},rt,rt',\beta} \langle i',\rho',h'\rangle$ $i \mapsto$ *and* $\langle i,\rho,h\rangle \rightsquigarrow (r,h_r)$*, then* $\mathbf{highResult}(r,h_r)$ *and* $h_r \sim_{k_{\text{obs}},\beta} h'$*.*

*Proof.* We do a structural induction on the instruction in $i$. This lemma is only applicable if the instruction at $i$ is either a return instruction or a possibly throwing instruction with uncaught exception.

- Return: the transfer rule has a constraint that $\vec{k_r}(\text{Norm})$ will be at least as high as *se* which is high. So by definition, we have $\mathbf{highResult}(r,h_r)$. Since return does not modify heaps, we know that $h_r \sim_{k_{\text{obs}},\beta} h'$.
- Invoke: the transfer rule where the instruction is throwing an uncaught exception $e$ has constraint saying that $\vec{k_r}(e)$ will be at least as high as *se*, the level of exception thrown by the method invoked, and the object level on which the method is invoked. We know *se* is high, so by definition, we have $\mathbf{highResult}(r,h_r)$. Heap wise, we know that the exception is newly generated so we can use Lemma 4.2.8 to say that $h_r \sim_{k_{\text{obs}},\beta} h'$.

- Iget: the transfer rule where the instruction is throwing an uncaught exception np has constraint saying that $\vec{k_r}(\text{np})$ will be at least as high as *se* and the security level of the object. We know *se* is high, so by definition, we have $\mathbf{highResult}(r, h_r)$. Heap wise, we know that the exception is newly generated so we can use Lemma 4.2.8 to say that $h_r \sim_{k_{\text{obs}},\beta} h'$.
- Iput: Same as Iget.
- Aget: the transfer rule where the instruction is throwing an uncaught exception np has constraint saying that $\vec{k_r}(\text{np})$ will be at least as high as *se* and the security level of the array. We know *se* is high, so by definition, we have $\mathbf{highResult}(r, h_r)$. Heap wise, we know that the exception is newly generated so we can use Lemma 4.2.8 to say that $h_r \sim_{k_{\text{obs}},\beta} h'$.
- Aput: Same as Aget.
- Arraylength: Same as Aget.
- Throw: the transfer rule has a constraint that for any uncaught exception $e$, $\vec{k_r}(e)$ will be at least as high as *se* which is high. So by definition, we have $\mathbf{highResult}(r, h_r)$. Throw does not modify heap as well, so we have $h_r \sim_{k_{\text{obs}},\beta} h'$.

All of the instructions that may be a return point have $\mathbf{highResult}(r, h_r)$ and $h_r \sim_{k_{\text{obs}},\beta} h'$, therefore the lemma holds. □

**Lemma 4.2.26** (High Region High Result)**.** *Let se be high in* $\mathbf{region}(s, \tau)$, $\mathbf{jun}(s, \tau)$ *is never defined,* $\langle i_0, \rho_0, h_0\rangle, \langle i', \rho', h'\rangle \in State_G$, $\langle i_0, \rho_0, h_0\rangle \sim_{k_{\text{obs}},rt_0,rt',\beta} \langle i', \rho', h'\rangle$ *and there is an execution trace*

$$\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i_k, \rho_k, h_k\rangle \rightsquigarrow_{m,\tau_k} (r, h_r)$$

*where* $\langle i_0, \rho_0, h_0\rangle \in \mathbf{region}(s, \tau)$*. Then* $\mathbf{highResult}(r, h_r)$ *and* $h_r \sim_{k_{\text{obs}},\beta} h'$*.*

*Proof.* We do induction on the length of the execution. For the base case where $k$ is 0, we can use Lemma 4.2.25. In the induction step, we know that $\langle i_k, \rho_k, h_k\rangle$ is in $\mathbf{region}(s, \tau)$ using SOAP2 and eliminating the case where it is a junction point by the assumption that $\mathbf{jun}(s, \tau)$ is never defined. Since we now have shorter execution length, we can apply induction hypothesis. Heap wise, we know from the transfer rules that field / array update will be bounded by *se*. Thus we have $h_r \sim_{k_{\text{obs}},\beta} h'$ by Lemma 4.2.5 and Lemma 4.2.6. □

**Lemma 4.2.27** (High Register Stays)**.** *Let* $\langle i_0, \rho_0, h_0\rangle \in State_G$ *and there is an execution step such that*

$$\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i_k, \rho_k, h_k\rangle \rightsquigarrow_{m,\tau_k} (r, h_r)$$

*where* $\langle i_0, \rho_0, h_0\rangle \in \mathbf{region}(s, \tau)$*, and se is high in* $\mathbf{region}(s, \tau)$*, if* $RT_0(r)$ *is high then* $RT_k(r)$ *is high.*

*Proof.* We do induction on the length of execution, and we do case analysis on possible instructions. If the length of execution is 0 we have $\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \langle i_k, \rho_k, h_k\rangle$. The

instruction is not a return point since it contradicts the assumption. If the instruction is an instruction that modifies $r$, we know that these instructions will update the register $r$ with security level at least as high as $se$, so the base case holds. In the induction step, we show using the same argument as the base case that $RT_1(r)$ is high, therefore now we can invoke induction hypothesis on the trace $\langle i_1, \rho_1, h_1\rangle \rightsquigarrow_{m,\tau_1} \ldots \langle i_k, \rho_k, h_k\rangle$ □

**Lemma 4.2.28** (Changed Register High)**.** *Let $\langle i_0, \rho_0, h_0\rangle \in State_G$ and there is an execution step such that*

$$\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i_k, \rho_k, h_k\rangle \rightsquigarrow_{m,\tau_k} (r, h_r)$$

*where $\langle i_0, \rho_0, h_0\rangle \in \mathbf{region}(s, \tau)$, and se is high in $\mathbf{region}(s, \tau)$ , $k = \mathbf{jun}(s, \tau)$, and the value of r is changed by one or more instruction in the execution trace, then $RT_k(r)$ is high.*

*Proof.* We do case analysis on where the first change might happen [1] then we do case analysis on all of the register-modifying instructions change register $r$ to high [2] and invoke Lemma 4.2.27 to claim that they stay high until it reaches $\langle i_k, \rho_k, h_k\rangle$. All these instructions which modify register $r$ will update the register $r$ with security level at least as high as $se$, so we already have [2]. Since we assume that there is a change, [1] trivially holds. □

**Lemma 4.2.29** (Unchanged Register Stays)**.** *Let $\langle i_0, \rho_0, h_0\rangle \in State_G$ and there is an execution step such that*

$$\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i_k, \rho_k, h_k\rangle$$

*and the value of r is not changed during the execution trace, then $RT_0(r) = RT_k(r)$ and $\rho_0(r) = \rho_k(r)$.*

*Proof.* We do induction on the length of execution, and we do case analysis on possible instructions. If the length of execution is 0, we have $\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \langle i_k, \rho_k, h_k\rangle$. The instruction is not a return point, since it contradicts the assumption. If the instruction is an instruction that modifies $r$, we know that it contradicts our assumption. If the instruction does not modify $r$, then the base case holds by definition. In the induction step, we show using the same argument as the base case that $RT_0(r) = RT_1(r)$ and $\rho_0(r) = \rho_1(r)$, therefore now we can invoke induction hypothesis on the trace $\langle i_1, \rho_1, h_1\rangle \rightsquigarrow_{m,\tau_1} \ldots \langle i_k, \rho_k, h_k\rangle$. □

**Lemma 4.2.30** (Junction Point Indistinguishable)**.** *Let $\beta$ a partial function $\beta \in L \rightharpoonup L$ and $\langle i_0, \rho_0, h_0\rangle, \langle i_0', \rho_0', h_0'\rangle \in \mathrm{State}_G$ two $DEX_G$ states such that*

$$\langle i_0, \rho_0, h_0\rangle \sim_{k_{\mathrm{obs}}, RT_{i_0}, RT_{i_0'}, \beta} \langle i_0', \rho_0', h_0'\rangle$$

*and $i_0 = i_0'$.*

*Suppose that se is high in region $\mathbf{region_m}(i_0, \tau_0)$ and also in region $\mathbf{region_m}(i_0', \tau_0')$. Suppose we have a derivation*

$$\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i_k, \rho_k, h_k\rangle \rightsquigarrow_{m,\tau_k} (r, h)$$

*and suppose this derivation is typable w.r.t. RT. Suppose we have a derivation*

$$\langle i'_0, \rho'_0, h'_0\rangle \rightsquigarrow_{m,\tau'_0} \ldots \langle i'_k, \rho'_k, h'_k\rangle \rightsquigarrow_{m,\tau'_k} (r', h')$$

*and suppose this derivation is typable w.r.t. RT. Then one of the following case holds:*

1. *there exists $j, j'$ with $0 \leq j \leq k$ and $0 \leq j' \leq k'$ s.t.*

$$i_j = i'_j \quad \text{and} \quad \langle i_j, \rho_j, h_j\rangle \sim_{k_{\text{obs}}, RT_{i_j}, RT_{i'_{j'}}, \beta} \langle i'_{j'}, \rho'_{j'}, h'_{j'}\rangle$$

2. $(r, h) \sim_{k_{\text{obs}}, \vec{k}_r, \beta} (r', h')$

*Proof.* We do a case analysis on whether a junction point is defined for both of the execution traces. There are three possible cases :

1. junction point is defined for both of the execution. We trace any changed registers during the execution. If the register is changed, then we can invoke Lemma 4.2.28 to claim that the register is high and we know that high register does not affect indistinguishability. If the register is not changed, then we can invoke Lemma 4.2.29 to obtain $RT_{i_j}(r) = RT_0(r)$ and $\rho_{i_j}(r) = \rho_0(r)$ and $\rho_{i'_{j'}}(r) = \rho'_0(r)$. If $RT_0(r)$ is low, then we know that $\rho_0(r) = \rho'_0(r)$, thus we can obtain that $\rho_{i_j}(r) = \rho_{i'_{j'}}(r)$. Otherwise, we know that $RT_{i_j}(r) = RT_{i'_{j'}}(r)$ is high. Whatever the case it does not affect indistinguishability. Since for all register in $RT_{i_j}$ ($RT_{i'_{j'}}$), they are either changed or unchanged, we can obtain $\langle i_j, \rho_j, h_j\rangle \sim_{k_{\text{obs}}, RT_{i_j}, RT_{i'_{j'}}, \beta} \langle i'_{j'}, \rho'_{j'}, h'_{j'}\rangle$, and we are in Case 1.

2. only one execution has junction point. For the part where junction point is not defined (assume it is the execution ending in $(r, h)$), we can invoke lemma 4.2.26 to obtain $\mathbf{highResult}(r, h)$. On the other execution path, we know from SOAP5 that the junction point is in the region ($\mathbf{jun_m}(i'_0, \tau'_0) \in \mathbf{region}(i_0, \tau_0)$). Hence we can invoke lemma 4.2.26 again to obtain $\mathbf{highResult}(r', h')$ since *se* is high in $\mathbf{region}(i_0, \tau_0)$, and prove that we are in Case 2.

3. both of the execution traces have no junction point. In this case, since we know that *se* is high in the region, we can just invoke lemma 4.2.26 on both executions to obtain $\mathbf{highResult}(r, h)$ and $\mathbf{highResult}(r', h')$, hence we are in Case 2.

All possible case leads to one of the cases. Therefore the lemma holds. □

**Lemma 4.2.31** (High Branching)**.** *Let all methods $m'$ in $P$ be non-interferent w.r.t. all the policies in $\mathbf{Policies_\Gamma}(m')$. Let $m$ be a method in $P$, $\beta \in L \rightharpoonup L$ a partial function, $s_1, s_2 \in \text{State}_G$ and two registers type $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ s.t.*

$$s_1 \sim_{k_{\text{obs}}, rt_1, rt_2, \beta} s_2$$

1. *If there exist two states* $\langle i_1', \rho_1', h_1' \rangle, \langle i_2', \rho_2', h_2' \rangle \in \text{State}_G$ *and two register types* $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ *s.t.* $i_1' \neq i_2'$ *and*

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} \langle i_1', \rho_1', h_1' \rangle & s_2 \rightsquigarrow_{m,\tau_2} \langle i_2', \rho_2', h_2' \rangle \\ i \vdash^{\tau_1} rt_1 \Rightarrow rt_1' & i \vdash^{\tau_2} rt_2 \Rightarrow rt_2' \end{array}$$

*then*

$$\begin{array}{c} se \text{ is high in } \mathbf{region}(i, \tau_1) \\ se \text{ is high in } \mathbf{region}(i, \tau_2) \end{array}$$

2. *If there exists a state* $\langle i_1', \rho_1', h_1' \rangle \in \text{State}_G$*, a final result* $(v_2, h_2') \in (V + L) \times \text{Heap}$ *and a register types* $rt_1' \in (\mathcal{R} \to \mathcal{S})$ *s.t.*

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} \langle i_1', \rho_1', h_1' \rangle & s_2 \rightsquigarrow_{m,\tau_2} (r_2, h_2') \\ i \vdash^{\tau_1} rt_1 \Rightarrow rt_1' & i \vdash^{\tau_2} rt_2 \Rightarrow \end{array}$$

*then*

$$se \text{ is high in } \mathbf{region}(i, \tau_1)$$

*Proof.* By case analysis on the instruction executed.

- **ifeq** and **ifneq**: the proof's outline follows from before as there is no possibility for an exception here.
- **invoke**: there are several cases to consider for this instruction to be a branching instruction:

  **1)** both are executing normally. Since the method we are invoking is non-interferent, and we have that $\rho_1 \sim_{k_{\text{obs}}, rt_1, rt_2, \beta} \rho_2$, we will also have indistinguishable results. Since they throw no exceptions there is no branching there.

  **2)** one of them is normal, the other throws an exception $e'$. Assume that the policy for the method called is $\vec{k_a'} \xrightarrow{k_h'} \vec{k_r'}$. This will imply that $\vec{k_r'}(e') \not\leq k_{\text{obs}}$ otherwise the output will be distinguishable. By the transfer rules we have that for all the regions *se* is to be at least as high as $\vec{k_r'}(e')$ (normal execution one is lub-ed with $k_e = \bigsqcup \{ \vec{k_r'}(e) \mid e \in \mathbf{excAnalysis}(m') \}$ where $e' \in \mathbf{excAnalysis}(m')$, and $\vec{k_r'}(e')$ by itself for the exception throwing one), thus we will have $se \not\leq k_{\text{obs}}$ throughout the regions. For the exception throwing one, if the exception is caught, then we know we will be in the first case. If the exception is uncaught, then we are in the second case.

  **3)** the method throws different exceptions (let's say $e_1$ and $e_2$). Assume that the policy for the method called is $\vec{k_a'} \xrightarrow{k_h'} \vec{k_r'}$. Again, since the outputs are indistinguishable, this means that $\vec{k_r'}(e_1) \not\leq k_{\text{obs}}$ and $\vec{k_r'}(e_2) \not\leq k_{\text{obs}}$. By the transfer rules, as before we will have *se* high in all the regions required. If the exceptions are both uncaught, then this lemma does not apply. Assume that $e_1$ is caught. We follow the argument from before that we will have *se* is high in $\mathbf{region}(i, \tau_1)$. If $e_2$ is caught, using the same argument we will have *se* is high in $\mathbf{region}(i, \tau_2)$

and we are in the first case. If $e_2$ is uncaught, then we know that we are in the second case. The rest of the cases will be dealt with by first assuming that $e_2$ is caught.

- object / array manipulation instructions that may throw a null pointer exception. This can only be a problem if one is null and the other is non-null. From this, we can infer that register pointing to object / array reference will have high security level (otherwise they have to have the same value). If this is the case, then from the transfer rules for handling null pointer we have that *se* is high in region $\mathbf{region}(i_1, \mathrm{np})$.

  Now, regarding the part which is not null, we also can deduce that it is from the transfer rules that we have *se* will be high in that region, which implies that *se* will be high in $\mathbf{region}(i_2, \mathrm{Norm})$.

- **throw**: Actually the reasoning for this instruction is closely similar to the case of object / array manipulation instruction that may throw a null pointer exception, except with additional possibility of the instruction throwing a different exception. Fortunately for us, this can only be the case if the security level to the register pointing to the object to throw is high. Therefore, the previous reasoning follows.

□

**Lemma 4.2.32** (Locally Respect (Specialized))**.** *Let all methods $m'$ in $P$ be non-interferent w.r.t. all the policies in $\mathbf{Policies}_\Gamma(m')$. Let $m$ be a method in $P$, $\beta \in L \rightharpoonup L$ a partial function, $s_1, s_2 \in \mathrm{State}_G$ two $DEX_G$ states at the same program point $i$ and two registers types $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ s.t. $s_1 \sim_{k_{\mathrm{obs}}, rt_1, rt_2, \beta} s_2$.*

1. *If there exists two states $s_1', s_2' \in \mathrm{State}_G$ and the program point of $s_1'$ is the same as $s_2'$ and two register types $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ s.t.*

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} s_1' & s_2 \rightsquigarrow_{m,\tau_2} s_2' \\ i \vdash^{\tau_1} rt_1 \Rightarrow rt_1' & i \vdash^{\tau_2} rt_2 \Rightarrow rt_2' \end{array}$$

   *then there exists $\beta' \in L \rightharpoonup L$ s.t.*

$$s_1' \sim_{k_{\mathrm{obs}}, rt_1', rt_2', \beta'} s_2' \quad \beta \subseteq \beta'$$

2. *If there exists a state $\langle i_1', \rho_1', h_1' \rangle \in \mathrm{State}_G$, a final result $(r_2, h_2') \in (V + L) \times \mathrm{Heap}$ and a registers types $rt_1' \in (\mathcal{R} \to \mathcal{S})$ s.t.*

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} \langle i_1', \rho_1', h_1' \rangle & s_2 \rightsquigarrow_{m,\tau_2} (r_2, h_2') \\ i \vdash^{\tau_1} rt_1 \Rightarrow rt_1' & i \vdash^{\tau_2} rt_2 \Rightarrow \end{array}$$

   *then there exists $\beta' \in L \rightharpoonup L$ s.t.*

$$h_1' \sim_{k_{\mathrm{obs}}, \beta'} h_2', \quad \mathbf{highResult}_{\mathbf{k_r}}(r_2, h_2') \quad \beta \subseteq \beta'$$

3. *If there exists two final results* $(r_1, h_1'), (r_2, h_2') \in (V + L) \times \text{Heap}$ *s.t.*

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} (r_1, h_1') & s_2 \rightsquigarrow_{m,\tau_2} (r_2, h_2') \\ i \vdash^{\tau_1} rt_1 \Rightarrow & i \vdash^{\tau_2} rt_2 \Rightarrow \end{array}$$

*then there exists* $\beta' \in L \rightharpoonup L$ *s.t.*

$$(r_1, h_1') \sim_{k_{\text{obs}}, \vec{k_r}, \beta'} (r_2, h_2') \quad \beta \subseteq \beta'$$

*Proof.* Since we have already proved this lemma for all the instructions apart from the exception cases, we only deal with the exception here. Moreover, we already proved for the heap that instructions without exception are indistinguishable. Therefore, only instructions which may cause an exception are considered here, and only consider the case where the registers may actually be distinguishable. Note that for exception case, the lemma is specialized only to affect those that have the same successor's program point.

- **invoke**: There are six possible successors here, but we only consider the four exception related ones (since one of them can be subsumed by the other) :

  **1)** One normal and one has a caught exception. In this case, we know that the lemma is not applicable since the successors have different program points.

  **2)** One normal and one has an uncaught exception (the case where one throws a caught exception and one throws an uncaught exception is proved using similar arguments). In this case, we have one successor state while the other will return a value or a location (case 2). So, in this case, we only need to prove that $\textbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$ (the part about heap indistinguishability is already proved). We can easily again appeal to output distinguishability since we already assumed that the method $m'$ is non-interferent. Since we have the exception $e$ returned by the method $m'$ as high ($\vec{k_r'}(e)$), we can now use the transfer rule which states that $\vec{k_r'}(e) \leq \vec{k_r}(e)$ and establish that $\vec{k_r}(e) \not\leq k_{\text{obs}}$ which in turn implies that $\textbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$, thus we are in Case 2.

  **3)** Both have caught exceptions. If they have the same exception, then we that the content of *ex* register will be the same thus the register will be indistinguishable. If they have different exception yet it is handled by the same program point, since we know that the method is non-interferent, we can use output indistinguishability which implies that the value of *ex* registers will be indistinguishable ($\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', ex, \beta} \rho_2'$). Since we have heap indistinguishability and registers indistinguishability, we have $s_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta} s_2'$ (Case 1).

  **4)** Both have uncaught exceptions (and different exceptions on top of that). Let's say the two exceptions are $e_1$ and $e_2$. For the beginning, we use the output indistinguishability of the method to establish that $\vec{k_r'}(e_1) \not\leq k_{\text{obs}}$ and $\vec{k_r'}(e_2) \not\leq k_{\text{obs}}$. Then, using the transfer rules for uncaught exceptions which states $\vec{k_r'}(e_1) \leq \vec{k_r}(e_1)$ and $\vec{k_r'} \leq \vec{k_r}(e_2)$ to establish that $\vec{k_r}(e_1)$ and $\vec{k_r}(e_2)$ are high

as well. Now, since they are both high, we can claim that they are indistinguishable (output-wise), therefore concluding the proof that we are in Case 3.

- **iget**: There are four cases to consider here:

  **1)** One is a normal execution and one has a caught null exception. In this case, we know that the lemma is not applicable since the successors have different program points.

  **2)** One is a normal execution and one has an uncaught null exception. The only difference with the previous case is that there will be one execution returning a location for exception instead. In this case, we only need to prove that this return of value is high ($\mathbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$). We know that $r_o$ (the register containing the object) is high ($rt(r_o) \not\leq k_{\text{obs}}$), otherwise $s_1 \not\sim_{k_{\text{obs}}, rt_1, rt_2 \beta} s_2$. The transfer rule for **iget** with uncaught exception states that $rt(r_o) \leq \vec{k_r}(\text{np})$, which will give us $\vec{k_r}(\text{np}) \not\leq k_{\text{obs}}$, which will imply that $\mathbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$, thus we are in Case 2.

  **3)** Both have caught null exceptions. In this case, there are two things that need consideration: the new objects in the heap, and the pseudo-register $ex$ containing the new null exception. Since we have $h_1 \sim_{k_{\text{obs}}, \beta} h_2$ and the exception is created fresh ($l_1 = \mathbf{fresh}(h_1)$, $l_1 \mapsto \mathbf{default}_{\text{np}}$, $l_2 = \mathbf{fresh}(h_2)$, $l_2 \mapsto \mathbf{default}_{\text{np}}$), by Lemma 4.2.8 we have that $h_1' \sim_{k_{\text{obs}}, \beta} h_2'$ as well. Now for the pseudo-register $ex$, we take the mapping $\beta'$ to be $\beta \oplus \{l_1 \mapsto l_2\}$, where $l_1 = \mathbf{fresh}(h_1)$ and $l_2 = \mathbf{fresh}(h_2)$, both are used to store the new exception. Under this mapping, we know that $l_1 \sim_{k_{\text{obs}}, \beta'} l_2$ and this will give us $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta'} \rho_2'$ since $\rho_1' = \{ex \mapsto l_1\}$ and $\rho_2' = \{ex \mapsto l_2\}$. Thus we are in Case 1.

  **4)** Both have uncaught null exceptions. Following the previous arguments, we have $h_1' \sim_{k_{\text{obs}}, \beta'} h_2'$, and $l_1 \sim_{k_{\text{obs}}, \beta'} l_2$, which will give us $(\langle l_1 \rangle, h_1') \sim_{k_{\text{obs}}, \vec{k_r}, \beta'} (\langle l_2 \rangle, h_2')$, thus we are in Case 3.

- **iput**, **aget**, and **aput**: The arguments closely follow that of **iget**

- **throw**: There are four cases to consider here :

  **1)** Two identical exceptions. In this case, we know that the exception will be the same. Therefore the value for $ex$ will be the same ($ex = \rho_1(r_e) = \rho_2(r_e)$), thus giving us $\rho_1' \sim_{k_{\text{obs}}, rt_1', rt_2', \beta'} \rho_2'$ if the exception is caught (Case 1). In the case where the exception is uncaught, we know that the value will be the same, that is $\rho_1(r_e)$. Therefore the output will be indistinguishable as well (Case 3).

  **2)** Two different exceptions, both are caught. In this case, we know that the lemma is not applicable since the successors have different handlers (thus program points).

  **3)** Two different exceptions, both are uncaught. The transfer rules state that $rt_1'(r_e) \leq \vec{k_r}(e_1)$ and $rt_2'(r_e) \leq \vec{k_r}(e_2)$ (where $r_e$ is the register containing the exception). Since $r_e$ must be high for the exceptions to be different values, we can

infer that $\vec{k_r}(e_1)$ and $\vec{k_r}(e_2)$ must be high as well. With this, we will have that $(r_1, h_1') \sim_{k_{\text{obs}}, \vec{k_r}, \beta'} (r_2, h_2')$ since both are high outputs (Case 3).

**4)** Two different exceptions, one is caught one is uncaught. Similar to the previous argument: we know that $\vec{k_r}(e)$ will be high. Therefore we will have $\mathbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$, $h_1' \sim_{k_{\text{obs}}, \beta'} h_2'$ (throw instruction does not modify the heap), and $\beta \subseteq \beta'$ (Case 2).

Including the previously proven locally respect lemma (Lemma 4.2.11, Lemma 4.2.14, Lemma 4.2.20), all of the possible scenarios will land on one of the cases. Therefore the lemma holds. □

**Lemma 4.2.33** (Typable DEX Program is Non-Interferent)**.** *Suppose we have $\beta$ a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and $\langle i_0, \rho_0, h_0\rangle, \langle i_0', \rho_0', h_0'\rangle \in \text{State}_G$ two $DEX_G$ states s.t. $i_0 = i_0'$ and $\langle i_0, \rho_0, h_0\rangle \sim_{k_{\text{obs}}, RT_{i_0}, RT_{i_0'}, \beta} \langle i_0', \rho_0', h_0'\rangle$. Suppose we have a derivation*

$$\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i_k, \rho_k, h_k\rangle \rightsquigarrow m, \tau_k(r, h)$$

*and suppose this derivation is typable w.r.t. RT. Suppose we also have another derivation*

$$\langle i_0', \rho_0', h_0'\rangle \rightsquigarrow_{m,\tau_0'} \ldots \langle i_k', \rho_k', h_k'\rangle \rightsquigarrow m, \tau_k'(r', h')$$

*and suppose this derivation is typable w.r.t. RT. Then what we want to prove is that there exsts $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ s.t.*

$$(r, h) \sim_{k_{\text{obs}}, \vec{k_v}, \beta'} (r', h') \text{ and } \beta \subseteq \beta'$$

*Proof.* Following the proof of the side effect safety, we use induction on the length of method call chain. For the base case, there is no **invoke** instruction involved (method call chain with length 0). A note about this setting is that we can use lemmas which assume that all the methods are non-interferent since we are not going to call another method. To start the proof in the base case of induction on method call chain length, we use induction on the length of $k$ and $k'$. The base case is when $k = k' = 0$. In this case, we can use case 3 of Lemma 4.2.32. There are several possible cases of the induction step:

1. $k > 0$ and $k' = 0$: then we can use case 2 of Lemma 4.2.32 to get the existence of $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ s.t.

$$h_1 \sim_{k_{\text{obs}}, \beta'} h', \quad \mathbf{highResult}_{\mathbf{k_r}}(r', h') \quad \text{and} \quad \beta \subseteq \beta' \quad [1]$$

Using case 2 of Lemma 4.2.31 we get

$$se \text{ is high in } \mathbf{region}(i_0, \tau_1)$$

where $\tau_1$ is the tag s.t. $i_0 \mapsto^{\tau_1} i_1$. SOAP2 gives us that either $i_1 \in \mathbf{region}(i_0, \tau_1)$ or $i_1 = \mathbf{jun}(i_0, \tau_1)$ but the latter case is rendered impossible due to SOAP3. Applying Lemma 4.2.26 we get

$$\mathbf{highResult}_{\mathbf{k_r}}(r, h_1') \quad \text{and} \quad h_1 \sim_{k_{\text{obs}}, \beta} h_1' \quad [2]$$

Combining [1] and [2] we get

$$h_1 \sim_{k_{\text{obs}},\beta'} h_1', \quad h_1 \sim_{k_{\text{obs}},\beta'} h', (r,h) \sim_{k_{\text{obs}},\vec{k_r},\beta} (r',h')$$

to conclude.

2. $k = 0$ and $k' > 0$ is symmetric to the previous case.

3. $k > 0$ and $k' > 0$. If the next instruction is at the same program point ($i_1 = i_1'$), we can conclude using Case 1 of Lemma 4.2.32 and the induction hypothesis. Otherwise we will have register typing $rt_1$ and $rt_1'$ s.t.

$$i_0 \vdash^{\tau_0} RT_{i_0} \Rightarrow rt_1, \quad rt_1 \sqsubseteq RT_{i_1}$$
$$i_0' \vdash^{\tau_0'} RT_{i_0'} \Rightarrow rt_1', \quad rt_1' \sqsubseteq RT_{i_1'}$$

Then using Case 1 of Lemma 4.2.31 we have

$$se \text{ is high in } \mathbf{region}(i_0, \tau)$$
$$se \text{ is high in } \mathbf{region}(i_0', \tau')$$

where $\tau, \tau'$ are tags s.t. $i_0 \mapsto^{\tau} i_1$ and $i_0' \mapsto^{\tau'} i_1'$. Using Case 1 of Lemma 4.2.32 there exists $\beta', \beta \subseteq \beta'$ s.t. (with the help of Lemma 4.2.13)

$$\langle i_1, \rho_1, h_1 \rangle \sim_{k_{\text{obs}},RT_{i_1},RT_{i_1'},\beta'} \langle i_1', \rho_1', h_1' \rangle$$

Invoking Lemma 4.2.30 will give us two cases:

- There exists $j, j'$ with $1 \leq j \leq k$ and $1 \leq j' \leq k'$ s.t. $i_j = i_j'$, and $\langle i_j, \rho_j, h_j \rangle \sim_{k_{\text{obs}},RT_{i_j},RT_{i_j'},\beta} \langle i_j', \rho_j', h_j' \rangle$. We can then use the induction hypothesis on the rest of the executions to conclude.
- $(r,h) \sim_{k_{\text{obs}},\vec{k_v},\beta'} (r',h')$ and we can directly conclude.

After we established the base case, we can then continue to prove by induction on method call chain. In the case where an instruction calls another method, we will have a non-interferent method since they necessarily have shorter call chain length (induction hypothesis).

□

Proof of Theorem 4.1.1 is direct application of Lemma 4.2.33 and Lemma 4.2.24.

Chapter 5

# Formalization of $\text{DEX}_{\mathcal{I}}$ and $\text{DEX}_{\mathcal{O}}$

We mechanize the pen and paper proof so that we can provide a higher confidence in the type system soundness. We already have the pen and paper proofs that the DEX type system is indeed sound (Section 4.2). Currently, we managed to formalize the $\text{DEX}_{\mathcal{I}}$ and $\text{DEX}_{\mathcal{O}}$ with about 14500 lines of definitions and proofs. We would like to be able to formalize also $\text{DEX}_{\mathcal{C}}$ and $\text{DEX}_{\mathcal{G}}$, but the size of the formalization is too big to fit into the scope of a Ph.D.

The structure for our formalization has much resemblance with the formalization of JVM done in Barthe et al. [2007]. We started with the definition of DEX which is based upon their definition of JVM, except that we replace the definition of operand stack to a mapping from registers to values. We also remove the exception handling mechanism to remove clutter. The formalization in Coq is available at

`http://users.cecs.anu.edu.au/~hengunadi/TranslationProof.html`

Inspired by the structure of the original proof, we start by formalizing the semantics of DEX instructions, the operational semantics of $\text{DEX}_{\mathcal{I}}$ and $\text{DEX}_{\mathcal{O}}$ instructions, and the notion of CDR. We then formalize the transfer rules based on the transfer rules defined in Chapter 4. The formalization then continues by providing the definition for indistinguishability relations which serve as the basis for non-interference definition. Finally, we formalize the proof that $\text{DEX}_{\mathcal{I}}$ and $\text{DEX}_{\mathcal{O}}$ are indeed non-interferent.

As highlighted in Section 4.2, there is a substantial difference between the proof for JVM type system and ours. Even though the structure of the definitions and the proofs are similar, the difference in the definition of indistinguishability and the fact that we do not have lift operation to the register typing means we have to modify and redesign the formalization in many places (resulting in about 6000 lines out of 14500 lines differences).

We assume several lemmas throughout the development of this formalization. Firstly, we used the axiom of excluded middle (which is also assumed in Barthe et al.'s formalization), then throughout the proof, we also assumed a lemma regarding the domain of mapping for register typing. In particular, the domain of $rt$ never changes, i.e., the domain is always the same as valid registers. We need this assumption because we define that the function $rt$ is total w.r.t the registers. We also assume

that the registers within a method never duplicates. Although this trivially holds, we have to specify this explicitly as a property of a method.

The formalization of $DEX_{\mathcal{I}}$ and $DEX_{\mathcal{O}}$ have many benefits for this work. Obviously, they give us more confidence in the correctness of our theoretical work. With the introduction of this machine-checked proof, we have increased confidence in the pen and paper proof. The development of this formalization also helps us to notice that we can not claim the step consistent lemma which exists for JVM, thus prompted us to redesign the way we prove non-interference.

## 5.1 The Semantics of DVM

In this section we will describe the components of the DVM:

- infrastructure
  - registers mapping
  - class
  - method and bytecode method
  - heap
- instructions

### 5.1.1 Infrastructure

We just focus on several main parts of the infrastructure, namely the registers mapping, class definition, method definition and heap definition.

**Registers Mapping** The main difference between DVM and JVM is the registers, which is implemented as a mapping from a register to a value. It is implemented as a mapping module which has three main functions: **get** is a function which takes a mapping and a variable and return a value, **update** is a function which takes a mapping, a variable $x$, and a value $v$, and return a new mapping where the content of variable $x$ is $v$, such that:

1. if we update a variable $x$ in the mapping with value $v$, then the content of $x$ in the new mapping will be $v$ (**get_update_new**); and
2. if a variable $x$ is not modified, then the content of $x$ in the new mapping will be the constant (**get_update_old**).

These two properties will be useful when we prove soundness.

```
Module Type DEX_REGISTERS.
  Parameter t : Type.
  Parameter get : t-> DEX_Reg -> option DEX_value.
  Parameter update : t -> DEX_Reg -> DEX_value -> t.
  Parameter dom : t -> list DEX_Reg.
```

```
  Parameter get_update_new :
    forall l x v, get (update l x v) x = Some v.
  Parameter get_update_old :
    forall l x y v, x<>y -> get (update l x v) y = get l y.
End DEX_REGISTERS.
```

**Class** Each class is distinguished by its name. A class may have a direct superclass, and class names form a hierarchy based on the superclass relation. Each class is also distinguished by the list of interfaces it implemented, and list of fields and methods which it has. Since $\text{DEX}_{\mathcal{I}}$ and $\text{DEX}_{\mathcal{O}}$ do not use any of those features, we just use the class itself as the container of the program without having to worry about class hierarchy and interface implemented. But it is useful to retain this structure for a complete formalization of DEX.

```
  Module Type DEX_CLASS_TYPE.

    Parameter name : DEX_Class -> DEX_ClassName.
    (** direct superclass *)
    Parameter superClass : DEX_Class -> option DEX_ClassName.
    (** list of implemented interfaces *)
    Parameter superInterfaces : DEX_Class -> list DEX_InterfaceName.

    Parameter field : DEX_Class -> DEX_ShortFieldName ->
      option DEX_Field.

    Parameter definedFields : DEX_Class -> list DEX_Field.
    Parameter in_definedFields_field_some : forall c f,
      In f (definedFields c) ->
      field c (DEX_FIELDSIGNATURE.name (DEX_FIELD.signature f))
        = Some f.
    Parameter field_some_in_definedFields : forall c f sfn,
      field c sfn = Some f -> In f (definedFields c).

    Parameter method : DEX_Class -> DEX_ShortMethodSignature ->
      option DEX_Method.
    Parameter method_signature_prop : forall cl mid m,
      method cl mid = Some m -> mid = DEX_METHOD.signature m.

    Definition defined_Method (cl:DEX_Class) (m:DEX_Method) :=
      method cl (DEX_METHOD.signature m) = Some m.

    (* modifiers *)
    Parameter isFinal : DEX_Class -> bool.
    Parameter isPublic : DEX_Class -> bool.
    Parameter isAbstract : DEX_Class -> bool.
```

```
  End DEX_CLASS_TYPE.
```

**Method** The method container is separated from its body because it is possible for a method to be abstract, that is its body needs to be fleshed out in the method of the class that implements the abstract method. The body of the method itself specifies the first address, the relationship between program points, and a mapping from program points to instructions. We also require that there are no duplicate registers (which is true by definition).

```
Module Type DEX_METHOD_TYPE.

    Parameter signature : DEX_Method -> DEX_ShortMethodSignature.
    (** A method that is not abstract has an empty method body *)
    Parameter body : DEX_Method -> option DEX_BytecodeMethod.

    (* modifiers *)
    Parameter isFinal : DEX_Method -> bool.
    Parameter isStatic : DEX_Method -> bool.
    Parameter isNative : DEX_Method -> bool.
    Definition isAbstract (m : DEX_Method) : bool :=
      match body m with
        None => true
      | Some _ => false
    end.

    Parameter visibility : DEX_Method -> DEX_Visibility.

    Definition valid_reg (m:DEX_Method) (x:DEX_Reg) : Prop :=
      forall bm, body m = Some bm ->
         (Reg_toN x) <= (DEX_BYTECODEMETHOD.max_locals bm).

    (* DEX additional for locR *)
    Definition within_locR (m:DEX_Method) (x:DEX_Reg) : Prop :=
      forall bm, body m = Some bm -> In x (DEX_BYTECODEMETHOD.locR bm).
    End DEX_METHOD_TYPE.

Module Type DEX_BYTECODEMETHOD_TYPE.
    Parameter firstAddress : DEX_BytecodeMethod -> DEX_PC.
    Parameter nextAddress : DEX_BytecodeMethod -> DEX_PC ->
      option DEX_PC.
    Parameter instructionAt : DEX_BytecodeMethod -> DEX_PC ->
      option DEX_Instruction.

    (** max number of local variables *)
```

```
  Parameter max_locals : DEX_BytecodeMethod -> nat.
  (** max number of elements on the operand stack *)
  Parameter max_operand_stack_size : DEX_BytecodeMethod -> nat.
  (* DEX for type system *)
  Parameter locR : DEX_BytecodeMethod -> list DEX_Reg.
  Parameter regs : DEX_BytecodeMethod -> list DEX_Reg.
  Parameter noDup_regs : forall bm, NoDup (regs (bm)).

  Definition DefinedInstruction (bm:DEX_BytecodeMethod) (pc:DEX_PC)
    : Prop := exists i, instructionAt bm pc = Some i.

End DEX_BYTECODEMETHOD_TYPE.
```

**Heap** The implementation of heap in DEX is the same as the one in JVM except that we only implement the dynamic field for objects (we do not implement array and static field). Heap is implemented as a mapping from an address to the reference of the object. There are three main interfaces to the heap, namely **get**, **update**, and **new**. **Get** is a function which returns the reference of the object, **update** is a function which updates the reference of the object contained in the mentioned address, and **new** is a function to allocate a new object in the heap.

```
Module Type DEX_HEAP.
 Parameter t : Type.

 Inductive DEX_AdressingMode : Set :=
   | DEX_DynamicField : DEX_Location -> DEX_FieldSignature
       -> DEX_AdressingMode.

 Inductive DEX_LocationType : Type :=
   | DEX_LocationObject : DEX_ClassName -> DEX_LocationType.

 Parameter get : t -> DEX_AdressingMode -> option DEX_value.
 Parameter update : t -> DEX_AdressingMode -> DEX_value -> t.
 Parameter typeof : t -> DEX_Location -> option DEX_LocationType.
 Parameter new : t -> DEX_Program -> DEX_LocationType ->
   option (DEX_Location * t).

 Inductive Compat (h:t) : DEX_AdressingMode -> Prop :=
   | CompatObject : forall cn loc f,
       typeof h loc = Some (DEX_LocationObject cn) ->
       Compat h (DEX_DynamicField loc f).

 Parameter get_update_same : forall h am v, Compat h am ->
   get (update h am v) am = Some v.
 Parameter get_update_old : forall h am1 am2 v, am1<>am2 ->
```

```
    get (update h am1 v) am2 = get h am2.
  Parameter get_uncompat : forall h am, ~ Compat h am ->
    get h am = None.

  Parameter typeof_update_same : forall h loc am v,
    typeof (update h am v) loc = typeof h loc.

  Parameter new_fresh_location : forall (h:t) (p:DEX_Program)
     (lt:DEX_LocationType) (loc:DEX_Location) (h':t),
    new h p lt = Some (loc,h') ->
    typeof h loc = None.

  Parameter new_typeof : forall (h:t) (p:DEX_Program)
      (lt:DEX_LocationType) (loc:DEX_Location) (h':t),
    new h p lt = Some (loc,h') ->
    typeof h' loc = Some lt.

  Parameter new_typeof_old : forall (h:t) (p:DEX_Program)
      (lt:DEX_LocationType) (loc loc':DEX_Location) (h':t),
    new h p lt = Some (loc,h') ->
    loc <> loc' ->
    typeof h' loc' = typeof h loc'.

  Parameter new_defined_object_field : forall (h:t) (p:DEX_Program)
      (cn:DEX_ClassName) (fs:DEX_FieldSignature) (f:DEX_Field)
      (loc:DEX_Location) (h':t),
    new h p (DEX_LocationObject cn) = Some (loc,h') ->
    is_defined_field p cn fs f ->
    get h' (DEX_DynamicField loc fs) = Some (init_field_value f).

  Parameter new_undefined_object_field : forall (h:t) (p:DEX_Program)
      (cn:DEX_ClassName) (fs:DEX_FieldSignature) (loc:DEX_Location) (h':t),
    new h p (DEX_LocationObject cn) = Some (loc,h') ->
    ~ defined_field p cn fs ->
    get h' (DEX_DynamicField loc fs) = None.

 Parameter new_object_no_change :
    forall (h:t) (p:DEX_Program) (cn:DEX_ClassName) (loc:DEX_Location)
      (h':t) (am:DEX_AdressingMode),
    new h p (DEX_LocationObject cn) = Some (loc,h') ->
    (forall (fs:DEX_FieldSignature), am <>
      (DEX_DynamicField loc fs)) -> get h' am = get h am.

End DEX_HEAP.
```

### 5.1.2 Instructions

There are 19 different instructions following the instruction outlined in Chapter 4:

- `| DEX_Nop`

  is the equivalent of **Nop**

- `| DEX_Move (k:DEX_ValKind) (rt:DEX_Reg) (rs:DEX_Reg)`

  is the equivalent of **Move**$(rt, rs)$. It is a dependent type which takes as arguments the type of the value and two registers, target register and source register.

- ```
  | DEX_Return
  | DEX_VReturn (k:DEX_ValKind) (rt:DEX_Reg)
  ```

  are the equivalent of **return**$(r)$. In practice, there is a possibility that the method is not returning any value, hence the two separate instructions. In the case where the method is returning a value, the instruction takes two arguments which indicate the return value type and the register from which the value is returned.

- `| DEX_Const (k:DEX_ValKind) (rt:DEX_Reg) (v:Z)`

  is the equivalent of **Const**$(r, v)$. It takes as argument the type of constant value, the target register, and the value of the constant.

- `| DEX_Goto (o:DEX_OFFSET.t)`

  is the equivalent of **Goto**$(t)$. The DEX_OFFSET is a type that captures the jump offset between the current program point ant the target program point.

- ```
  | DEX_PackedSwitch (rt:DEX_Reg) (firstKey:Z)
      (size:nat) (l:list DEX_OFFSET.t)
  ```

  is not contained in the list of instructions, but is part of the actual DEX bytecode instructions. The instruction takes as arguments the register from which the value is compared, the value of the first key, the size of the jump table, and the list of offset corresponding to each key.

- ```
  | DEX_SparseSwitch (rt:DEX_Reg) (size:nat)
      (l:list (Z * DEX_OFFSET.t))
  ```

  is not contained in the list of instructions, but is part of the actual DEX bytecode instructions. The instruction takes as arguments the register from which the value is compared, the size of the jump table, and a list of pairs consisting a key and its corresponding jump offset.

- ```
  | DEX_Ifcmp (cmp:DEX_CompInt) (ra:DEX_Reg)
      (rb:DEX_Reg) (o:DEX_OFFSET.t)
  | DEX_Ifz (cmp:DEX_CompInt) (r:DEX_Reg) (o:DEX_OFFSET.t)
  ```

are the equivalent of **ifeq**$(r,t)$. The two are basically a conditional branch instruction where **Ifcmp** compares the value of the two registers, and **Ifz** compares the value of the register with 0.

- ```
  | DEX_Ineg (rt:DEX_Reg) (rs:DEX_Reg)
  | DEX_Inot (rt:DEX_Reg) (rs:DEX_Reg)
  | DEX_I2b (rt:DEX_Reg) (rs:DEX_Reg)
  | DEX_I2s (rt:DEX_Reg) (rs:DEX_Reg)
  ```

  are not contained in the list of instructions, but are part of the actual DEX bytecode instructions. They are unary instructions which do an operation on the source register and store the result in the target register.

- ```
  | DEX_Ibinop (op:DEX_BinopInt) (rt:DEX_Reg)
       (ra:DEX_Reg) (rb:DEX_Reg)
  |  DEX_IbinopConst (op:DEX_BinopInt) (rt:DEX_Reg)
       (r:DEX_Reg) (v:Z)
  ```

  are the equivalent of **binop**$(op,r,ra,rb)$. The difference between the two is that the second operand of **IbinopConst** is a constant value $v$. They both take as arguments the type of binary operation, the target register, and the register which contains the first operand.

- ```
  | DEX_Iget (t:DEX_type) (rt:DEX_Reg) (ro:DEX_Reg)
     (f:DEX_FieldSignature)
  ```

  is the equivalent of **iget**$(r,r_o,f)$. It takes as argument the type of the value, the target register, the register which contains the reference to the object, and the field.

- ```
  | DEX_Iput (t:DEX_type) (rs:DEX_Reg) (ro:DEX_Reg)
     (f:DEX_FieldSignature)
  ```

  is the equivalent of **iput**$(r_s,r_o,f)$. It takes as argument the type of the value, the source register, the register which contains the reference to the object, and the field.

- ```
  | DEX_New (rt:DEX_Reg) (c:DEX_ClassName)
  ```

  is the equivalent of **new**$(r,c)$. It takes as argument the target register and the class reference.

### 5.1.3 The Operational Semantic of $DEX_{\mathcal{I}}$ and $DEX_{\mathcal{O}}$ Instructions

As mentioned in Chapter 4, the state of the operational semantics is a tuple of program point, heap, and registers mapping. In the case where a method is returning a value, the return state is a pair of heap and a single value. In this section, we will walk through each of the formalized operational semantics and how it matched the definition that we have outlined in Chapter 4, or what it means if the instruction is not in the list. There is explicit mentioning of the register domain even though
  ```

in theory this is not crucial. The reason we put them here is to ease the burden of having to prove that the registers involved are in the domain even though we already assumed that the registers involved in a method are total, meaning that every register involved in an operation will always be in the domain.

```
| nop : forall h m pc pc' regs,

  instructionAt m pc = Some DEX_Nop ->
  next m pc = Some pc' ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs))
```

**Nop** is not mentioned in the list, and basically, it stands for no operation. Upon reaching the program point with **Nop**, the state just transitions into its successor (next program point) without making any changes to the register nor the heap.

```
| const : forall h m pc pc' regs regs' k rt v,

  instructionAt m pc = Some (DEX_Const k rt v) ->
  In rt (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  (-2^31 <= v < 2^31)%Z ->
  DEX_METHOD.valid_reg m rt ->
  regs' = DEX_Registers.update regs rt (Num (I (Int.const v))) ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs'))
```

**Const** updates the mapping register with a constant value $v$. It matched with the operational semantics rule of **Const**: $\frac{P_m[i] = \mathbf{const}(r, v)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\mathrm{Norm}} \langle i+1, \rho \oplus \{r \mapsto v\}, h\rangle}$. The update of $\rho \oplus \{r \mapsto v\}$ corresponds to the update of $regs'$, the next program points $pc'$ corresponds to $i+1$, and there are no changes to the heap.

```
| move_step_ok : forall h m pc pc' regs regs' k rt rs v,

  instructionAt m pc = Some (DEX_Move k rt rs) ->
  In rt (DEX_Registers.dom regs) ->
  In rs (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some v = DEX_Registers.get regs rs ->
  DEX_METHOD.valid_reg m rt ->
  DEX_METHOD.valid_reg m rs ->
  regs' = DEX_Registers.update regs rt v ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs'))
```

**Move** copy a value stored in the source register to the target register. This corresponds to the operational semantic rule $\frac{P_m[i] = \mathbf{move}(r, r_s) \;\; r_s \in \mathbf{dom}(\rho)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\mathrm{Norm}} \langle i+1, \rho \oplus \{r \mapsto \rho(r_s)\}, h\rangle}$ where the update of $\rho \oplus \{r \mapsto \rho(r_s)\}$ corresponds to the update of $regs'$ and $v$ corresponds to the value of $\rho(r_s)$. There are no changes to the heap, so the next step also has the same heap $h$.

```
| goto_step_ok : forall h m pc regs o,

  instructionAt m pc = Some (DEX_Goto o) ->
  DEX_NormalStep p m (pc, (h, regs))
    ((DEX_OFFSET.jump pc o), (h, regs))
```

Just like **Nop**, **Goto** just modifies a program point and does not change any registers nor the heap. It corresponds to $\frac{P_m[i] = \mathbf{goto}(t)}{\langle i, \rho, h\rangle \;\rightsquigarrow\; \langle t, \rho, h\rangle}$ where $t$ corresponds to the offset jump.

```
| packedswitch_step_ok1 : forall h m pc l v r firstKey size
    list_offset n o,

  instructionAt m pc =
    Some (DEX_PackedSwitch r firstKey size list_offset) ->
  Some (Num (I v)) = DEX_Registers.get l r ->
  (firstKey <= Int.toZ v < firstKey + (Z_of_nat size))%Z ->
  length list_offset = size ->
  Z_of_nat n = ((Int.toZ v) - firstKey)%Z ->
  nth_error list_offset n = Some o ->
  DEX_METHOD.valid_reg m r ->

  DEX_NormalStep p m (pc, (h, l)) ((DEX_OFFSET.jump pc o), (h, l))

| packedswitch_step_ok2 : forall h m pc pc' l v r firstKey size
    list_offset,

  instructionAt m pc =
    Some (DEX_PackedSwitch r firstKey size list_offset) ->
  Some (Num (I v)) = DEX_Registers.get l r ->
  length list_offset = size ->
  (Int.toZ v < firstKey \/ firstKey + (Z_of_nat size) <= Int.toZ v)%Z ->
  next m pc = Some pc' ->
  DEX_METHOD.valid_reg m r ->

  DEX_NormalStep p m (pc, (h, l)) (pc', (h, l))
```

In the case where the content of the register is within $(firstKey + size)$, then the program point will be transferred to the corresponding offset from the list. In the case where the value falls out of range, then the program point will be transferred to the next program point. There are no changes to the registers mapping nor the heap. The two rules correspond to these two scenarios.

```
| sparseswitch_step_ok1 : forall h m pc l v v' o r size listkey,

  instructionAt m pc = Some (DEX_SparseSwitch r size listkey) ->
  length listkey = size ->
  Some (Num (I v)) = DEX_Registers.get l r ->
  List.In (pair v' o) listkey ->
  v' = Int.toZ v ->
  DEX_METHOD.valid_reg m r ->

  DEX_NormalStep p m (pc, (h, l)) ((DEX_OFFSET.jump pc o), (h, l))

| sparseswitch_step_ok2 : forall h m pc pc' l v r size listkey,

  instructionAt m pc = Some (DEX_SparseSwitch r size listkey) ->
  length listkey = size ->
  Some (Num (I v)) = DEX_Registers.get l r ->
  (forall v' o, List.In (pair v' o) listkey ->  v' <> Int.toZ v) ->
  next m pc = Some pc' ->
  DEX_METHOD.valid_reg m r ->

  DEX_NormalStep p m (pc, (h, l)) (pc', (h, l))
```

In the case where the content of the register is in one of the key offset pairs, then the program point will be transferred to the corresponding offset. If the value does not match any of the keys in the list, then the program point will be transferred to the next program point instead. Just like **Packedswitch**, there are no changes to the registers mapping nor the heap, and the two rules correspond to the two scenarios.

```
| ifcmp_step_jump : forall h m pc regs va vb cmp ra rb o,

  instructionAt m pc = Some (DEX_Ifcmp cmp ra rb o) ->
  In ra (DEX_Registers.dom regs) ->
  In rb (DEX_Registers.dom regs) ->
  Some (Num (I va)) = DEX_Registers.get regs ra ->
  Some (Num (I vb)) = DEX_Registers.get regs rb ->
  SemCompInt cmp (Int.toZ va) (Int.toZ vb) ->
  DEX_METHOD.valid_reg m ra ->
  DEX_METHOD.valid_reg m rb ->
```

```
  DEX_NormalStep p m (pc, (h, regs))
    ((DEX_OFFSET.jump pc o), (h, regs))

| ifcmp_step_continue : forall h m pc pc' regs va vb cmp ra rb o,

  instructionAt m pc = Some (DEX_Ifcmp cmp ra rb o) ->
  In ra (DEX_Registers.dom regs) ->
  In rb (DEX_Registers.dom regs) ->
  Some (Num (I va)) = DEX_Registers.get regs ra ->
  Some (Num (I vb)) = DEX_Registers.get regs rb ->
  ~SemCompInt cmp (Int.toZ va) (Int.toZ vb) ->
  next m pc = Some pc' ->
  DEX_METHOD.valid_reg m ra ->
  DEX_METHOD.valid_reg m rb ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs))

| ifz_step_jump : forall h m pc regs v cmp r o,

  instructionAt m pc = Some (DEX_Ifz cmp r o) ->
  In r (DEX_Registers.dom regs) ->
  Some (Num (I v)) = DEX_Registers.get regs r ->
  SemCompInt cmp (Int.toZ v) (0) ->
  DEX_METHOD.valid_reg m r ->

  DEX_NormalStep p m (pc, (h, regs))
    ((DEX_OFFSET.jump pc o), (h, regs))

| ifz_step_continue : forall h m pc pc' regs v cmp r o,

  instructionAt m pc = Some (DEX_Ifz cmp r o) ->
  In r (DEX_Registers.dom regs) ->
  Some (Num (I v)) = DEX_Registers.get regs r ->
  ~SemCompInt cmp (Int.toZ v) (0) ->
  next m pc = Some pc' ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs))
```

These rules correspond to the two operational semantics rules for **ifz**. If the test is true, then the program point will be transferred to the offset pointed by the instruction. Otherwise, the instruction will fall through to the next instruction. There are no modifications to the registers mapping and the heap. These rules correspond to the operational semantics rule $\frac{P[i]_m = \textbf{ifeq}(r, j) \quad \rho(r) = 0}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle t, \rho, h\rangle}$ where the test is true and

$\frac{P_m[i] = \mathbf{ifeq}(r,t) \quad \rho(r) \neq 0}{\langle i,\rho,h\rangle \rightsquigarrow_{m,\mathrm{Norm}} \langle i+1,\rho,h\rangle}$ where the test is not true.

```
| ineg_step : forall h m pc regs regs' pc' rt rs v,

  instructionAt m pc = Some (DEX_Ineg rt rs) ->
  In rt (DEX_Registers.dom regs) ->
  In rs (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some (Num (I v)) = DEX_Registers.get regs rs ->
  DEX_METHOD.valid_reg m rt ->
  DEX_METHOD.valid_reg m rs ->
  regs' = DEX_Registers.update regs rt (Num (I (Int.neg v))) ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs'))

| inot_step : forall h m pc regs regs' pc' rt rs v,

  instructionAt m pc = Some (DEX_Inot rt rs) ->
  In rt (DEX_Registers.dom regs) ->
  In rs (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some (Num (I v)) = DEX_Registers.get regs rs ->
  DEX_METHOD.valid_reg m rt ->
  DEX_METHOD.valid_reg m rs ->
  regs' = DEX_Registers.update regs rt (Num (I (Int.not v))) ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs'))

| i2b_step_ok : forall h m pc pc' regs regs' rt rs v,

  instructionAt m pc = Some (DEX_I2b rt rs) ->
  In rt (DEX_Registers.dom regs) ->
  In rs (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some (Num (I v)) = DEX_Registers.get regs rs ->
  DEX_METHOD.valid_reg m rt ->
  DEX_METHOD.valid_reg m rs ->
  regs' = DEX_Registers.update regs rt (Num (I (b2i (i2b v)))) ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs'))

| i2s_step_ok : forall h m pc pc' regs regs' rt rs v,

  instructionAt m pc = Some (DEX_I2s rt rs) ->
```

```
  In rt (DEX_Registers.dom regs) ->
  In rs (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some (Num (I v)) = DEX_Registers.get regs rs ->
  DEX_METHOD.valid_reg m rt ->
  DEX_METHOD.valid_reg m rs ->
  regs' = DEX_Registers.update regs rt (Num (I (s2i (i2s v)))) ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs'))
```

The four instructions here are unary operators, and they have similar operational semantics except that each instruction has a different effect. **i2s** convert integer to short, **i2b** convert integer to byte, **ineg** gives the two's complement of the value contained in the register and **inot** gives the one's complement of the value contained in the register. The program point will then be transferred into the next instruction.

```
| ibinop_step_ok : forall h m pc pc' regs regs' op rt ra rb va vb,

  instructionAt m pc = Some (DEX_Ibinop op rt ra rb) ->
  In rt (DEX_Registers.dom regs) ->
  In ra (DEX_Registers.dom regs) ->
  In rb (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some (Num (I va)) = DEX_Registers.get regs ra ->
  Some (Num (I vb)) = DEX_Registers.get regs rb ->
  DEX_METHOD.valid_reg m rt ->
  DEX_METHOD.valid_reg m ra ->
  DEX_METHOD.valid_reg m rb ->
  regs' = DEX_Registers.update regs rt
    (Num (I (SemBinopInt op va vb))) ->

  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs'))

| ibinopconst_step_ok : forall h m pc pc' regs regs' op rt r va v,

  instructionAt m pc = Some (DEX_IbinopConst op rt r v) ->
  In r (DEX_Registers.dom regs) ->
  In rt (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some (Num (I va)) = DEX_Registers.get regs r ->
  DEX_METHOD.valid_reg m rt ->
  DEX_METHOD.valid_reg m r ->
  regs' = DEX_Registers.update regs rt
    (Num (I (SemBinopInt op va (Int.const v)))) ->
```

```
  DEX_NormalStep p m (pc, (h, regs)) (pc', (h, regs'))
```

These two instructions have very similar operational semantics corresponding to **binop**: $\frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b) \quad r_a, r_b \in \mathbf{dom}(\rho) \quad n = \rho(r_a)\ \underline{op}\ \rho(r_b)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto n\}, h\rangle}$. The registers mapping will update the value with the result of applying the binary operation to the values. The program point will then be transferred to the next instruction.

```
| new : forall h m pc pc' regs regs' c rt loc h',

  instructionAt m pc = Some (DEX_New rt c) ->
  In rt (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  DEX_Heap.new h p (DEX_Heap.DEX_LocationObject c) = Some (pair loc h') ->
  regs' = DEX_Registers.update regs rt (Ref loc) ->

  DEX_NormalStep p m (pc,(h,regs)) (pc',(h',regs'))
```

The operational semantics of **new**: $\frac{P_m[i] = \mathbf{new}(r, c) \quad l = \mathbf{fresh}(h)}{\langle i, \rho, h\rangle \rightsquigarrow \langle i+1, \rho \oplus \{r \mapsto l\}, h \oplus \{l \mapsto \mathbf{default}(c)\}\rangle}$ matches this instruction. The heap is updated with a mapping to the new object, and the register mapping is updated with the corresponding fresh location. The program point is then transferred to the next instruction.

```
| iput : forall h m pc pc' regs f rs ro loc cn k v,

  instructionAt m pc = Some (DEX_Iput k rs ro f) ->
  In rs (DEX_Registers.dom regs) ->
  In ro (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some (Ref loc) = DEX_Registers.get regs ro ->
  Some v = DEX_Registers.get regs rs ->
  DEX_Heap.typeof h loc = Some (DEX_Heap.DEX_LocationObject cn) ->
  defined_field p cn f ->
  assign_compatible p h v (DEX_FIELDSIGNATURE.type (snd f)) ->

  DEX_NormalStep p m (pc,(h,regs))
    (pc',(DEX_Heap.update h (DEX_Heap.DEX_DynamicField loc f) v, regs))
```

The semantics $\frac{P_m[i] = \mathbf{iput}(r_s, r_o, f) \quad \rho(r_o) \in \mathbf{dom}(h) \quad f \in \mathbf{dom}(h(\rho(r_o)))}{\langle i, \rho, h\rangle \rightsquigarrow_{n,\text{Norm}} \langle i+1, \rho, os, h \oplus \{\rho(r_o) \mapsto h(\rho(r_o)) \oplus \{f \mapsto \rho(r_s)\}\}\rangle}$ matches this instruction. The field $f$ of the object in the heap which corresponds to the object referenced by $r_o$ is updated with the value that is contained in $r_s$. There are no updates to the registers mapping, and then the program will continue execution with the next instruction. Although in Figure 4.2 there are more semantics for **iput**, but only one is applicable in this case. This is because $\text{DEX}_{\mathcal{O}}$ does not have exception handling mechanism hence the other possible semantics do not apply.

```
| getfield : forall h m pc pc' regs regs' rt ro loc f k v cn,

  instructionAt m pc = Some (DEX_Iget k rt ro f) ->
  In rt (DEX_Registers.dom regs) ->
  In ro (DEX_Registers.dom regs) ->
  next m pc = Some pc' ->
  Some (Ref loc) = DEX_Registers.get regs ro ->
  DEX_Heap.typeof h loc = Some (DEX_Heap.DEX_LocationObject cn) ->
  defined_field p cn f ->
  DEX_Heap.get h (DEX_Heap.DEX_DynamicField loc f) = Some v ->
  regs' = DEX_Registers.update regs rt v ->

  DEX_NormalStep p m (pc,(h, regs)) (pc',(h, regs'))
```

The operational semantics $\frac{P_m[i] = \mathbf{iget}(r, r_o, f) \quad \rho(r_o) \in \mathbf{dom}(h) \quad f \in \mathbf{dom}(h(\rho(r_o)))}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto h(\rho(r_o)).f\}, h\rangle}$ matches with this instruction. The registers mapping will be updated with the value from the field $f$ of the object in the heap which is referred by $r_o$. There are no changes to the heap, and the execution will continue with the next execution. Like **iput**, there is only one semantics applicable since $DEX_{\mathcal{O}}$ does not have exception handling mechanism.

```
| void_return : forall h m pc regs,

  instructionAt m pc = Some DEX_Return ->
  DEX_METHODSIGNATURE.result (DEX_METHOD.signature m) = None ->

  DEX_ReturnStep p m (pc, (h, regs)) (h, Normal None)

| vreturn : forall h m pc regs val t k rs,
  instructionAt m pc = Some (DEX_VReturn k rs) ->
  In rs (DEX_Registers.dom regs) ->
  DEX_METHODSIGNATURE.result (DEX_METHOD.signature m) = Some t ->
  assign_compatible p h val t ->
  compat_ValKind_value k val ->
  Some val = DEX_Registers.get regs rs ->

  DEX_ReturnStep p m (pc, (h, regs)) (h, Normal (Some val))
```

In the case where the method is not returning a value, the method execution will stop there. When a method is method is returning a value, the content of the register will become the return value. It corresponds to the rule $\frac{P[i]_m = \mathbf{return}(r_s) \quad r_s \in \mathbf{dom}(\rho)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \rho(r_s), h}$.

### 5.1.4 Successor Relation and CDR

**Successor Relation** The successor relation for DEX is governed by simple rules: a **goto** will have target program point as its successor, a branching instruction will have two successors, i.e., its next instruction and the target branch, a return point will have no successor, and other instructions will have its next instruction as the successor. Except the **PackedSwitch** and **SparseSwitch**, everything else follows the rule (as can be seen by the formalization).

```
| DEX_goto : forall i (o:DEX_OFFSET.t),
  DEX_step i (DEX_Goto o) (Some (DEX_OFFSET.jump i o))
```

corresponds to the rule of **goto** which has its target program point as its successor,

```
| DEX_ifcmp : forall i j (cmp:DEX_CompInt) (ra:DEX_Reg) (rb:DEX_Reg)
    (o:DEX_OFFSET.t),
  next m i = Some j \/ j = DEX_OFFSET.jump i o ->
  DEX_step i (DEX_Ifcmp cmp ra rb o) (Some j)

| DEX_ifz : forall i j (cmp:DEX_CompInt) (r:DEX_Reg) (o:DEX_OFFSET.t),
  next m i = Some j \/ j = DEX_OFFSET.jump i o ->
  DEX_step i (DEX_Ifz cmp r o) (Some j)
```

corresponds to the rule where branching instructions will have two successors; its next instruction and its target program point,

```
| DEX_return_s : forall i,
  DEX_step i DEX_Return None

| DEX_vReturn : forall i (k:DEX_ValKind) (rt:DEX_Reg),
  DEX_step i (DEX_VReturn k rt) None
```

corresponds to the rule where return instructions does not have a successor,

```
| DEX_nop : forall i j,
  next m i = Some j ->
  DEX_step i DEX_Nop (Some j)

| DEX_move : forall i j (k:DEX_ValKind) (rt:DEX_Reg) (rs:DEX_Reg),
  next m i = Some j ->
  DEX_step i (DEX_Move k rt rs) (Some j)

| DEX_const : forall i j (k:DEX_ValKind) (rt:DEX_Reg) (v:Z),
  next m i = Some j ->
  DEX_step i (DEX_Const k rt v) (Some j)

| DEX_ineg : forall i j (rt:DEX_Reg) (rs:DEX_Reg),
```

```
  next m i = Some j ->
  DEX_step i (DEX_Ineg rt rs) (Some j)

| DEX_inot : forall i j (rt:DEX_Reg) (rs:DEX_Reg),
  next m i = Some j ->
  DEX_step i (DEX_Inot rt rs) (Some j)

| DEX_i2b : forall i j (rt:DEX_Reg) (rs:DEX_Reg),
  next m i = Some j ->
  DEX_step i (DEX_I2b rt rs) (Some j)

| DEX_i2s : forall i j (rt:DEX_Reg) (rs:DEX_Reg),
  next m i = Some j ->
  DEX_step i (DEX_I2s rt rs) (Some j)

| DEX_ibinop : forall i j (op:DEX_BinopInt) (rt:DEX_Reg) (ra:DEX_Reg)
    (rb:DEX_Reg),
  next m i = Some j ->
  DEX_step i (DEX_Ibinop op rt ra rb) (Some j)

| DEX_ibinopConst : forall i j (op:DEX_BinopInt) (rt:DEX_Reg)
    (r:DEX_Reg) (v:Z),
  next m i = Some j ->
  DEX_step i (DEX_IbinopConst op rt r v) (Some j)

| DEX_iput : forall i j k rs ro f,
  next m i = Some j ->
  DEX_step i (DEX_Iput k rs ro f) (Some j)

| DEX_iget : forall i j k r ro f,
  next m i = Some j ->
  DEX_step i (DEX_Iget k r ro f) (Some j)

| DEX_new : forall i j r c,
  next m i = Some j ->
  DEX_step i (DEX_New r c) (Some j)
```

corresponds to the rule for other instructions they have their immediate instruction as their successor,

```
| DEX_packedSwitch : forall i j (reg:DEX_Reg) (firstKey:Z) (size:nat)
    (l:list DEX_OFFSET.t),
  next m i = Some j ->
  DEX_step i (DEX_PackedSwitch reg firstKey size l) (Some j)
```

```
| DEX_packedSwitch_jump : forall i j (reg:DEX_Reg) (firstKey:Z)
    (size:nat) (l:list DEX_OFFSET.t),
  In j l ->
  DEX_step i (DEX_PackedSwitch reg firstKey size l)
    (Some (DEX_OFFSET.jump i j))

| DEX_sparseSwitch_default : forall i j (reg:DEX_Reg) (size:nat)
    (l:list (Z * DEX_OFFSET.t)),
  next m i = Some j ->
  DEX_step i (DEX_SparseSwitch reg size l) (Some j)

| DEX_sparseSwitch_jump : forall i (j:DEX_OFFSET.t) (reg:DEX_Reg)
    (size:nat) (l:list (Z * DEX_OFFSET.t)),
  In j (@map _ _ (@snd _ _) l) ->
  DEX_step i (DEX_SparseSwitch reg size l)
    (Some (DEX_OFFSET.jump i j))
```

except for **PackedSwitch** and **SparseSwitch** which have multiple successors. Apart from their immediate instruction which is always included as one of their successor, the target program points in contained in their list are also their successors.

**CDR**

```
Record CDR : Type := make_CDR {
     region : PC -> PC -> Prop;
     junc : PC -> PC -> Prop;
     junc_func: forall i j1 j2,
       junc i j1 -> junc i j2 -> j1=j2;
     soap1: forall i j k,
       step i (Some j) ->
       step i (Some k) ->
       j <> k ->
       region i k \/ junc i k;
     soap2:forall i j k,
       region i j->
       step j (Some k) ->
       region i k \/ junc i k;
     soap3 : forall i j k,
       region i j ->
       result j ->
       ~ junc i k
  }
```

*junc_func* specifies the uniqueness of a junction point. There are three SOAP properties here because we do not deal with exception cases. We can also ignore the exception tag $\tau$ from each of SOAP statement.

- SOAP1 states that $\forall i, j, k \in \mathcal{PP}$ and tag $\tau$ if $i \mapsto j$ and $i \mapsto^{\tau} k$ and $j \neq k$ ($i$ is hence a branching point) then $k \in \mathbf{region}(i, \tau)$ or $k = \mathbf{jun}(i, \tau)$. We can see that the statement translates component by component. "$i \mapsto j$" corresponds to "step i (Some j)", $i \mapsto k$ corresponds to "step i (Some k)", "$k \in \mathbf{region}(i, \tau)$" corresponds to "region i k", and "$k = \mathbf{jun}(i, \tau)$" corresponds to "junc i k".
- SOAP2 states that $\forall i, j, k \in \mathcal{PP}$ and tag $\tau$, if $j \in \mathbf{region}(i, \tau)$ and $j \mapsto k$, then either $k \in \mathbf{region}(i, \tau)$ or $k = \mathbf{jun}(i, \tau)$. This statement also translates directly. "$j \in \mathbf{region}(i, \tau)$" corresponds to "region i j", "$j \mapsto k$" corresponds to "step j (Some k)", "$k \in \mathbf{region}(i, \tau)$" corresponds to "region i k", and "$k = \mathbf{jun}(i, \tau)$" corresponds to "junc i k".
- SOAP3 states that $\forall i, j \in \mathcal{PP}$ and tag $\tau$, if $j \in \mathbf{region}(i, \tau)$ and $j$ is a return point then $\mathbf{jun}(i, \tau)$ is undefined. "$j \in \mathbf{region}(i, \tau)$" corresponds to "region i j", $j$ is a return point corresponds to "result j", and undefined "$\mathbf{jun}(i, \tau)$" corresponds to " junc i k" because whatever program point is $k$, it will never be the junction point of $i$.

## 5.2 Formalization of $DEX_{\mathcal{I}}$

Having defined the semantics for DVM, we are now going for the non-interference proof for $DEX_{\mathcal{I}}$. We separate the formalization for $DEX_{\mathcal{I}}$ and $DEX_{\mathcal{O}}$ because there are some complications when we introduce object definitions which complicate the definitions, e.g., the definition of indistinguishability for locations and the treatment of $\beta$ mapping. Firstly we will start with the definition of the transfer rules.

### 5.2.1 Transfer Rules

In this section, we show the correspondence between the transfer rules and its formalization in Coq.

```
| DEX_nop : forall i rt,
  texec i DEX_Nop rt (Some rt)

| DEX_goto : forall i (rt:TypeRegisters) (o:DEX_OFFSET.t),
  texec i (DEX_Goto o) rt (Some rt)
```

**Nop** and **Goto** do not have any particular constraints. They just have a rule that their successor has to have a registers typing which is at least as restricted as the instruction's registers typing.

```
| DEX_move : forall i (rt:TypeRegisters) k_rs (k:DEX_ValKind)
    (r:DEX_Reg) (rs:DEX_Reg),
  In r (MapList.dom rt) ->
  In rs (MapList.dom rt) ->
  MapList.get rt rs = Some k_rs ->
```

```
  texec i (DEX_Move k r rs) rt
    (Some (MapList.update rt r ((se i) U k_rs)))

| DEX_ineg : forall i ks (rt:TypeRegisters) (r:DEX_Reg) (rs:DEX_Reg),
  In r (MapList.dom rt) ->
  In rs (MapList.dom rt) ->
  MapList.get rt rs = Some ks ->
  texec i (DEX_Ineg r rs) rt
    (Some (MapList.update rt r (L.Simple ((se i) U ks))))

| DEX_inot : forall i ks (rt:TypeRegisters) (r:DEX_Reg) (rs:DEX_Reg),
  In r (MapList.dom rt) ->
  In rs (MapList.dom rt) ->
  MapList.get rt rs = Some ks ->
  texec i (DEX_Inot r rs) rt
    (Some (MapList.update rt r (L.Simple ((se i) U ks))))

| DEX_i2b : forall i ks (rt:TypeRegisters) (r:DEX_Reg) (rs:DEX_Reg),
  In r (MapList.dom rt) ->
  In rs (MapList.dom rt) ->
  MapList.get rt rs = Some ks ->
  texec i (DEX_I2b r rs) rt
    (Some (MapList.update rt r (L.Simple ((se i) U ks))))

| DEX_i2s : forall i ks (rt:TypeRegisters) (r:DEX_Reg) (rs:DEX_Reg),
  In r (MapList.dom rt) ->
  In rs (MapList.dom rt) ->
  MapList.get rt rs = Some ks ->
  texec i (DEX_I2s r rs) rt
    (Some (MapList.update rt r (L.Simple ((se i) U ks))))
```

**Move** also does not have any constraints, just the constraint that the next registers typing have to be at least as restricted as the current registers typing with register $r$ updated with the least upper bound of the security level of register $r_s$ and the current security environment. This corresponds to the "texec i (DEX_Move k r rs) rt (Some (MapList.update rt r ((se i) U k_rs)))". Similarly, unary operators also have the same rule where we need the subsequent registers typing to be at least as restricted as the current registers typing modulo the updated register, represented by the statement "texec i (DEX_I2b r rs) rt (Some (MapList.update rt r (L.Simple ((se i) U ks))))'"

```
| DEX_return_ : forall i (rt:TypeRegisters),
  sgn.(DEX_resType) = None ->
  texec i (DEX_Return) rt None

| DEX_vReturn : forall i (rt:TypeRegisters) k_r kv (k:DEX_ValKind)
```

```
    (r:DEX_Reg),
  In r (MapList.dom rt) ->
  MapList.get rt r = Some k_r ->
  sgn.(DEX_resType) = Some kv ->
  ((se i) U k_r) <= kv ->
  texec i (DEX_VReturn k r) rt None
```

If the return instruction does not return a value, then there is no constraint involved. When it is returning a value, we have to make sure that the security value of the returned value is less than the one specified in the policy. This corresponds to the statement " ((se i) U k_r) <= kv" where "kv" is the policy for return value.

```
| DEX_const : forall i (rt:TypeRegisters) (k:DEX_ValKind)
    (r:DEX_Reg) (v:Z),
  In r (MapList.dom rt) ->
  texec i (DEX_Const k r v) rt
    (Some (MapList.update rt r (L.Simple (se i))))
```

Similar to **Move**, **Const** also does not have a constraint except that the successor's register typing have to be at least as restricted as the current registers typing with the register $r$ updated with the current security environment. This correspond to the statement "texec i (DEX_Const k r v) rt (Some (MapList.update rt r (L.Simple (se i))))".

```
| DEX_packedSwitch : forall i k (rt:TypeRegisters) (r:DEX_Reg)
    (firstKey:Z) (size:nat) (l:list DEX_OFFSET.t),
  MapList.get rt r = Some k ->
  (forall j, region i j -> k <= se j) ->
  texec i (DEX_PackedSwitch r firstKey size l) rt (Some rt)

| DEX_sparseSwitch : forall i k (rt:TypeRegisters) (reg:DEX_Reg)
    (size:nat) (l:list (Z * DEX_OFFSET.t)),
  MapList.get rt reg = Some k ->
  (forall j, region i j -> k <= se j) ->
  texec i (DEX_SparseSwitch reg size l) rt (Some rt)

| DEX_ifcmp : forall i ka kb (rt:TypeRegisters) (cmp:DEX_CompInt)
    (ra:DEX_Reg) (rb:DEX_Reg) (o:DEX_OFFSET.t),
  In ra (MapList.dom rt) ->
  In rb (MapList.dom rt) ->
  MapList.get rt ra = Some ka ->
  MapList.get rt rb = Some kb ->
  (forall j, region i j -> (ka U kb) <= se j) ->
  texec i (DEX_Ifcmp cmp ra rb o) rt (Some rt)

| DEX_ifz : forall i k (rt:TypeRegisters) (cmp:DEX_CompInt)
```

```
    (r:DEX_Reg) (o:DEX_OFFSET.t),
  In r (MapList.dom rt) ->
  MapList.get rt r = Some k ->
  (forall j, region i j -> k <= se j) ->
  texec i (DEX_Ifz cmp r o) rt (Some rt)
```

For all the branching instructions, the registers typing of the successors have to be at least as restricted as the current registers typing. There is an additional constraint in that the instructions executing under the guard of the current instruction (executing under the region) will have their security environment lifted to the least upper bound of the security level of the registers involved. This statement, e.g., "(forall j, region i j -> k <= se j)" corresponds to the typing constraint "$\forall j' \in \mathbf{region}(i, \text{Norm}), se(i) \sqcup rt(r) \leq se(j')$".

```
| DEX_ibinop : forall i ka kb (rt:TypeRegisters) (op:DEX_BinopInt)
    (r:DEX_Reg) (ra:DEX_Reg) (rb:DEX_Reg),
  In r (MapList.dom rt) ->
  In ra (MapList.dom rt) ->
  In rb (MapList.dom rt) ->
  MapList.get rt ra = Some ka ->
  MapList.get rt rb = Some kb ->
  texec i (DEX_Ibinop op r ra rb) rt
    (Some (MapList.update rt r (L.Simple ((ka U kb) U (se i)))))

| DEX_ibinopConst : forall i ks (rt:TypeRegisters) (op:DEX_BinopInt)
    (r:DEX_Reg) (rs:DEX_Reg) (v:Z),
  In r (MapList.dom rt) ->
  In rs (MapList.dom rt) ->
  MapList.get rt rs = Some ks ->
  texec i (DEX_IbinopConst op r rs v) rt
    (Some (MapList.update rt r (L.Simple (ks U (se i)))))
```

Binary operators behave similarly to **Move** and other unary instructions except that the security level involved in the update is the least upper bound of the registers involved. To be more explicit, the update " texec i (DEX_Ibinop op r ra rb) rt (Some (MapList.update rt r (L.Simple ((ka U kb) U (se i)))))" corresponds to the statement $rt \oplus \{r \mapsto \big(rt(r_a) \sqcup rt(r_b) \sqcup se(i)\big)\}$.

### 5.2.2 Indistinguishability Relations

The indistinguishability itself is simple in nature; it revolves around being able to define what it means for a state to be indistinguishable. We define what it means for any two values to be indistinguishable, what it means for any two registers to be indistinguishable, what it means for any two registers mappings to be indistinguishable, and finally what it means for any two states to be indistinguishable. Subsequently, we

also define some properties of these indistinguishability relations which are useful in proving the soundness theorem.

**Definitions**

```
Inductive Value_in : DEX_value -> DEX_value -> Prop :=
| Value_in_num: forall n,
  Value_in (Num n) (Num n).

Inductive Value_in_opt :
  option DEX_value -> option DEX_value -> Prop :=
| Value_in_opt_some:
  forall v v',
    Value_in v v' ->
    Value_in_opt (Some v) (Some v')
| Value_in_opt_none: Value_in_opt None None.
```

These two definitions above are the simple definitions of what it means for a value to be indistinguishable. Simply put, the value is indistinguishable if the value is the same, or in the case of optional value, two values are indistinguishable also when both of them are none. This corresponds to Definition 3.3.3.

```
Inductive Reg_in (observable:L.t) (r r': DEX_Registers.t)
  (rt rt': TypeRegisters) (rn:DEX_Reg) : Prop :=
| Reg_high_in : forall k k', MapList.get rt rn = Some k ->
    MapList.get rt' rn = Some k' -> ~(L.leql k observable) ->
    ~(L.leql k' observable) -> Reg_in observable r r' rt rt' rn
| Reg_nhigh_in : Value_in_opt (DEX_Registers.get r rn)
    (DEX_Registers.get r' rn) -> Reg_in observable r r' rt rt' rn.
```

For two registers to be indistinguishable from each other, they must satisfy:

- the security level for both of them are higher than the capability of the observer, which is captured by the first clause (Reg_high_in); or
- they have to have the same value, which is captured by the second clause (Reg_nhigh_in).

This corresponds to Definition 4.1.2.

```
Inductive Regs_in (observable:L.t) (r r': DEX_Registers.t)
  (rt rt': TypeRegisters) : Prop :=
| Build_Regs_in : eq_set (MapList.dom rt) (MapList.dom rt') ->
  (forall (rn:DEX_Reg), Reg_in observable r r' rt rt' rn) ->
  Regs_in observable r r' rt rt'.
```

Two registers mapping are indistinguishable w.r.t. their respective registers typing if every register in the mapping is indistinguishable. This is captured by the only clause in the inductive definition "(forall (rn:DEX_Reg), Reg_in observable r r' rt rt' rn)", and corresponds to Definition 4.1.3.

```
Inductive st_in (observable:L.t) (rt rt':TypeRegisters) :
  DEX_PC * DEX_Registers.t ->
  DEX_PC * DEX_Registers.t -> Prop :=
| Build_st_in: forall pc pc' r r',
    Regs_in observable r r' rt rt' ->
    st_in observable rt rt' (pc,r) (pc',r').
```

Two states are indistinguishable if the registers mappings are indistinguishable. This definition corresponds to Definition 4.2.1.

```
Inductive indist_return_value (observable:L.t) (s:DEX_sign) :
    DEX_ReturnVal -> DEX_ReturnVal -> Prop :=
| indist_return_val : forall v1 v2 k,
  s.(DEX_resType) = Some k ->
  (L.leql k observable -> Value_in v1 v2) ->
  indist_return_value observable s
    (Normal (Some v1)) (Normal (Some v2))
| indist_return_void :
  s.(DEX_resType) = None ->
  indist_return_value observable s (Normal None) (Normal None).
```

Since there are two return instructions, there are also two cases where the return value is deemed to be indistinguishable:

- If the instruction is returning a value, then if the security level is lower than the capability of the observer, the returned values have to be the same. This statement " (L.leql k observable -> Value_in v1 v2)" corresponds to the first clause of Definition 4.1.7 ($\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \quad \vec{k_r}[n] \leq k_{\text{obs}} \Rightarrow v_1 \sim_\beta v_2}{(v_1,h_1) \sim_{k_{\text{obs}},\vec{k_r},\beta} (v_2,h_2)}$).
- If the instruction is not returning any value, then it is indistinguishable if both of the return values are indeed nothing (indist_return_void).

```
Inductive state : Type :=
  intra : DEX_IntraNormalState -> TypeRegisters -> state
| ret : DEX_ReturnVal -> state.

Inductive indist (observable:L.t) (p:DEX_ExtendedProgram)
  (m:DEX_Method) (sgn:DEX_sign) : state -> state -> Prop :=
| indist_intra : forall pc pc' r r' rt rt',
  st_in observable rt rt' (pc,r) (pc',r') ->
  indist observable p m sgn (intra (pc,r) rt) (intra (pc',r') rt')
| indist_return : forall v v',
  indist_return_value observable sgn v v'->
  indist observable p m sgn (ret v) (ret v').
```

Finally indistinguishable itself is defined between two return values or two states of execution.

**Properties** In particular, we are interested in the symmetry and transitivity of the indistinguishability relations defined above.

```
Lemma Value_in_sym : forall v1 v2, Value_in v1 v2 -> Value_in v2 v1.

Lemma Value_in_opt_sym : forall v1 v2,
  Value_in_opt v1 v2 -> Value_in_opt v2 v1.

Lemma Value_in_trans : forall v1 v2 v3,
  Value_in v1 v2 -> Value_in v2 v3 -> Value_in v1 v3.

Lemma Value_in_opt_trans : forall v1 v2 v3,
  Value_in_opt v1 v2-> Value_in_opt v2 v3 -> Value_in_opt v1 v3.

Lemma Reg_in_sym : forall obs r r' rt rt' rn,
  Reg_in obs r r' rt rt' rn ->  Reg_in obs r' r rt' rt rn.

Lemma Regs_in_sym : forall r1 r2 rt1 rt2,
  Regs_in kobs r1 r2 rt1 rt2 -> Regs_in kobs r2 r1 rt2 rt1.

Lemma st_in_sym : forall rt rt' r r',
  st_in kobs rt rt' r r' -> st_in kobs rt' rt r' r.
```

There are also some useful lemmas.

```
Lemma Reg_in_upd_low:
  forall k (v v' : DEX_value) (r r' : DEX_Registers.t)
      (rt rt' : TypeRegisters) (reg : DEX_Reg) (b b': FFun.t DEX_Location),
    Reg_in kobs b b' r r' rt rt' reg ->
    Value_in b b' v v' ->
    L.leql k kobs ->
    Reg_in kobs b b' (DEX_Registers.update r reg v)
      (DEX_Registers.update r' reg v') (MapList.update rt reg k)
      (MapList.update rt' reg k) reg.
```

This lemma specifies that update in low security environment with the same value will preserve register indistinguishability.

```
Lemma Reg_in_upd_high:
  forall k k' (v v' : DEX_value) (r r' : DEX_Registers.t)
      (rt rt' : TypeRegisters) (reg : DEX_Reg) (b b': FFun.t DEX_Location),
    Reg_in kobs b b' r r' rt rt' reg ->
    ~L.leql k kobs ->
    ~L.leql k' kobs ->
```

```
    Reg_in kobs b b' (DEX_Registers.update r reg v)
      (DEX_Registers.update r' reg v') (MapList.update rt reg k)
      (MapList.update rt' reg k') reg.
```

This lemma specifies that update in high security environment will always preserve register indistinguishability.

### 5.2.3 Non-Interference Proof for DEX$_{\mathcal{I}}$

**High Result** For a return value to be a high result, then the security level of the return value must be higher than the observer capability. The second clause corresponds to Definition 4.2.3.

```
Inductive high_result (observable:L.t) (s:DEX_sign) :
  DEX_ReturnVal -> Prop :=
| high_result_void :
  s.(DEX_resType) = None ->
  high_result observable s (Normal None)
| high_result_value : forall v k,
  s.(DEX_resType) = Some k ->
  ~ L.leql k observable ->
  high_result observable s (Normal (Some v)).
```

**High Branching** The high branching lemma states that in the case where two executions from the same program point branched to different program points, then all the program points executing in the region will have high security environment. This corresponds to Lemma 4.2.12

```
Lemma soap2_intra_normal :
 forall sgn pc pc2 pc2' i r1 rt1 r1' rt1' r2 r2' rt2 rt2' ,
   instructionAt m pc = Some i ->
   NormalStep se reg m sgn i (pc,r1) rt1 (pc2,r2) rt2 ->
   NormalStep se reg m sgn i (pc,r1') rt1' (pc2',r2') rt2' ->
   pc2 <> pc2' ->
   st_in kobs rt1 rt1' (pc,r1) (pc,r1') ->

    forall j, reg pc j -> ~ L.leql (se j) kobs.
```

"st_in kobs rt1 rt' (pc,r1) pc,r1')" corresponds to "$s_1 \sim_{k_{\text{obs}},rt_1,rt_2} s_2$". "NormalStep se reg m sgn i (pc,r1) rt1 (pc2,r2) rt2" corresponds to "$s_1 \rightsquigarrow \langle i_1, \rho_1' \rangle$" and "$i \vdash rt_1 \Rightarrow rt_1'$", "NormalStep se reg m sgn i (pc, r1') rt1' (pc2', r2') rt2' " corresponds to " $s_2 \rightsquigarrow \langle i_2, \rho_2' \rangle$" and "$i \vdash rt_2 \Rightarrow rt_2'$", "pc2 <> pc2' " corresponds to "$i_1 \neq i_2$", " forall j, reg pc j -> L.leql (se j) kobs" corresponds to "$\forall j \in \textbf{region}(i), se(j) \not\leq k_{\text{obs}}$".

**Locally Respect** The locally respect lemma states that at the same program point, an instruction will execute into two indistinguishable states or return value. Although this is different from the specialized case where the successors have to be the same

program point, this lemma still holds for the case where there is no exception involved.

```
Lemma indist2_intra : forall m sgn se rt ut ut' s s' u u',
      forall H0:P (SM _ _ m sgn),
        indist sgn rt rt  s s' ->
        pc s = pc s' ->
        exec m s (inl _ u) ->
        exec m s' (inl _ u') ->
        texec m (PM_P _ H0) sgn se (pc s) rt (Some ut) ->
        texec m (PM_P _ H0) sgn se (pc s) rt (Some ut') ->
          indist sgn ut ut' u u'.

Lemma indist2_return : forall (m : Method) (sgn : Sign) (se : PC -> L.t)
      (rt : registertypes) (s s' : istate) (u u' : rstate) ,
      forall H:P (SM Method Sign m sgn),
        indist sgn rt rt s s' ->
        pc s = pc s' ->
        exec m s (inr istate u) ->
        exec m s' (inr istate u') ->
        texec m (PM_P _ H) sgn se (pc s) rt None ->
        texec m (PM_P _ H) sgn se (pc s) rt None ->
          rindist sgn u u'.
```

In the formalization, we split the lemma into two lemmas, one case is where the successors are still within the program, and the other case is when the successors are return points. We first show the correspondence for the case where the successors are still within the program. "indist sgn rt rt s s' " corresponds to "$s_1 \sim_{k_{\text{obs}},rt_1,rt_2} s_2$", "exec m s (inl _ u)" corresponds to "$s_1 \rightsquigarrow s_1'$", "texec m (PM_P _ H0) sgn se (pc s) rt (Some ut)" corresponds to "$i \vdash rt_1 \Rightarrow rt_1'$", "exec m s' (inl _ u')" corresponds to "$s_2 \rightsquigarrow s_2'$", "texec m (PM_P _ H0) sgn se (pc s) rt (Some ut')" corresponds to "$i \vdash rt_2' \Rightarrow rt_2'$", "indist sgn ut ut' u u' " corresponds to "$s_1' \sim_{k_{\text{obs}},rt_1',rt_2'} s_2'$".

For the case where it involves return points, "exec m s (inr istate u)" corresponds to "$s_1 \rightsquigarrow v_1$", "texec m (PM_P _ H) sgn se (pc s) rt None" corresponds to "$i \vdash rt_1 \Rightarrow$", "exec m s' (inr istate u')" corresponds to "$s_2 \rightsquigarrow v_2$", "texec m (PM_P _ H) sgn se (pc s) rt None" corresponds to "$i \vdash rt_2' \Rightarrow$", "rindist sgn u u' " corresponds to "$k_r \leq k_{\text{obs}}$ implies $v_1 \sim v_2$".

**Indistinguishability at Junction Point** Ultimately, the main part of our proof which is different from that of Barthe et al. is in proving the indistinguishability of program point at the junction point after a branch, after which we can use the induction hypothesis to conclude the proof. It relies on several definitions below, namely the execution path, that the path is in the region, and the notion of change.

```
Inductive path (m:Method) (i:istate) : istate -> Type :=
  | path_base : forall j, exec m i (inl j) -> path m i j
```

```
  | path_step : forall j k, path m k j -> exec m i (inl k)
      -> path m i j.

Inductive path_in_region (m:Method) (cdr: CDR (step m)) (s:PC)
(i j:istate) : (path m i j) -> Prop :=
  | path_in_reg_base : forall (Hexec:exec m i (inl j)),
      region cdr s (pc i) ->
      path_in_region m cdr s i j (path_base m i j Hexec)
  | path_in_reg_ind : forall k (Hexec:exec m i (inl k)) (p:path m k j),
      region cdr s (pc i) -> path_in_region m cdr s k j p ->
      path_in_region m cdr s i j (path_step m i j k p Hexec).
```

The path is defined as the transitive closure of the execution step between any two states in the execution path. The base of this induction is taking just one execution step, and we can construct the path by adding one more step to the existing path. We decided not to use a path of length 0 as the base because it is possible for an execution path to be a loop, thus creating ambiguity. The inductive definition path_in_region corresponds to the statement "an execution trace $\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i_k, \rho_k, h_k\rangle \rightsquigarrow_{m,\tau_k} (r, h_r)$ where $\langle i_0, \rho_0, h_0\rangle \in \mathbf{region}(s, \tau)$" in Lemma 4.2.26, Lemma 4.2.27 and Lemma 4.2.28.

```
Inductive changed_at (m:Method) (i:istate) (r:Reg) : Prop :=
  | const_change : forall k v, instructionAt m (pc i) =
      Some (DEX_Const k r v) -> changed_at m i r
  | move_change : forall k rs, instructionAt m (pc i) =
      Some (DEX_Move k r rs) -> changed_at m i r
  | ineg_change : forall rs, instructionAt m (pc i) =
      Some (DEX_Ineg r rs) -> changed_at m i r
  | inot_change : forall rs, instructionAt m (pc i) =
      Some (DEX_Inot r rs) -> changed_at m i r
  | i2b_change : forall rs, instructionAt m (pc i) =
      Some (DEX_I2b r rs) -> changed_at m i r
  | i2s_change : forall rs, instructionAt m (pc i) =
      Some (DEX_I2s r rs) -> changed_at m i r
  | ibinop_change : forall op ra rb, instructionAt m (pc i) =
      Some (DEX_Ibinop op r ra rb) -> changed_at m i r
  | ibinopConst_change : forall op rs v, instructionAt m (pc i) =
      Some (DEX_IbinopConst op r rs v) -> changed_at m i r.

Inductive changed (m:Method) (i j: istate) :
(path m i j) -> Reg -> Prop :=
  | changed_onestep : forall r (p:path m i j), changed_at m i r
      -> changed m i j p r
  | changed_path : forall k r (p:path m k j) (H:exec m i (inl k)),
      changed m k j p r -> changed m i j
      (path_step Method istate rstate exec m i j k p H) r.
```

The definition of changed refers to any modifications that happen to a register. The change can happen anywhere within a path, but at least change must exist for this predicate to be true. The change itself is defined as the target register of any instructions that modify its value, regardless whether the instruction itself is actually modifying the value. It is possible to change the register's value to the same value, and it is still counted as a change.

```
Lemma not_changed_same :
  forall m sgn i j (Hpath: path m i j) r (H: P (SM _ _ m sgn)) ,
  ~changed m i j Hpath r -> (same_reg_val i j r) /\
    (high_reg (RT m sgn (pc i)) r -> high_reg (RT m sgn (pc j)) r).
```

This lemma states that in an execution trace, any register that is not changed will have the same value and security level. This lemma corresponds to Lemma 4.2.29 where "(indist_reg_val i j r)" corresponds to "$\rho_0(r) = \rho_k(r)$" and "(high_reg (RT m sgn (pc i)) r -> high_reg (RT m sgn (pc j)) r)" corresponds to "$RT_0(r) = RT_k(r)$".

```
Lemma changed_high : forall m sgn s i j r (H:P (SM _ _ m sgn))
    (Hpath: path m (* sgn (PM_P _ H) *) i j),
  (forall k:PC, region (cdr m (PM_P _ H)) s k ->
    ~ L.leql (se m sgn k) kobs) ->
  path_in_region m (cdr m (PM_P _ H)) s i j Hpath ->
  region (cdr m (PM_P _ H)) s (pc i) ->
  junc (cdr m (PM_P _ H)) s (pc j) ->
  changed m i j Hpath r -> high_reg (RT m sgn (pc j)) r.
```

This lemma states that any changes in the high region will render the security level of the affected register to be lifted to high as well. This corresponds to Lemma 4.2.28, where "path_in_region m (cdr m (PM_P _ H)) s k" corresponds to "$\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i_k, \rho_k, h_k\rangle$ where $\langle i_0, \rho_0, h_0\rangle \in \mathbf{region}(s, \tau)$". " (forall k:PC, region (cdr m (PM_P _ H)) s k -> L.leql (se m sgn k) kobs)" corresponds to "$se$ is high in $\mathbf{region}(s, \tau)$", "junc (cdr m (PM_P _ H)) s (pc j)" corresponds to "$k = \mathbf{jun}(s, \tau)$", "changed m i j Hpath r" corresponds to "the value of $r$ is changed by one or more instruction in the execution trace", "high_reg (RT m sgn (pc j)) r" corresponds to "$RT_k(r)$ is high".

```
Lemma junction_indist :
  forall m sgn ns ns' s s' u u' res res' i (H: P (SM m sgn)),
  indist sgn (RT m sgn (pc s)) (RT m sgn (pc s')) s s' ->
  exec m s (inl u) -> exec m s' (inl u') ->
  region (cdr m (PM_P _ H)) i (pc u) ->
  region (cdr m (PM_P _ H)) i (pc u') ->
  high_region m (PM_P _ H) sgn i ->
  evalsto m ns u res -> evalsto m ns' u' res' ->
  indist sgn (RT m sgn (pc u)) (RT m sgn (pc u')) u u' ->
  (exists v, exists v', exists ps, exists ps',
```

```
    evalsto m ps v res /\ ps <= ns /\
    evalsto m ps' v' res' /\ ps' <= ns' /\
    junc (cdr m (PM_P _ H)) i (pc v) /\
    junc (cdr m (PM_P _ H)) i (pc v') /\
    indist sgn (RT m sgn (pc v)) (RT m sgn (pc v')) v v')
  \/ (high_result sgn res /\ high_result sgn res').

Lemma junction_indist_2 :
  forall m sgn ns ns' s s' u u' res res' i (H: P (SM m sgn)),
  indist sgn (RT m sgn (pc s)) (RT m sgn (pc s')) s s' ->
  exec m s (inl u) -> exec m s' (inl u') ->
  region (cdr m (PM_P _ H)) i (pc u) ->
  junc (cdr m (PM_P _ H)) i (pc u') ->
  high_region m (PM_P _ H) sgn i ->
  evalsto m ns u res -> evalsto m ns' u' res' ->
  indist sgn (RT m sgn (pc u)) (RT m sgn (pc u')) u u' ->
  (exists v, exists ps,
    evalsto m ps v res /\ ps <= ns /\
    junc (cdr m (PM_P _ H)) i (pc v) /\
    indist sgn (RT m sgn (pc v)) (RT m sgn (pc u')) v u').
```

Essentially, these two lemmas are the same except that the second lemma deals with the case where one of the successors is a junction point. This corresponds to Lemma 4.2.30, although it is not a precise implementation in that there is no mentioning of the indistinguishability of the successors. The reason for the difference is that since locally respect lemma is still not specialized without the exception mechanism, we decided to use it directly. This implementation serves as the basis to extend even to include exception mechanism, as we use this additional assumption to enable us to use Lemma 4.2.28. In the exception submachine where there is no additional assumption, we need to show that executing one step instruction indeed still resulting in the change of the register to high security level.

**Type Check Definition**

```
Definition check : bool := for_all_P p
  (fun m sgn =>
    (check_rt0 m sgn) &&
    for_all_steps_m m
    (fun i ins oj =>
      DEX_tcheck m sgn (se m sgn) (selift m sgn) (RT m sgn) i ins)
  ).

Definition check_ni (p:DEX_ExtendedProgram) reg jun se RT : bool :=
  check_well_formed_lookupswitch p &&
  check_all_cdr p reg jun &&
  check p se RT reg.
```

The definition of the type check corresponds to Definition 4.1.1. "(check_rt0 m sgn)" corresponds to "$RT_1 = \vec{k_a}$", i.e., the initial configuration of the registers typing, and the other clause corresponds to the two conditions of type check for instructions that executes within a method and return points. In particular, it corresponds to the following conditions: $\forall i, j \in \mathcal{PP}, e \in \{\text{Norm} + \mathcal{C}\}$:

- $i \mapsto^e j$ implies there exists $rt \in (\mathcal{R} \to \mathcal{S})$ such that $\Gamma$, **ft**, **region**, $se$, $sgn$, $i \vdash^e RT_i \Rightarrow$ $rt$ and $rt \sqsubseteq RT_j$;
- $i \mapsto^e$ implies $\Gamma$, **ft**, **region**, $se$, $sgn$, $i \vdash^e RT_i \Rightarrow$.

**Type System Soundness**

```
Definition NI (p:DEX_ExtendedProgram) : Prop :=
  forall kobs m sgn i r1 r2 res1 res2,
    P p (SM _ _ m sgn) ->
    init_pc m i ->
    indist kobs sgn (rt0 m sgn) (rt0 m sgn) (i,r1) (i,r2) ->
    DEX_BigStepAnnot.DEX_BigStep p.(DEX_prog) m (i,r1) (res1) ->
    DEX_BigStepAnnot.DEX_BigStep p.(DEX_prog) m (i,r2) (res2) ->
      indist_return_value kobs sgn res1 res2.

Theorem check_ni_correct : forall p reg jun se RT,
  check_ni p reg jun se RT = true ->
  NI p.
```

The definition of non-interference match Definition 4.1.8. In particular,

- "indist kobs sgn (rt0 m sgn) (rt0 m sgn) (i,r1) (i,r2)" matches "$\rho_1 \sim_{k_{\text{obs}}, RT_0, RT_0, \beta} \rho_2$",
- "DEX_BigStepAnnot.DEX_BigStep p.(DEX_prog) m (i,r1) (res1)" corresponds to "$\langle 1, \rho_1, h_1 \rangle \rightsquigarrow_m^+ v_1, h_1'$",
- "DEX_BigStepAnnot.DEX_BigStep p.(DEX_prog) m (i,r1) (res1)" corresponds to "$\langle 1, \rho_2, h_2 \rangle \rightsquigarrow_m^+ v_2, h_2'$",
- "indist_return_value kobs sgn res1 res2" corresponds to "$(v_1, h_1') \sim_{k_{\text{obs}}, \vec{k_r}, \beta'} (v_2, h_2')$".

The final theorem states that if a program is typable then it is non-interferent.

## 5.3 Formalization of $DEX_{\mathcal{O}}$

In this section, we provide the formalization for $DEX_{\mathcal{O}}$. There are many similarities with the proof for $DEX_{\mathcal{I}}$. So, we will focus only on the differences here. Proving $DEX_{\mathcal{O}}$ serves to show that there is no problem in extending the proof a la Barthe et al. to include the method calling, exceptions, and arrays.

### 5.3.1 Transfer Rules

In this section, we show the correspondence between the transfer rules and its formalization in Coq.

```
| DEX_iget : forall i f k ko r ro rt,
  In r (MapList.dom rt) ->
  In ro (MapList.dom rt) ->
  MapList.get rt ro = Some ko ->
  texec i (DEX_Iget k r ro f) rt (Some
    (MapList.update rt r (L.Simple ((se i) U (ko U (DEX_ft p f))))))
```

**Iget** does not have any constraints, just the constraint that the next registers typing have to be at least as restricted as the current registers typing with register $r$ updated with the lower bound of the current security environment, the security level of $r_o$, and the field security level ($\mathbf{ft}(f)$).

```
| DEX_iput : forall i f k ko ks ro rs rt,
  In rs (MapList.dom rt) ->
  In ro (MapList.dom rt) ->
  MapList.get rt ro = Some ko ->
  MapList.get rt rs = Some ks ->
  ks <= DEX_ft p f ->
  ko <= DEX_ft p f ->
  se i <= DEX_ft p f ->
  texec i (DEX_Iput k rs ro f) rt (Some rt)
```

In the context of DEX$_{\mathcal{O}}$, **Iput** only has one constraint, i.e., $(rt(r_o) \sqcup se(i)) \sqcup^{\text{ext}} rt(r_s) \leq \mathbf{ft}(f)$. This constraint is captured by the three constraint in the formalization: "ks <= DEX_ft p f", "ko <= DEX_ft p f", and "se i <= DEX_ft p f", where "ks" is "$rt(r_s)$", and "ko" is "$rt(r_o)$".

```
| DEX_new : forall i r rt c,
  In r (MapList.dom rt) ->
  texec i (DEX_New r c) rt
    (Some (MapList.update rt r (L.Simple (se i))))
```

**New** also does not have any constraints, just the constraint that the next registers typing have to be at least as restricted as the current registers typing with register $r$ updated with the current security environment. This corresponds to the "texec i (DEX_New r c) rt (Some (MapList.update rt r (L.Simple (se i))))". Implicit in this typing rule is that the $\beta$ mapping needs to be updated when the current security environment is low. This is best captured by the combination of the operational semantics and typing rule.

```
Inductive NormalStep_new (reg:DEX_Reg) (c:DEX_ClassName)
```

```
(m:DEX_Method) (sgn:DEX_sign)  :
  DEX_IntraNormalState -> TypeRegisters ->
  FFun.t DEX_Location -> DEX_IntraNormalState ->
  TypeRegisters -> FFun.t DEX_Location -> Prop :=
| new : forall pc pc' h h' r r' loc rt rt' b,

  In reg (DEX_Registers.dom r) ->
  In reg (MapList.dom rt) ->
  next m pc = Some pc' ->
  DEX_Heap.new h p (DEX_Heap.DEX_LocationObject c) =
    Some (pair loc h') ->
  r' = DEX_Registers.update r reg (Ref loc) ->
  rt' = MapList.update rt reg (se pc) ->

  NormalStep_new reg c m sgn (pc,(h,r)) rt b (pc',(h',r')) rt'
    (newb (se pc) b loc).
```

where "newb" is a function which either returns the previous $\beta$ mapping if the current security environment is high or updates the $\beta$ mapping if the current security environment is low.

```
Definition newb (k:L.t) (b:FFun.t DEX_Location)
    (loc:DEX_Location) : FFun.t DEX_Location :=
  if L.leql_dec k kobs then (FFun.extends b loc) else b.
```

### 5.3.2 Indistinguishability Relations

The indistinguishability relations are really similar to that of $DEX_{\mathcal{I}}$, with the difference that the value indistinguishability to also include indistinguishable object reference. We also have indistinguishability between objects and indistinguishability between heaps. The state indistinguishability also depends on heap indistinguishability now. The indistinguishability itself is simple in nature; it revolves around being able to define what it means for a state to be indistinguishable. We define what it means for any two values to be indistinguishable, what it means for any two registers to be indistinguishable, what it means for any two registers mappings to be indistinguishable, and finally what it means for any two states to be indistinguishable. Subsequently, we also define some properties of these indistinguishability relations which are useful in proving the soundness theorem.

**Definitions**

```
Inductive Value_in  (b b':FFun.t DEX_Location) :
  DEX_value -> DEX_value -> Prop :=
| Value_in_null: Value_in b b' Null Null
| Value_in_num: forall n,
  Value_in b b' (Num n) (Num n)
| Value_in_ref: forall loc loc' n,
```

```
  FFun.lookup b n = Some loc ->
  FFun.lookup b' n = Some loc' ->
  Value_in b b' (Ref loc) (Ref loc').

Inductive Value_in_opt (b b':FFun.t DEX_Location) :
  option DEX_value -> option DEX_value -> Prop :=
| Value_in_opt_some:
  forall v v',
    Value_in b b' v v' ->
    Value_in_opt b b' (Some v) (Some v')
| Value_in_opt_none: Value_in_opt b b' None None.
```

These value indistinguishability definitions are similar to before except the additional clause when they are dealing with reference type value. Two reference type value are indistinguishable if both of them are null, or they have the same index in the $\beta$ mapping. This is slightly different from Definition 3.3.3, but they behave in the same manner.

```
Record hp_in (observable:L.t) (ft:DEX_FieldSignature -> L.t)
    (b b': FFun.t DEX_Location) (h h': DEX_Heap.t) : Prop :=
  make_hp_in {
    object_in : forall n loc loc' f cn cn',
      FFun.lookup b n = Some loc ->
      FFun.lookup b' n = Some loc' ->
      DEX_Heap.typeof h loc = Some (DEX_Heap.DEX_LocationObject cn) ->
      DEX_Heap.typeof h' loc' = Some (DEX_Heap.DEX_LocationObject cn') ->
      L.leql (ft f) observable->
      Value_in_opt b b'
      (DEX_Heap.get h (DEX_Heap.DEX_DynamicField loc f))
      (DEX_Heap.get h' (DEX_Heap.DEX_DynamicField loc' f));
    class_object_in : forall n loc loc',
      FFun.lookup b n = Some loc ->
      FFun.lookup b' n = Some loc' ->
      DEX_Heap.typeof h loc = DEX_Heap.typeof h' loc';
    compat_ffun : FFun.compat b b';
    left_inj : FFun.is_inj b;
    right_inj : FFun.is_inj b';
    left_heap_compat : ffun_heap_compat b h;
    right_heap_compat : ffun_heap_compat b' h'
  }.
```

For two heaps to be indistinguishable from each other, they must satisfy:

- for any two objects with indistinguishable location, if $\mathbf{ft}(f)$ is low, then the field must contain the same value;

- $\beta$ and $\beta'$ is a bijection; and
- for every location in $\beta$, there is a corresponding mapping in the heap.

This corresponds to Definition 4.1.4 with the minor difference in the $\beta$ mapping.

```
Inductive st_in (observable:L.t) (ft:DEX_FieldSignature -> L.t)
    (b b':FFun.t DEX_Location) (rt rt':TypeRegisters) :
  DEX_PC * DEX_Heap.t * DEX_Registers.t ->
  DEX_PC * DEX_Heap.t * DEX_Registers.t -> Prop :=
| Build_st_in: forall pc pc' h h' r r',
    Regs_in observable b b' r r' rt rt' ->
    hp_in observable ft b b' h h' ->
    st_in observable ft b b' rt rt' (pc,h,r) (pc',h',r').
```

Two states are indistinguishable if the registers mappings are indistinguishable and the heaps are indistinguishable. This definition corresponds to Definition 4.2.1.

The definition of indistinguishability between return values is really similar with $DEX_{\mathcal{I}}$ with the addition that now it also involves heap. The final indistinguishability relation itself is the same as $DEX_{\mathcal{I}}$, i.e., it is defined between two return values or two states of execution.

**Properties** The properties of indistinguishability relations defined in $DEX_{\mathcal{I}}$ carry over to $DEX_{\mathcal{O}}$ with a minor adjustment of heap and $\beta$ mapping. Additionally, there are also some useful properties of the heap and also the auxiliary lemmas.

```
Lemma hp_in_sym : forall h1 h2 b1 b2,
    hp_in kobs ft b1 b2 h1 h2 -> hp_in kobs ft b2 b1 h2 h1.

Lemma hp_in_trans : forall h1 h2 h3 b1 b2 b3,
    hp_in kobs ft b1 b2 h1 h2 ->
    hp_in kobs ft b2 b3 h2 h3 ->
    hp_in kobs ft b1 b3 h1 h3.

Lemma hp_in_putfield_high_update_left : forall loc b b' h h' f v cn,
    hp_in kobs ft b b' h h' ->
    ~ (L.leql (ft f) kobs) ->
    DEX_Heap.typeof h loc = Some (DEX_Heap.DEX_LocationObject cn) ->
    hp_in kobs ft b b'
    (DEX_Heap.update h (DEX_Heap.DEX_DynamicField loc f) v) h'.
```

This lemma corresponds to Lemma 4.2.5, i.e., an update in high security environment preserves heap indistinguishability.

```
Lemma ffun_extends_hp_in_new_left: forall c b b' h h' hn loc,
    hp_in kobs ft b b' h h' ->
    DEX_Heap.new h p (DEX_Heap.DEX_LocationObject c) =
```

```
      Some (pair loc hn) ->
    hp_in kobs ft b b' hn h'.

Lemma ffun_extends_hp_in_new_right: forall c b b' h h' hn' loc,
    hp_in kobs ft b b' h h' ->
    DEX_Heap.new h' p (DEX_Heap.DEX_LocationObject c) =
      Some (pair loc hn') ->
    hp_in kobs ft b b' h hn'.

Lemma ffun_extends_hp_in_simpl: forall c c' b b' h h' hn hn' loc loc',
    hp_in kobs ft b b' h h' ->
    DEX_Heap.new h p (DEX_Heap.DEX_LocationObject c) =
      Some (pair loc hn) ->
    DEX_Heap.new h' p (DEX_Heap.DEX_LocationObject c') =
      Some (pair loc' hn') ->
    hp_in kobs ft b b' hn hn'.
```

These three lemmas constitute Lemma 4.2.8.

### 5.3.3 Non-Interference Proof for DEX$_{\mathcal{O}}$

The definition of high result is still the same as DEX$_{\mathcal{I}}$ with the addition of the heap. The definition of high branching lemma locally respect lemma are also the same with the addition of the heap and $\beta$ mapping.

```
Lemma indist2_intra : forall m sgn se rt ut ut' s s' u u' b b',
      forall H0:P (SM _ _ m sgn),
        indist sgn rt rt b b' s s' ->
        pc s = pc s' ->
        exec m s (inl _ u) ->
        exec m s' (inl _ u') ->
        texec m (PM_P _ H0) sgn se (pc s) rt (Some ut) ->
        texec m (PM_P _ H0) sgn se (pc s) rt (Some ut') ->
        exists bu, exists bu',
          border b bu /\ border b' bu' /\
          indist sgn ut ut' bu bu' u u'.

Lemma indist2_return : forall (m : Method) (sgn : Sign) (se : PC -> L.t)
      (rt : registertypes) (s s' : istate) (u u' : rstate) (b b' : pbij),
      forall H:P (SM Method Sign m sgn),
        indist sgn rt rt b b' s s' ->
        pc s = pc s' ->
        exec m s (inr istate u) ->
        exec m s' (inr istate u') ->
        texec m (PM_P _ H) sgn se (pc s) rt None ->
```

```
        texec m (PM_P _ H) sgn se (pc s) rt None ->
        exists bu, exists bu',
          border b bu /\ border b' bu' /\
          rindist sgn bu bu' u u'.
```

The requirement "border b bu /\ border b' bu' " corresponds to $\beta \subseteq \beta'$ in Lemma 4.2.14.
**Indistinguishability at Junction Point** The definition of path and change are the same as that of $DEX_{\mathcal{I}}$ with the addition of the three new instructions **iget**, **iput**, and **new**.

```
Inductive changed_at (m:Method) (i:istate) (r:Reg) : Prop :=
  | const_change : forall k v, instructionAt m (pc i) =
      Some (DEX_Const k r v) -> changed_at m i r
  | move_change : forall k rs, instructionAt m (pc i) =
      Some (DEX_Move k r rs) -> changed_at m i r
  | ineg_change : forall rs, instructionAt m (pc i) =
      Some (DEX_Ineg r rs) -> changed_at m i r
  | inot_change : forall rs, instructionAt m (pc i) =
      Some (DEX_Inot r rs) -> changed_at m i r
  | i2b_change : forall rs, instructionAt m (pc i) =
      Some (DEX_I2b r rs) -> changed_at m i r
  | i2s_change : forall rs, instructionAt m (pc i) =
      Some (DEX_I2s r rs) -> changed_at m i r
  | ibinop_change : forall op ra rb, instructionAt m (pc i) =
      Some (DEX_Ibinop op r ra rb) -> changed_at m i r
  | ibinopConst_change : forall op rs v, instructionAt m (pc i) =
      Some (DEX_IbinopConst op r rs v) -> changed_at m i r
  | iget_change : forall t ro f, instructionAt m (pc i) =
      Some (DEX_Iget t r ro f) -> changed_at m i r
  | new_change : forall c, instructionAt m (pc i) =
      Some (DEX_New r c) -> changed_at m i r..
```

Since now we also need to make sure that during the execution the heap indistinguishability is maintained, we have proven the following lemmas.

```
Lemma high_path_heap_indist_onestep_left : forall m sgn s i i' b b' j
    (H: P (SM _ _ m sgn)),
  (forall k:PC, region (cdr m (PM_P _ H)) s k ->
    ~ L.leql (se m sgn k) kobs) ->
  region (cdr m (PM_P _ H)) s (pc i) ->
  indist_heap i i' b b' ->
  exec m i (inl j) ->
    indist_heap j i' b b'.

Lemma high_path_heap_indist_onestep_right : forall m sgn s i i' b b' j
    (H: P (SM _ _ m sgn)),
```

```
  (forall k:PC, region (cdr m (PM_P _ H)) s k ->
    ~ L.leql (se m sgn k) kobs) ->
  region (cdr m (PM_P _ H)) s (pc i') ->
  exec m i' (inl j) ->
  indist_heap i i' b b' ->
    indist_heap i j b b'.

Lemma high_path_heap_indist : forall m sgn s i i' b b' j
    (H:P (SM _ _ m sgn)) (Hpath: path m i j),
  (forall k:PC, region (cdr m (PM_P _ H)) s k ->
    ~ L.leql (se m sgn k) kobs) ->
  path_in_region m (cdr m (PM_P _ H)) s i j Hpath ->
  region (cdr m (PM_P _ H)) s (pc i) ->
  junc (cdr m (PM_P _ H)) s (pc j) ->
  indist_heap i i' b b' ->
    indist_heap j i' b b'.

Lemma high_step_indist_heap_result : forall m sgn u u' b b' res res' i
    (H: P (SM _ _ m sgn)),
  (forall k:PC, region (cdr m (PM_P _ H)) i k ->
    ~ L.leql (se m sgn k) kobs) ->
  (forall jun : PC, ~ junc
    (cdr m (PM_P {| unSign := m; sign := sgn |} H)) i jun) ->
  region (cdr m (PM_P {| unSign := m; sign := sgn |} H)) i (pc u) ->
  region (cdr m (PM_P {| unSign := m; sign := sgn |} H)) i (pc u') ->
  indist_heap u u' b b' ->
  exec m u (inr res) -> exec m u' (inr res') ->
    indist_heap_result res res' b b'.
```

These lemmas are saying that in a high region, the heap indistinguishability is preserved throughout the execution.

```
Lemma junction_indist : forall m sgn ns ns' s s' u u' b b' bu bu'
    res res' i (H: P (SM m sgn)),
  indist sgn (RT m sgn (pc s)) (RT m sgn (pc s')) b b' s s' ->
  exec m s (inl u) -> exec m s' (inl u') ->
  region (cdr m (PM_P _ H)) i (pc u) ->
  region (cdr m (PM_P _ H)) i (pc u') ->
  high_region m (PM_P _ H) sgn i ->
  evalsto m ns u res ->
  evalsto m ns' u' res' ->
  indist sgn (RT m sgn (pc u)) (RT m sgn (pc u')) bu bu' u u' ->
  border b bu -> border b' bu' ->
  (exists v, exists v', exists ps, exists ps', exists bv, exists bv',
    evalsto m ps v res /\ ps <= ns /\
```

```
    evalsto m ps' v' res' /\ ps' <= ns' /\
    junc (cdr m (PM_P _ H)) i (pc v) /\
    junc (cdr m (PM_P _ H)) i (pc v') /\
    border bu bv /\ border bu' bv' /\
    indist sgn (RT m sgn (pc v)) (RT m sgn (pc v')) bv bv' v v')
  \/ (exists br, exists br', border bu br /\ border bu' br' /\
      indist_heap_result res res' br br' /\
      high_result sgn res /\ high_result sgn res').

Lemma junction_indist_2 : forall m sgn ns ns' s s' u u' b b' bu bu'
    res res' i (H: P (SM m sgn)),
  indist sgn (RT m sgn (pc s)) (RT m sgn (pc s')) b b' s s' ->
  exec m s (inl u) -> exec m s' (inl u') ->
  region (cdr m (PM_P _ H)) i (pc u) ->
  junc (cdr m (PM_P _ H)) i (pc u') ->
  high_region m (PM_P _ H) sgn i ->
  evalsto m ns u res ->
  evalsto m ns' u' res' ->
  indist sgn (RT m sgn (pc u)) (RT m sgn (pc u')) bu bu' u u' ->
  border b bu -> border b' bu' ->
  (exists v, exists ps, exists bv,
    evalsto m ps v res /\ ps <= ns /\
    junc (cdr m (PM_P _ H)) i (pc v) /\
    border bu bv /\
    indist sgn (RT m sgn (pc v)) (RT m sgn (pc u')) bv bu' v u').
```

These two lemmas are similar to the one in $DEX_{\mathcal{I}}$ with the additional requirement that the $\beta$ mapping must be in order ($\beta \subseteq \beta'$).

The definition of type check is still the same as that of $DEX_{\mathcal{I}}$. The definition of non-interference and type system soundness are also similar to $DEX_{\mathcal{I}}$ with the addition of $\beta$ mapping and heap.

**Type System Soundness**

```
Definition NI (p:DEX_ExtendedProgram) : Prop :=
  forall kobs m sgn i h1 h2 r1 r2 hr1 hr2 res1 res2 b1 b2,
    P p (SM _ _ m sgn) ->
    init_pc m i ->
    indist kobs p sgn (rt0 m sgn) (rt0 m sgn) b1 b2
      (i,(h1,r1)) (i,(h2,r2)) ->
    DEX_BigStepAnnot.DEX_BigStep p.(DEX_prog) m (i,(h1,r1)) (hr1,res1) ->
    DEX_BigStepAnnot.DEX_BigStep p.(DEX_prog) m (i,(h2,r2)) (hr2,res2) ->
    exists br1, exists br2,
      border b1 br1 /\ border b2 br2 /\
      hp_in kobs (DEX_ft p) br1 br2 hr1 hr2 /\
      indist_return_value kobs hr1 hr2 sgn res1 res2 br1 br2.
```

Chapter 6

# Type-Preserving Compilation of Android Bytecode

We have proposed a type system design to ensure non-interference on Android bytecode, and we also have shown that this type system is sound, i.e., a typable DEX bytecode is non-interferent. Now we prove type-preserving compilation for Android bytecode. We first analyze the translation process that is done in the actual dx tool and then show that the translation process preserves typing, i.e., typable JVM bytecode will yield typable DEX bytecode.

We decided to take this approach to leverage existing security approach to JVM bytecode due to the close relationship between JVM bytecode and DEX bytecode. It is also closer to our bigger goal where we provide a framework for the developer to provide a formal guarantee. We could target DEX bytecode directly, but then a failure to type DEX bytecode will not give any information whatsoever about what could cause the issue to the developer. With this approach, the typability of DEX bytecode depends on the typability of JVM bytecode, whose relationship has been studied (see Section 3.1 for the discussion).

## 6.1 Translation Phase

We now describe the translation process from JVM to DVM. This is an abstracted version of what is implemented in the dx tool of Android.

The dx tool translates JVM in blocks of code. To formalize this, it is useful to first define what we call a *Basic Block*. The Basic block is a construct containing a group of code that has one entry point and one exit point (not necessarily one successor/one parent), has a parent list, a successor list, a primary successor, and its order. The basic block also contains translated DEX instruction for the contained group of code. Formally, a basic block is a tuple

$$\{parents;\ succs;\ pSucc;\ order;\ JVM\_insn;\ DEX\_insn;\ handlers\}$$

where

- $parents \subseteq \mathcal{Z}$ is a set of the block's parents,

- $succs \subseteq \mathcal{Z}$ is a set of the block's successors,
- $pSucc \in \mathcal{Z}$ is the primary successor of the block (if the block does not have a primary successor it will have $-1$ as the value),
- $order \in \mathcal{Z}$ is the order of the block in the output phase, and
- $JVM\_insn \subseteq JVMins$ is the JVM instructions contained in the block
- $DEX\_insn \subseteq DEXins$ is the DEX instructions contained in the block (as a result of translating $JVM\_insn$).
- *handlers* is the associated exception handlers for the block.

The set of *BasicBlock* is denoted as *BasicBlocks*. When instantiating a basic block, we define a default object *NewBlock*, which will be a basic block with

$$\{parents = \varnothing;\ succs = \varnothing;\ pSucc = -1;\ order = -1;\\ JVM\_insn = \varnothing;\ DEX\_insn = \varnothing;\ handler = \varnothing\}$$

We also need some auxiliary functions to define the translation:

**BMap:** $\mathcal{PP} \rightarrow BasicBlock$ is a function from program pointers in JVM bytecode to a DEX basic block. Initially, the mapping is empty.

**PMap:** $\mathcal{PP} \rightarrow \mathcal{PP}$ is a function from program points in JVM to its starting point of the block that contains the program point (refer to Section 6.1.1). Initially, the mapping contains all of the program points mapped to itself ($\forall pp \in \mathcal{PP}, \mathbf{PMap}(pp) = pp$).

**SBMap:** $\mathcal{PP} \rightarrow boolean$ Similar to BMap, this function takes a program pointer in JVM bytecode and returns whether that instruction is the start of a basic block. Initially, the mapping contains all of the program points mapped to false ($\forall pp \in \mathcal{PP}, \mathbf{SBMap}(pp) = false$).

**TSMap:** $\mathcal{PP} \rightarrow \mathcal{Z}$ A function that maps a program pointer in JVM bytecode to an integer denoting the index to the top of the stack. This mapping is initialized with the number of local variables as that number is the index which will be used by DEX to simulate the stack ($\forall pp \in \mathcal{PP}, \mathbf{TSMap}(pp) = locN$ where $locN$ is the number of local variables).

**NewBlock:** *BasickBlock* A function which returns a *NewBlock*.

Since DEX is register-based whereas JVM is stack-based, to bridge this gap, the translation uses registers to simulate JVM stacks. This is done as follows:

- We set aside $l$ number of registers to hold local variables (registers $1, \ldots, l$). We denote these registers with $locR$;
- A stack of size $s$ is simulated by registers $l+1, \ldots, l+s$.

Note that although in principle, the stack can grow indefinitely, it is impossible to write a program that does so in Java, due to the strict stack discipline in Java. We assume the JVM bytecode has passed the Java bytecode verifier (see Lindholm et al. [2013] for the verifier's specifications), which ensures, among others, that the maximum height of the operand stack in a method is fixed. Similarly, bytecode verifier guarantees that the (Java) types of the operand stack (and by implication, also its height) at each program point is fixed. This makes it possible to statically map each operand stack location to a register in DEX. (cf. TSMap above and); see Davis et al. [2003] for a discussion on how this can be done.

There are several phases involved to translate JVM bytecode into DEX bytecode. To help illustrate each phase, we use the following idealized JVM bytecode with its abstracted labels:

| | | | | | |
|---|---|---|---|---|---|
| 0 | : | Load 1 | 5 | : | Push 1 |
| 1 | : | Goto 7 | 6 | : | Ireturn |
| 2 | : | Ifeq 5 | 7 | : | Load 2 |
| 3 | : | Push 0 | 8 | : | Sub |
| 4 | : | Ireturn | 9 | : | Goto 2 |

For this particular example, we also assume that the number of local variables is 2. After each phase, we will show the result of the transformation applied to above code. In general, we apply the transformations to $P$, the source JVM program that we want to translate. For this particular case, the program containing the piece of code is $P$.

### 6.1.1 Starting Instruction of a Block (StartBlock)

A program point is *a start of a block* (subsequently it will be called *the starting point of a block*) if it fulfills one of the following:

- It is the first instruction in a method;
- It is an instruction after a branching instruction (**ifeq**);
- it is the target of a branching instruction (**goto** and **ifeq**);

This step mainly indicates the program point at which the instruction starts a block, then creates a new block for each of these program points and associates it with a new empty block. Applying this step to our piece of code will result in program point $\{0, 2, 3, 5, 7\}$ (those displayed with checkmarks next to them below) indicated as the start of a block

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | : | Load 1 | ✓ | 5 | : | Push 1 | ✓ |
| 1 | : | Goto 7 | | 6 | : | Ireturn | |
| 2 | : | Ifeq 5 | ✓ | 7 | : | Load 2 | ✓ |
| 3 | : | Push 0 | ✓ | 8 | : | Sub | |
| 4 | : | Ireturn | | 9 | : | Goto 2 | |

and the resulting SBMap is:

$$\begin{array}{lllllllllll} \text{SBMap} = \{ & 0 & \mapsto & true, & 1 & \mapsto & false, & 2 & \mapsto & true, & 3 & \mapsto & true, \\ & 4 & \mapsto & false, & 5 & \mapsto & true, & 6 & \mapsto & false, & 7 & \mapsto & true, \\ & 8 & \mapsto & false, & 9 & \mapsto & false\} & & & & & & \end{array}$$

Technically speaking, this phase will update the mapping SBMap and BMap for those program points. We update these program points in SBMap to be true and associate empty blocks to those program points in BMap.

This phase is done by sweeping through the JVM instructions in the program (still in the form of a list). In the implementation, this phase will update the mapping SBMap. Apart from the first instruction, which will be the starting block regardless of the instruction, the instructions that become the start of a block have the characteristics that either they are a target of a branching instruction, or the previous instruction ends a block.

More concretely, case by case translation behavior is the following:

- $P[i]$ is Unconditional jump (goto $t$): the target instruction will be *the starting point of a block*. It is implicit in this instruction that the next instruction should also be the start of a block, but another jump will handle this. We do not take care of the case where no jump instruction addresses this next instruction (the next instruction is unreachable), i.e.
  - **BMap** $\oplus \{t \mapsto NewBlock\}$; and
  - **PMap** $\oplus \{t \mapsto t\}$
- $P[i]$ is Conditional jump (ifeq $t$): both the target instruction and the next instruction will be *the starting point of a block*, i.e.
  - **BMap** $\oplus \{t \mapsto NewBlock, (i+1) \mapsto NewBlock\}$; and
  - **PMap** $\oplus \{t \mapsto t, (i+1) \mapsto (i+1)\}$.
- $P[i]$ is Return: the next instruction will be *the starting point of a block*. This instruction will update the mapping of the next instruction for **BMap** and **SBMap** if this instruction is not at the end of the instruction list. The reason is that we already assumed that there is no dead code (no unreachable code), so the next instruction must be part of some execution path. To be more explicit, if there is a next instruction $i+1$ then
  - **BMap** $\oplus \{(i+1) \mapsto NewBlock\}$; and
  - **PMap** $\oplus \{(i+1) \mapsto (i+1)\}$
- $P[i]$ is an instruction which may throw an exception: just like return instruction, the next instruction will be *the starting point a new block*.
  - **BMap** $\oplus \{(i+1) \mapsto NewBlock\}$;
  - **PMap** $\oplus \{(i+1) \mapsto (i+1)\}$;

During this phase, there is also the setup for the additional block containing the sole instruction of **moveexception** which serves as an intermediary between the block with throwing instruction and its exception handler. Then for each associated exception handler, the following are indicated as starting of a block:

startPC program counter (pc) which serves as the starting point (inclusive) of which the exception handler is active;

* **BMap** $\oplus \{sPC \mapsto NewBlock\}$;
* **PMap** $\oplus \{sPC \mapsto sPC\}$;

endPC program counter which serves as the ending point (exclusive) of which the exception handler is active; and

* **BMap** $\oplus \{ePC \mapsto NewBlock\}$;
* **PMap** $\oplus \{ePC \mapsto ePC\}$;

handlerPC program counter which points to the start of the exception handler

* **BMap** $\oplus \{hPC \mapsto NewBlock\}$;
* **PMap** $\oplus \{hPC \mapsto hPC\}$;

Additionally, the intermediary block is also indicated as the starting of a block. For handler $h$, the intermediary block will have label $intPC = maxLabel + h.\textbf{handlerPC}$.

– **BMap** $\oplus \{intPC \mapsto NewBlock\}$;
– **PMap** $\oplus \{intPC \mapsto intPC\}$;

To reduce clutter, we write $sPC$ to stand for $h.\textbf{startPC}$, $ePC$ to stand for $h.\textbf{endPC}$, and $hPC$ to stand for $h.\textbf{handlerPC}$.

- $P[i]$ is any other instruction : no changes to **BMap** and **PMap**.

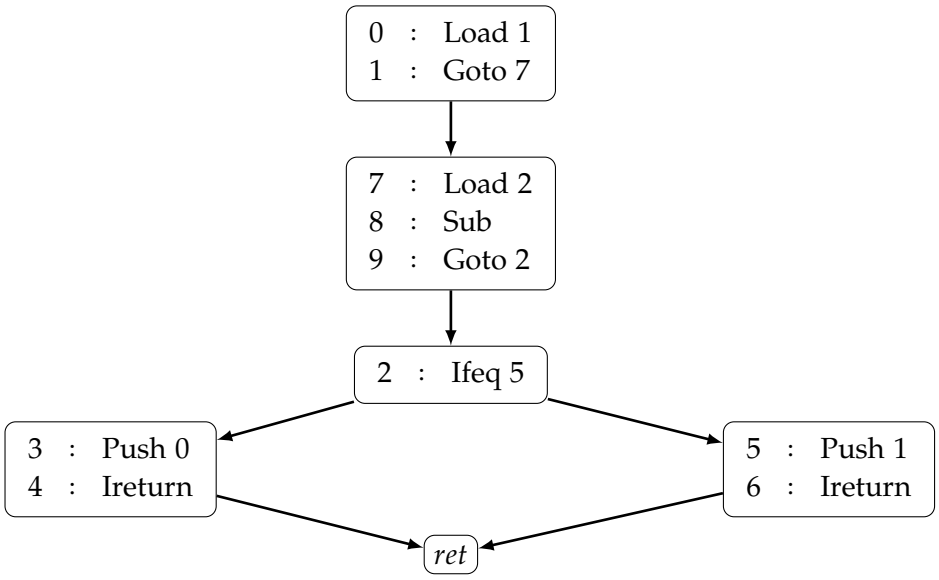
### 6.1.2 Resolving Parents-Successors Relationship (TraceParentChild)

In this step, we trace the relationship between blocks. For each block, resolve the parent blocks, successors blocks and its primary successor (if it exists). Implicit in this phase is a step creating a temporary return block used to hold successors of the block containing return instruction. At this point, we assume there is a special label called *ret* to address this temporary return block.

The creation of a temporary return block depends on whether the function returns a value. If it is return void, then this block contains only the instruction return-void. Otherwise depending on the type returned (integer, wide, object, etc.), the instruction is translated into the corresponding **move** and **return**. The **move** instruction moves the value from the register simulating the top of the stack to register 0 ($r_0$). Then **return** will just return $r_0$.

Mainly the mapping updated in this step is the BMap, although during this step we also update the TSMap. Intuitively, the mapped value will depend on the type

of instruction and the value for the current program point, e.g., if the value at the current program point is $n$, then the successor of the push instruction will have the value of $n+1$. Applying this step to the output of our previous phase will result in the following blocks and relations.




and the resulting TSMap is:

$$\begin{aligned}\text{TSMap} = \{\ & 0 \mapsto 3, \quad 5 \mapsto 3, \quad 1 \mapsto 4, \quad 6 \mapsto 4, \\ & 2 \mapsto 4, \quad 7 \mapsto 4, \quad 3 \mapsto 3, \quad 8 \mapsto 5, \\ & 4 \mapsto 4, \quad 9 \mapsto 4\}\end{aligned}$$

Before we mention the procedure to establish the parents-successors relationship, we need to introduce an additional function **getAvailableLabel**. Although defined clearly in the dx compiler itself, we will abstract away from the detail and assume the function produces a fresh label everytime it is called and labels for the additional intermediate block before exception handler blocks. These additional blocks before exception handler blocks are basically a block with a single instruction **moveexception** with the primary successor of the handler. Suppose the handler is at program point $i$, then this block will have a label of $maxLabel + i$ with the primary successor $i$. Furthermore, when a block has this particular handler as one of its successors, the successor index is pointed to $maxLabel + i$ (the block containing **moveexception** instead of $i$). In the sequel, whenever we say to add a handler to a block $b$, it means that adding this additional block as a successor of $b$, e.g. in the JVM bytecode, block $i$ has exception handlers at $j$ and $k$, so during translation block $i$ will have successors of $\{maxLabel + j, maxLabel + k\}$, block $j$ and $k$ will have additional parent of block $maxLabel + j$ and $maxLabel + k$, and they each will have block $i$ as their sole parent.

This phase is also done by sweeping through the JVM instructions but with the additional help of **BMap** and **PMap** mapping. More concretely, case by case translation behavior is the following:

- $P[i]$ is Unconditional jump (**goto** $t$): update the successors of the current block with the target branching, and the target block to have its parent list include the current block, i.e.
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{t\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = t$; and
  - $\mathbf{BMap}(t).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
- $P[i]$ is Conditional jump (**ifeq** $t$): since there will be two successors from this instruction, the current block will have two additional successor blocks, and both of the blocks will also update their list of parents to include the current block, i.e.,
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{i+1, t\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = i+1$;
  - $\mathbf{BMap}(i+1).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$; and
  - $\mathbf{BMap}(t).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
- $P[i]$ is **Return**: just add the return block as the current block's successors, and also update the parent of return block to include the current block, i.e.,
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{ret\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = ret$; and
  - $\mathbf{BMap}(ret).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
- $P[i]$ is one of the object manipulation instructions. The idea is that the next instruction will be the primary successor of this block, and should there be exception handler(s) associated with this block, they will be added as successors as well. We are making a little bit of a simplification here where we add the next instruction as the block's successor directly, i.e.,
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{i+1\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = i+1$;
  - $\mathbf{BMap}(i+1).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$; and
  - for each exception handler $j$ associated with $i$, let $intPC = maxLabel + j.\mathbf{handlerPC}$ and $hPC = j.\mathbf{handlerPC}$:
    - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{intPC\}$;
    - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{handlers} \cup \{j\}$;
    - $\mathbf{BMap}(intPC).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
    - $\mathbf{BMap}(intPC).\mathbf{succs} \cup \{hPC\}$
    - $\mathbf{BMap}(intPC).\mathbf{insn} = \{\mathbf{moveexception}\}$
    - $\mathbf{BMap}(hPC).\mathbf{parents} \cup \{intPC\}$

where $j$.**handlerPC** is the program point to the beginning of the exception handler block.

In the original dx tool, they add a new block to contain a pseudo instruction in between the current instruction and the next instruction, which will be removed during translation

- $P[i]$ is method invocation instruction. The treatment here is similar to that of object manipulation, where the next instruction is the primary successor, and the exception handler for this instruction is added as successors as well. The difference lies in that where the additional block is bypassed in object manipulation instruction, this time we really add a block with an instruction **moveresult** (if the method is returning a value) with a fresh label $l$ = **getAvailableLabel** and the sole successor of $i+1$. The current block will then have $l$ as its primary successor, and the next instruction ($i+1$) will have $l$ added to its list of parents, i.e.,
  - $l$ = **getAvailableLabel**;
  - **BMap**(**PMap**($i$)).**succs** $\cup \{l\}$;
  - **BMap**(**PMap**($i$)).**pSucc** = $l$;
  - **BMap**(**PMap**($i$)).**parents** = $\{i\}$;
  - **BMap** $\oplus \{l \mapsto NewBlock\}$;
  - **BMap**($l$).**succs** = $\{i+1\}$;
  - **BMap**($l$).**pSucc** = $(i+1)$;
  - **BMap**($l$).**insn** = {**moveresult**}
  - **BMap**($i+1$).**parents** $\cup \{l\}$; and
  - for each exception handler $j$ associated with $i$, let $intPC$ = $maxLabel$ + $j$.**handlerPC** and $hPC$ = $j$.**handlerPC**:
    - **BMap**(**PMap**($i$)).**succs** $\cup \{intPC\}$;
    - **BMap**(**PMap**($i$)).**handlers** $\cup \{j\}$;
    - **BMap**($intPC$).**parents** $\cup$ {**PMap**($i$)}
    - **BMap**($intPC$).**succs** $\cup \{hPC\}$
    - **BMap**($intPC$).**insn** = {**moveexception**}
    - **BMap**($hPC$).**parents** $\cup \{intPC\}$
- $P[i]$ is throw instruction. This instruction only adds the exception handlers to the block without updating other block's relationship, i.e., if the current block is $i$, then for each exception handler $j$ associated with $i$, let $intPC$ = $maxLabel$ + $j$.**handlerPC** and $hPC$ = $j$.**handlerPC**:
  - **BMap**(**PMap**($i$)).**succs** $\cup \{intPC\}$;
  - **BMap**(**PMap**($i$)).**handlers** $\cup \{j\}$;

- $\mathbf{BMap}(intPC).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
- $\mathbf{BMap}(intPC).\mathbf{succs} \cup \{hPC\}$
- $\mathbf{BMap}(intPC).\mathbf{insn} = \{\mathbf{moveexception}\}$
- $\mathbf{BMap}(hPC).\mathbf{parents} \cup \{intPC\}$

- $P[i]$ is any other instruction: depending on whether the next instruction is a start of a block or not.

  - If the next instruction is a start of a block, then update the successor of the current block to include the block of the next instruction and the parent of the block of the next instruction to include the current block i.e.
    - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{i+1\}$; and
    - $\mathbf{BMap}(i+1).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
  - If the next instruction is not a start of a block, then just point the next instruction to have the same pointer as the current block, i.e., $\mathbf{PMap}(i+1) = \mathbf{PMap}(i)$

### 6.1.3 Reading Java Bytecodes (Translate)

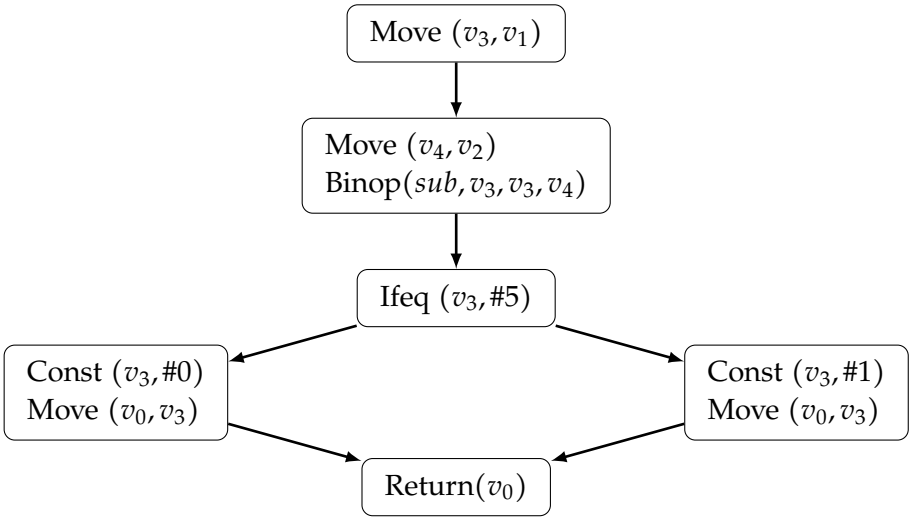
Table 6.1 list the resulting DEX translation for each of the JVM bytecode instruction listed in Section 3.3. The full translation scheme with their typing rules can be seen in Table 6.2. A note about these instructions is that during this parsing of JVM bytecodes, the dx translation will also modify the top of the stack for the next instruction. Since the dx translation only happens for verified JVM bytecode, we can safely assume that these top of the stacks will be consistent (even though an instruction may have many parents, the resulting top of the stack from the parent instruction will be consistent with each other). To improve readability, we abuse the notation $r(x)$ to also mean $r_x$.

There is a note about the translation scheme, in particular, the goto instruction. Although we do have goto instruction in DEX, it is not a direct translation from goto in JVM bytecode, but will be added in the last step of compiling. The goto from JVM bytecode is only useful to mark the relationship between blocks, and then it is removed from the further compilation process.

Applying the translation scheme to the output of the previous step, we will get:

| Translation | | | Side effect |
|---|---|---|---|
| $[\![\textbf{push}]\!]$ | = | $\textbf{const}(r(\textbf{TS}_i), n)$ | $\textbf{TS}(i+1) = \textbf{TS}(i)+1$ |
| $[\![\textbf{pop}]\!]$ | = | $\varnothing$ | $\textbf{TS}(i+1) = \textbf{TS}(i)-1$ |
| $[\![\textbf{load } x]\!]$ | = | $\textbf{move}(r(\textbf{TS}_i), r_x)$ | $\textbf{TS}(i+1) = \textbf{TS}(i)+1$ |
| $[\![\textbf{store } x]\!]$ | = | $\textbf{move}(r_x, r(\textbf{TS}_i-1))$ | $\textbf{TS}(i+1) = \textbf{TS}(i)-1$ |
| $[\![\textbf{binop } op]\!]$ | = | $\textbf{binop}(op, r(\textbf{TS}_i-2), r(\textbf{TS}_i-2),$ $r(\textbf{TS}_i-1))$ | $\textbf{TS}(i+1) = \textbf{TS}(i)-1$ |
| $[\![\textbf{swap}]\!]$ | = | $\textbf{move}(r(\textbf{TS}_i), r(\textbf{TS}_i-2))$ $\textbf{move}(r(\textbf{TS}_i+1), r(\textbf{TS}_i-2))$ $\textbf{move}(r(\textbf{TS}_i-1), r(\textbf{TS}_i+1))$ $\textbf{move}(r(\textbf{TS}_i-2), r(\textbf{TS}_i))$ | $\textbf{TS}(i+1) = \textbf{TS}(i)$ |
| $[\![\textbf{goto } t]\!]$ | = | $\varnothing$ | $\textbf{TS}(t) = \textbf{TS}(i)$ |
| $[\![\textbf{ifeq } t]\!]$ | = | $\textbf{ifeq}(r(\textbf{TS}_i-1), t)$ | $\textbf{TS}(i+1) = \textbf{TS}(i)-1$ $\textbf{TS}(t) = \textbf{TS}(i)-1$ |
| $[\![\textbf{return}]\!]$ | = | $\textbf{move}(r_0, r(\textbf{TS}_i-1))$ $\textbf{return}(r_0)$ or $\textbf{goto}(ret)$ | |
| $[\![\textbf{new } C]\!]$ | = | $\textbf{new}(r(\textbf{TS}_i-1), C)$ | $\textbf{TS}(i+1) = \textbf{TS}(i)+1$ |
| $[\![\textbf{getfield } f]\!]$ | = | $\textbf{iget}(r(\textbf{TS}_i-1), r(\textbf{TS}_i-1), f)$ | $\textbf{TS}(i+1) = \textbf{TS}(i)+1$ |
| $[\![\textbf{putfield } f]\!]$ | = | $\textbf{iput}(r(\textbf{TS}_i-1), r(\textbf{TS}_i-2), f)$ | $\textbf{TS}(i+1) = \textbf{TS}(i)-2$ |
| $[\![\textbf{newarray } t]\!]$ | = | $\textbf{newarray}(r(\textbf{TS}_i-1), r(\textbf{TS}_i-1), t)$ | $\textbf{TS}(i+1) = \textbf{TS}(i)$ |
| $[\![\textbf{arraylength}]\!]$ | = | $\textbf{arraylength}(r(\textbf{TS}_i-1), r(\textbf{TS}_i-1))$ | $\textbf{TS}(i+1) = \textbf{TS}(i)$ |
| $[\![\textbf{arrayload}]\!]$ | = | $\textbf{aget}(r(\textbf{TS}_i-2), r(\textbf{TS}_i-2), r(\textbf{TS}_i-1))$ | $\textbf{TS}(i+1) = \textbf{TS}(i)-1$ |
| $[\![\textbf{arraystore}]\!]$ | = | $\textbf{aput}(r(\textbf{TS}_i-1), r(\textbf{TS}_i-3), r(\textbf{TS}_i-2))$ | $\textbf{TS}(i+1) = \textbf{TS}(i)-3$ |
| $[\![\textbf{invoke } m]\!]$ | = | $\textbf{invoke}(n, m, \vec{p})$ | $l = \textbf{getAvailableLabel}$ |
| | | $\textbf{moveresult}(r(\textbf{TS}_i-n))$ at block $l$ | $\textbf{TS}(i+1) = \textbf{TS}(i)-n$ |
| $[\![\textbf{throw}]\!]$ | = | $\textbf{throw}(r(\textbf{TS}_i-1))$ | |

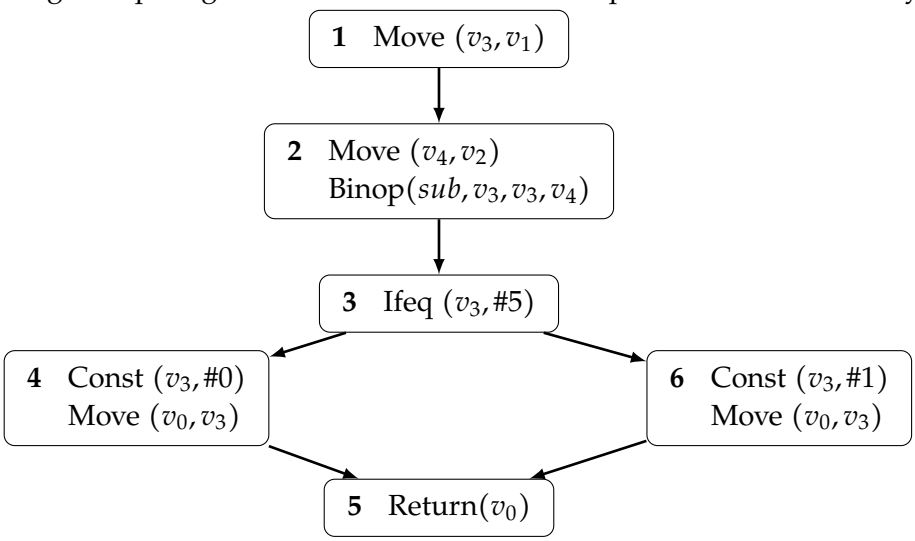
Table 6.1: Instruction Translation Table




### 6.1.4 Ordering Blocks (PickOrder)

The next phase in the translation scheme is order blocks according to "trace analysis". The "trace analysis" itself is quite simple in essence, that is for each block we assign an integer denoting the order of appearance of that particular block in the analysis. Starting from the initial block, we first assign an order to the selected block. Then we pick the first unordered successor, giving priority to its primary successor (if any). The tracing continues until there is no more successor. The selected successor will not form a loop, i.e., the analysis will not pick a block that has already been given an order.

After we reached one end, we pick an unordered block and do the trace analysis again. However, this time we trace its source ancestor first, by tracing an unordered parent block and stop when there is no more unordered parent block or already forming a loop. Algorithm 4 describes how we implement this trace analysis.




and an integer denoting the current order $o$ starting from 1.

**Algorithm 4** PickOrder($blocks$)

```
order := 0;
while there is still block x ∈ blocks without order; do
  var := PickStartingPoint(x, {x});
  order = TraceSuccessors(source, order);
return order;
```

- **Pick Starting Point (Algorithm 5)**
  This function is a recursive function with an auxiliary data structure to prevent ancestor loop from the viewpoint of block $x$. On each recursion, we pick a parent $p$ from $x$ which primary successor is $x$, not yet ordered, and not yet in the loop. The function then return **PickStartingPoint**($p$).

**Algorithm 5** PickStartingPoint($x, loop$)

```
for all p ∈ BMap(x).parents do
  if p ∈ loop then return x;
  bp = BMap(p);
  if bp.pSucc = x and bp.order = −1 then
    loop = loop ∪ {p};
    return PickStartingPoint(p, loop)
return order;
```

- **Trace Successors (Algorithm 6)**
  This function is also a recursive function with an argument of block $x$. It starts by assigning the current order $o$ to $x$ then increment $o$ by 1. Then it does recursive call to **TraceSuccessors** giving one successor of $x$ which is not yet ordered as the argument (giving priority to the primary successor of $x$ if there is one).

**Algorithm 6** TraceSuccessors($x, order$)

```
BMap(x).order = order;
if BMap(x).psucc ≠ −1 then
  pSucc = BMap(x).pSucc;
  if BMap(pSucc).order = −1 then return TraceSuccessors(pSucc, order + 1);
  for all s ∈ BMap(x).succs do
    if BMap(pSucc).order = −1 then return TraceSuccessors(s, order + 1);
return order;
```

### 6.1.5 Output DEX Instructions (Output)

Since the translation phase already translated the JVM instruction and ordered the block, this phase basically just outputs the instructions in order of the block. Nevertheless, there is some housekeeping to do alongside producing output of instructions.

- Remember the program counter for the first instruction in the block within DEX program. This is mainly useful for fixing up the branching target later on.

- Add gotos to the successor when needed for each block that is not ending in branch instruction like **goto** or if. The main reason to do this is to maintain the successor relation in the case where the next block in order is not the expected block. More specifically, this is step here is in order to satisfy the property 6.2.8. There is a special case for branching instruction (ifeq). If the next block to output is, in fact, the target of branching instead of its primary successor, then the branching instruction will be replaced by its opposite (ifneq).

- Instantiate the return block.

- Reading the list of DEX instructions and fix up the target of jump instructions.

- Collecting information about exception handlers. It is done by sweeping through the block in ordered fashion, inspecting the exception handlers associated with each block. We assume that the variable $DEXHandler$ is a global variable that stores the information about exception handler in the DEX bytecode. The function **newHandler**$(cS, cE, hPC, t)$ will create a new handler (for DEX) with $cS$ as the start PC, $cE$ as the end PC, $hPC$ as the handler PC, and $t$ as the type of exception caught by this new handler.

**Algorithm 7** makeHandlerEntry$(cH, cS, cE)$

**for all** handler $h \in cH$ **do**
  $hPC = h.handlerPC$;
  $t = h.catchType$;
  $DEXHandler = DEXHandler + newHandler(cS, cE, hPC, t)$;

The only information that is needed to produce the information about exception handlers in DEX is the basic blocks contained in **BMap**. The procedure **translateExceptionHandlers** (Algorithm 8) take these basic blocks *blocks* and make use the procedure **makeHandlerEntry** to create the exception handlers in DEX.

A note about the last make entry is that the algorithm will leave one set of handlers hanging at the end of loop. Therefore, we need to make that set of handlers into entry in the DEX exception handlers.

**Algorithm 8** translateExceptionHandlers(*blocks*)

```
cH = ∅; // current handler
cS // current start PC
cE // current start PC
for all block x in order do
  if x.handlers is not empty then
    if cH = x.handlers; then
      cE = x.endPC;
    else if cH ≠ x.handlers then
      makeHandlerEntry(cH, cS, cE);
      cS = x.startPC;
      cE = x.endPC;
      cH = x.handlers;
makeHandlerEntry(cH, cS, cE);
```

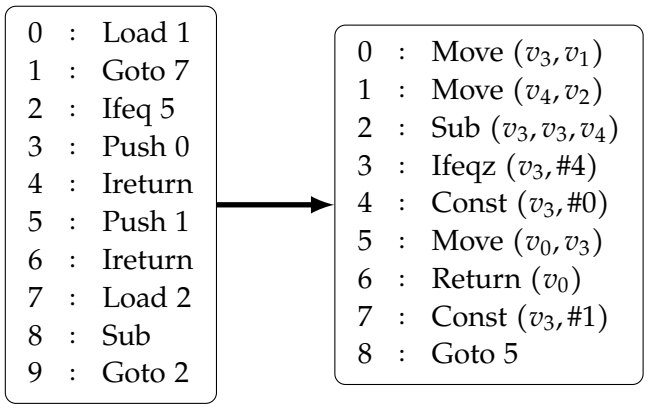
**Algorithm 9** output

```
blocks = ordered blocks ∈ BMap;
lbl = ∅; // label mapping
out = ∅; // list of DEX output
pc = 0; // DEX program counter
for all block x in order do
  next = next block in order;
  lbl[x] = pc;
  pc = pc + x.insn.length;
  out = out + x.insn;
  if p.pSucc ≠ next then
    if x.insn.last is ifeq then
      t = x.insn.last.target;
      if t = next then
        out.last = oppositeCondition(x.insn.last);
      else
        out = out + goto(next);
    else
      out = out + goto(next);
for all index i in out do
  if out[i] is a jump instruction then
    out[i].target = lbl[out[i].target];
translateExceptionHandlers(blocks);
```

For simplicity, we overload the length of instructions list to also mean the total length of instructions contained in the list. The operator + here is also taken to mean list append operation. The function **oppositeCondition** takes an **ifeq**$(r, t)$ and returns its opposite **ifneq**$(r, t)$. Finally, we assume that the target of jump instruction can be accessed using the field **target**, e.g., **ifeq**$(r, t)$.**target** = $t$. The detail of the steps

in this phase is contained in Algorithm 9.

The final result of applying all the steps to our example piece of code:




**Definition 6.1.1** (Translated JVM Program)**.** *The translation of a JVM program $P$ into blocks and have their JVM instructions translated into DEX instructions is denoted by $\lfloor\!\lfloor P \rfloor\!\rfloor$, where*

$$\lfloor\!\lfloor P \rfloor\!\rfloor = \textbf{Translate}(\textbf{TraceParentChild}(\textbf{StartBlock}(P))).$$

**Definition 6.1.2** (Output Translated Program)**.** *The output of the translated JVM program $\lfloor\!\lfloor P \rfloor\!\rfloor$ in which the blocks are ordered and then output into DEX program is denoted by $\lceil\!\lceil \lfloor\!\lfloor P \rfloor\!\rfloor \rceil\!\rceil$, where*

$$\lceil\!\lceil \lfloor\!\lfloor P \rfloor\!\rfloor \rceil\!\rceil = \textbf{Output}(\textbf{PickOrder}(\lfloor\!\lfloor P \rfloor\!\rfloor)).$$

**Definition 6.1.3** (Compiled JVM Program)**.** *The compilation of a JVM program $P$ is denoted by $[\![P]\!]$, where $[\![P]\!] = \lceil\!\lceil \lfloor\!\lfloor P \rfloor\!\rfloor \rceil\!\rceil$.*

The full translation scheme from JVM to DEX can be seen in table 6.2.

## 6.2 Proof that Translation Preserves Typability

### 6.2.1 Compilation of CDR and Security Environments

Since now we will be working on blocks, we need to know how the CDRs of the JVM and that of the translated DEX are related. First, we need to define the definition of the successor relation between blocks.

**Definition 6.2.1** (Block Successor)**.** *Suppose $a \mapsto b$ and $a$ and $b$ are on different blocks. Let $B_a$ be the block containing $a$ and $B_b$ be the block containing $b$. Then $B_b$ will be the successor of $B_a$ denoted by abusing the notation $B_a \mapsto B_b$.*

Before we continue on with the properties of CDR and SOAP, we first need to define the properties of the translation of **region** and **jun** since we assume that the JVM bytecode comes equipped with **region** and **jun**.

| Original Transfer Rule | Related DEX Transfer Rule |
|---|---|
| $\frac{P[i] = \textbf{Push}v}{se, i \vdash^{\text{Norm}} st \Rightarrow se(i) :: st}$ | $\frac{P[i] = \textbf{Const}(r, n)}{se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(i)\}}$ |
| $\frac{P[i] = \textbf{Pop}}{i \vdash st \Rightarrow st}$ | None |
| $\frac{P[i] = \textbf{Load}\ x}{se, i \vdash^{\text{Norm}} st \Rightarrow \big(se(i) \sqcup \vec{k_a}(x)\big) :: st}$ | $\frac{P[i] = \textbf{Move}(r, r_s)}{se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto \big(se(i) \sqcup rt(r_s)\big)\}}$ *) $x$ is related to $r_s$, e.g. $x = 5$ becomes $r_s = r_5$s |
| $\frac{P[i] = \textbf{Store}\ x \quad k \sqcup se(i) \leq \vec{k_a}(x)}{\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash^{\text{Norm}} k :: st \Rightarrow st}$ | $\frac{P[i] = \textbf{Move}(r, r_s)}{se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(i) \sqcup rt(r_s)\}}$ *) $x$ is related to $r$, e.g. $x = 5$ becomes $r = r_5$s |
| $\frac{P[i] = \textbf{Binop}}{se, i \vdash^{\text{Norm}} a :: b :: st \Rightarrow \big(se(i) \sqcup a \sqcup b\big) :: st}$ | $\frac{P[i] = \textbf{Binop}(r, r_a, r_b)}{se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto \big(se(i) \sqcup rt(r_a) \sqcup rt(r_b)\big)\}}$ |
| $\frac{P[i] = \textbf{Swap}}{i \vdash^{\text{Norm}} k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st}$ | $\frac{P[i] = \textbf{Move}(r, r_s)}{se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto \big(se(i) \sqcup rt(r_s)\big)\}}$ |
| $\frac{P[i] = \text{Goto}\ t}{i \vdash st \Rightarrow st}$ | $\frac{P[i] = \text{Goto}\ t}{i \vdash rt \Rightarrow rt}$ *) Not directly translated |
| $\frac{P[i] = \textbf{ifeq}t \quad \forall j' \in \textbf{region}(i), k \leq se(j')}{\textbf{reigon}, se, i \vdash^{\text{Norm}} k :: st \Rightarrow \text{lift}_k(st)}$ Ifeq may be translated into Ifneq on certain condition | $\frac{P[i] = \textbf{ifeq}(r, t) \quad \forall j' \in \text{region}(i), se(i) \sqcup rt(r) \leq se(j')}{\textbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt}$ $\frac{P[i] = \textbf{ifneq}(r, t) \quad \forall j' \in \text{region}(i), se(i) \sqcup rt(r) \leq se(j')}{\textbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt}$ |
| $\frac{P[i] = \textbf{return} \quad se(i) \sqcup k \leq k_r}{\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash k :: st \Rightarrow}$ | $\frac{P[i] = \textbf{Move}(r_0, r_s)}{se, i \vdash rt \Rightarrow rt \oplus \{r \mapsto \big(se(i) \sqcup rt(r_s)\big)\}}$ **and** $\frac{P[i] = \textbf{goto}(t)}{i \vdash rt \Rightarrow rt}$ **or** $\frac{P[i] = \textbf{return}(r_s)\ se(i) \sqcup rt(r_s) \leq k_r}{\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash rt \Rightarrow}$ |
| $\frac{P[i] = \textbf{new}C}{se, i \vdash^{\text{Norm}} st \Rightarrow se(i) :: st}$ | $\frac{P[i] = \textbf{new}(r, c)}{se, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(i)\}}$ |

| **Original Typing Rule** | **Related DEX Typing Rule** |
|---|---|
| $$\frac{P[i] = \mathbf{getfield} f \quad k \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \leq se(j)}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow \mathbf{lift_k}((k \sqcup \mathbf{ft}(f) \sqcup se(i)) :: st)}$$ | $$\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(r_o) \leq se(j)}{\mathbf{ft}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto rt(r_o) \sqcup \mathbf{ft}(f) \sqcup se(i)\}}$$ |
| $$\frac{P[i] = \mathbf{getfield} f \quad k \in \mathcal{S} \quad \mathbf{Handler}(i, \mathrm{np}) = t \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \leq se(j)}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$$ | $$\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{ft}, se, i \vdash^{\mathrm{np}} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto rt(r_o) \sqcup se(i)\}}$$ |
| $$\frac{P[i] = \mathbf{getfield} f \quad k \in \mathcal{S} \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \leq se(j) \quad k \leq \vec{k_r}(\mathrm{np})}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k :: st \Rightarrow}$$ | $$\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad se(i) \sqcup rt(r_o) \leq \vec{k_r}(\mathrm{np}) \quad \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, se, i \vdash^{\mathrm{np}} rt \Rightarrow}$$ |
| $$\frac{P[i] = \mathbf{putfield} f \quad k_h \leq \mathbf{ft}(f) \quad k_1 \sqcup se(i) \sqcup k_2 \leq \mathbf{ft}(f) \quad k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k_2 \leq se(j)}{\mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} k_1 :: k_2 :: st \Rightarrow \mathbf{lift_{k_2}}(st)}$$ | $$\frac{P[i] = \mathbf{iput}(r_s, r_o, f) \quad k_h \leq \mathbf{ft}(f) \quad rt(r_o) \in \mathcal{S} \quad rt(r_o) \sqcup se(i) \sqcup rt(r_s) \leq \mathbf{ft}(f) \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(r_o) \leq se(j)}{\mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt}$$ |
| $$\frac{P[i] = \mathbf{putfield} f \quad k_1 \sqcup se(i) \sqcup k_2 \leq \mathbf{ft}(f) \quad k_1 \in \mathcal{S}^{\mathrm{ext}} \quad \mathbf{Handler}(i, \mathrm{np}) = t \quad k_2 \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k_2 \leq se(j)}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$$ | $$\frac{P[i] = \mathbf{iput}(r_s, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad rt(r_o) \sqcup se(i) \sqcup rt(r_s) \leq \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) = t \quad \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \leq se(j)}{\mathbf{ft}, se, i \vdash^{\mathrm{np}} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto rt(r_o) \sqcup se(i)\}}$$ |
| $$\frac{P[i] = \mathbf{putfield} f \quad k_1 \sqcup se(i) \sqcup k_2 \leq \mathbf{ft}(f) \quad k_1 \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k_2 \leq se(j) \quad k_2 \in \mathcal{S} \quad k_2 \leq \vec{k_r}(\mathrm{np}) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow}{\mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: st \Rightarrow}$$ | $$\frac{P[i] = \mathbf{iput}(r_s, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad rt(r_o) \sqcup se(i) \sqcup rt(r_s) \leq \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \leq se(j) \quad se(i) \sqcup rt(r_o) \leq \vec{k_r}(\mathrm{np})}{\mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, se, i \vdash^{\mathrm{np}} rt \Rightarrow}$$ |
| $$\frac{P[i] = \mathbf{newarray} t \quad k \in \mathcal{S}}{i \vdash^{\mathrm{Norm}} k :: st \Rightarrow k[\mathbf{at}(i)] :: st}$$ | $$\frac{P[i] = \mathbf{newarray}(r, r_l, t) \quad rt(r_l) \in \mathcal{S}}{i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto rt(r_l)[\mathbf{at}(i)]\}}$$ |
| $$\frac{P[i] = \mathbf{arraylength} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \leq se(j) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}}}{\mathbf{region}, se, i \vdash^{\mathrm{Norm}} k[k_c] :: st \Rightarrow \mathbf{lift_k}(k :: st)}$$ | $$\frac{P[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \leq se(j)}{\mathbf{region}, se, i \vdash^{norm} rt \Rightarrow rt \oplus \{r \mapsto k\}}$$ |
| $$\frac{P[i] = \mathbf{arraylength} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \leq se(j) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{region}, se, i \vdash^{\mathrm{np}} k[k_c] :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$$ | $$\frac{P[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{region}, se, i \vdash^{np} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}}$$ |
| $$\frac{P[i] = \mathbf{arraylength} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \leq se(j) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad k \leq \vec{k_r}(\mathrm{np})}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{np}} k[k_c] :: st \Rightarrow}$$ | $$\frac{P[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad se(i) \sqcup k \leq \vec{k_a}[\mathrm{np}]}{\vec{k_a} \to \vec{k_r}, \mathbf{region}, se, i \vdash^{np} rt \Rightarrow}$$ |

| Original Typing Rule | Related DEX Typing Rule |
|---|---|
| $\dfrac{P[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{Norm}) k_2 \leq se(j)}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{Norm}} k_1 :: k_2[k_c] :: st \Rightarrow \textbf{lift}_{\mathbf{k_2}}(((k_1 \sqcup k_2) \sqcup^{\text{ext}} k_c) :: st)}$ | $\dfrac{P[i] = \textbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k_c \in \mathcal{S}^{\text{ext}} \quad k, rt(r_i) \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \text{Norm}), k \leq se(j)}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{norm} rt \Rightarrow rt \oplus \{r \mapsto ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} k_c)\}}$ |
| $\dfrac{P[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}) k_2 \leq se(j) \quad \textbf{Handler}(i, \text{np}) = t}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2[k_c] :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$ | $\dfrac{P[i] = \textbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k_c \in \mathcal{S}^{\text{ext}} \quad k, rt(r_i) \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \text{np}), k \leq se(j) \quad \textbf{Handler}(i, \text{np}) = t}{\textbf{region}, se, i \vdash^{np} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}}$ |
| $\dfrac{P[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad k_2 \leq \vec{k_r}(\text{np}) \quad \forall j \in \textbf{region}(i, \text{np}) k_2 \leq se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2[k_c] :: st \Rightarrow}$ | $\dfrac{P[i] = \textbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k_c \in \mathcal{S}^{\text{ext}} \quad k, rt(r_i) \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \text{np}), k \leq se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow \quad se(i) \sqcup k \leq \vec{k_r}(\text{np})}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{np} rt \Rightarrow}$ |
| $\dfrac{P[i] = \textbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \quad k_2, k_3 \in \mathcal{S} \quad ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \leq^{\text{ext}} k_c \quad \forall j \in \textbf{region}(i, \text{Norm}), k_2 \leq se(j)}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{Norm}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow \textbf{lift}_{\mathbf{k_2}}(st)}$ | $\dfrac{P[i] = \textbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{Norm}), k \leq se(i)}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt}$ |
| $\dfrac{P[i] = \textbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \quad ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \leq^{\text{ext}} k_c \quad k_2, k_3 \in \mathcal{S} \quad \textbf{Handler}(i, \text{np}) = t \quad \forall j \in \textbf{region}(i, \text{np}), k_2 \leq se(j)}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$ | $\dfrac{P[i] = \textbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k \leq se(i) \quad \textbf{Handler}(i, \text{np}) = t}{\textbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}}$ |
| $\dfrac{P[i] = \textbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \quad k_2, k_3 \in \mathcal{S} \quad ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \leq^{\text{ext}} k_c \quad \forall j \in \textbf{region}(i, \text{np}), k_2 \leq se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow \quad k_2 \leq \vec{k_r}(\text{np})}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow}$ | $\dfrac{P[i] = \textbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k \leq se(i) \quad \textbf{Handler}(i, \text{np}) = t \quad se(i) \sqcup k \leq \vec{k_r}(\text{np})}{\vec{k_a} \to \vec{k_r}\textbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow}$ |
| $\dfrac{P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad \textbf{length}(st_1) = \text{nbArguments}(m_I D) \quad k \leq \vec{k'_a}[0] \quad \forall i \in [0, \textbf{length}(st_1) - 1] st_1[i] \leq \vec{k'_a}[i+1] \quad k \sqcup k_h \sqcup se(i) \leq k'_h \quad k_e = \bigsqcup\{\vec{k'_r}(e) \mid e \in \textbf{excAnalysis}(m_{\text{ID}})\} \quad \Gamma_{m_{ID}}[k] = \vec{k'_a} \xrightarrow{k'_h} k'_r \quad \forall j \in \textbf{region}(i, \text{Norm}), k \sqcup k_e \leq se(j)}{\Gamma, \textbf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\text{Norm}} st_1 :: k :: st_2 \Rightarrow \textbf{lift}_{\mathbf{k \sqcup k_e}}((k'_r \sqcup k \sqcup se(i)) :: st_2)}$ | $\dfrac{P_m[i] = \textbf{invoke}(n, m, \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \xrightarrow{k'_h} k'_r \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall_{0 \leq j < n} rt(\vec{p}[j]) \leq \vec{k'_a}[j] \quad k_e = \bigsqcup\{\vec{k'_r}(e) \mid e \in \textbf{excAnalysis}(m')\} \quad \forall j \in \textbf{region}(i, \text{Norm}), rt(\vec{p}[0]) \sqcup k_e \leq se(j)}{\Gamma, \textbf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{ret \mapsto k'_r[n] \sqcup rt(\vec{p}[0]) \sqcup se(i)\}}$ <br> $\dfrac{P_m[i] = \textbf{moveresult}(r)}{i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto rt(ret)\}}$ |

| **Original Typing Rule** | **Related DEX Typing Rule** |
|---|---|
| $\dfrac{P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad k \le \vec{k'_a}[0] \quad \textbf{length}(st_1) = \text{nbArguments}(m_I D) \quad \forall i \in [0, \textbf{length}(st_1) - 1] st_1[i] \le \vec{k'_a}[i+1] \quad k \sqcup k_h \sqcup se(i) \le k'_h \quad e \in \textbf{excAnalysis}(m_{\text{ID}}) \cup \{\text{np}\} \quad \Gamma_{m_{ID}}[k] = \vec{k'_a} \xrightarrow{k'_h} k'_r \quad \forall j \in \textbf{region}(i,e), k \sqcup k_e \le se(j) \quad \textbf{Handler}(i,e) = t}{\Gamma, \textbf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^e st_1 :: k :: st_2 \Rightarrow (k \sqcup \vec{k'_r}(e)) :: \epsilon}$ | $\dfrac{P_m[i] = \textbf{invoke}(n,m,\vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \xrightarrow{k'_h} k'_r \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \le k'_h \quad \forall_{0 \le j < n} rt(\vec{p}[j]) \le \vec{k'_a}[j] \quad e \in \textbf{excAnalysis}(m') \cup \{\text{np}\}\} \quad \textbf{Handler}(i,e) = t \quad \forall j \in \textbf{region}(i,e), rt(\vec{p}[0]) \sqcup \vec{k'_r}(e) \le se(j)}{\Gamma, \textbf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash^e rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto k'_r[e] \sqcup rt(\vec{p}[0])\}}$ |
| $\dfrac{P_m[i] = \textbf{invoke}\ m_{\text{ID}} \quad \textbf{length}(st_1) = \text{nbArguments}(m_I D) \quad k \le \vec{k'_a}[0] \quad \forall i \in [0, \textbf{length}(st_1) - 1] st_1[i] \le \vec{k'_a}[i+1] \quad k \sqcup k_h \sqcup se(i) \le k'_h \quad e \in \textbf{excAnalysis}(m_{\text{ID}}) \cup \{\text{np}\} \quad \Gamma_{m_{ID}}[k] = \vec{k'_a} \xrightarrow{k'_h} k'_r \quad \forall j \in \textbf{region}(i,e), k \sqcup k_e \le se(j) \quad \textbf{Handler}(i,e) \uparrow \quad k \sqcup se(i) \sqcup \vec{k'_r}(e) \le \vec{k_r}(e)}{\Gamma, \textbf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^e st_1 :: k :: st_2 \Rightarrow}$ | $\dfrac{P_m[i] = \textbf{invoke}(n,m,\vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \xrightarrow{k'_h} k'_r \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \le k'_h \quad \forall_{0 \le j < n} rt(\vec{p}[j]) \le \vec{k'_a}[j] \quad e \in \textbf{excAnalysis}(m') \cup \{\text{np}\}\} \quad \textbf{Handler}(i,e) \uparrow \quad \forall j \in \textbf{region}(i,e), rt(\vec{p}[0]) \sqcup k'_r[e] \le se(j) \quad rt(\vec{p}[0]) \sqcup se(i) \sqcup \vec{k'_r}(e) \le \vec{k_r}(e)}{\Gamma, \textbf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash^e rt \Rightarrow}$ |

Table 6.2: Translation Table

**Property 6.2.1** (Region Translation and Compilation)**.** Given a JVM $\textbf{region}(i)$ and $P[i]$ is a branching instruction, let $i_b$ be the program point in $\llbracket i \rrbracket$ such that $P_{\text{DEX}}[i_b]$ is a branching instruction, then

$$\llbracket \textbf{region}(i) \rrbracket = \textbf{region}(i_b) = \bigcup_{j \in \textbf{region}(i)} \llbracket j \rrbracket \text{ and } \llbracket\!\llbracket \textbf{region}(i) \rrbracket\!\rrbracket = \textbf{region}(i_b) = \bigcup_{j \in \textbf{region}(i)} \llbracket\!\llbracket j \rrbracket\!\rrbracket$$

**Property 6.2.2** (Region for appended **goto** instruction)**.**

$$\begin{aligned} \forall b \in \llbracket P \rrbracket. \quad & P_{\text{DEX}}[\lceil b.\textbf{lastAddress} \rceil + 1] = \textbf{goto} \\ & \rightarrow \big( \forall .i \in \mathcal{PP}_{\text{DEX}}.b.\textbf{lastAddress} \in \textbf{region}(i) \\ & \rightarrow (\lceil b.\textbf{lastAddress} \rceil + 1) \in \textbf{region}(i) \big) \end{aligned}$$

where $\rightarrow$ indicates logical implication.

**Property 6.2.3** (Junction Translation and Compilation)**.** $\forall i, j.j = \textbf{jun}(i, \tau)$, and $P[i]$ is a branching instruction, let $i_b$ be the program point in $\llbracket i \rrbracket$ such that $P_{\text{DEX}}[i_b]$ is a branching instruction, then

$$\llbracket j \rrbracket[0] = \textbf{jun}(\llbracket i \rrbracket[i_b]) \text{ in } \llbracket P \rrbracket \qquad \text{and} \qquad \llbracket\!\llbracket j \rrbracket\!\rrbracket[0]\rrbracket = \textbf{jun}(\llbracket\!\llbracket i \rrbracket\!\rrbracket[i_b]\rrbracket) \text{ in } \llbracket\!\llbracket P \rrbracket\!\rrbracket.$$

**Property 6.2.4** (Security Environment Translation and Compilation)**.** $\forall i \in \mathcal{PP}, j \in \llbracket i \rrbracket.se(j) = se(i)$ in $\llbracket P \rrbracket$ and $\forall i \in \mathcal{PP}, j \in \llbracket i \rrbracket.se(\lceil j \rceil) = se(i)$ in $\llbracket\!\llbracket P \rrbracket\!\rrbracket$.

Then we need these subsequent lemmas which state that the successor relation is preserved through compilation process.

**Lemma 6.2.1.** *Let $P$ be a JVM program and $P[a] = Ins_a$ and $P[b] = Ins_b$ be two of its instructions at program points $a$ and $b$ (both are non-invoke instructions). Let $n_a > 0$ be the number of instructions translated from $Ins_a$. If $a \mapsto^{\text{Norm}} b$, then either*

$$\begin{array}{c} \llbracket \lfloor\!\lfloor a \rfloor\!\rfloor [n-1] \rrbracket \mapsto^{\text{Norm}} \llbracket \lfloor\!\lfloor b \rfloor\!\rfloor [0] \rrbracket \\ \textit{or} \\ \left( \begin{array}{c} \llbracket \lfloor\!\lfloor a \rfloor\!\rfloor [n-1] \rrbracket \mapsto^{\text{Norm}} (\llbracket \lfloor\!\lfloor a \rfloor\!\rfloor [n-1] \rrbracket + 1) \\ \textit{and} \\ (\llbracket \lfloor\!\lfloor a \rfloor\!\rfloor [n-1] \rrbracket + 1) \mapsto^{\text{Norm}} \llbracket \lfloor\!\lfloor b \rfloor\!\rfloor [0] \rrbracket \end{array} \right) \end{array}$$

*in $[\![P]\!]$.*

*Proof.* To prove this lemma, we first unfold the definition of compilation. Using the information that $a \mapsto b$, there are several possible cases to output the block depending on whether what instruction is $Ins_a$ and where $a$ and $b$ are located. Either:

- the instructions in $\lfloor\!\lfloor b \rfloor\!\rfloor$ are placed directly after $\lfloor\!\lfloor a \rfloor\!\rfloor$ and $\lfloor\!\lfloor a \rfloor\!\rfloor [n-1]$ is sequential instruction;
  In this case, appealing to the definition of successor relation the lemma holds trivially as the first case.

- $\lfloor\!\lfloor Ins_a \rfloor\!\rfloor$ ends in a sequential instruction and will have a **goto** instruction appended that points to $\llbracket \lfloor\!\lfloor b \rfloor\!\rfloor [0] \rrbracket$;
  Again appealing to the definition of successor relation this trivially holds as the second case, where
  $P_{\text{DEX}}[(\llbracket \lfloor\!\lfloor a \rfloor\!\rfloor [n-1] \rrbracket + 1)] = \textbf{goto}(\llbracket \lfloor\!\lfloor b \rfloor\!\rfloor [0] \rrbracket)$.

- $\lfloor\!\lfloor Ins_a \rfloor\!\rfloor [n-1]$ is a branching instruction and $b$ is one of its children, and $\lfloor\!\lfloor b \rfloor\!\rfloor$ is placed directly after $\lfloor\!\lfloor a \rfloor\!\rfloor$ or is pointed to by the branching instruction;
  Either case, using the definition of successor relation to establish that we are in the first case.

- $\lfloor\!\lfloor Ins_a \rfloor\!\rfloor [n-1]$ is a branching instruction and $b$ is one of its children, nevertheless $\lfloor\!\lfloor b \rfloor\!\rfloor$ is not placed directly after $\lfloor\!\lfloor a \rfloor\!\rfloor$ nor is pointed to by the branching instruction;
  In this case, according to the **Output** phase, a **goto** instruction will be appended in $(\llbracket \lfloor\!\lfloor a \rfloor\!\rfloor [n-1] \rrbracket + 1)$, and thus we are in the second case. Use the definition of successor relation to conclude the proof.

□

**Lemma 6.2.2.** *Let $P$ be a JVM program and $P[a] = Ins_a$ and $P[b] = Ins_b$ be two of its instructions at program points $a$ and $b$ where $b$ is the address of the first instruction in the exception handler $h$ for a throwing exception $\tau$. Let $e$ be the index to the instructions within $\lfloor\!\lfloor a \rfloor\!\rfloor$ that throws exception. If $a \mapsto^{\tau} b$, then $\llbracket \lfloor\!\lfloor a \rfloor\!\rfloor [e] \rrbracket \mapsto^{\tau} \llbracket \lfloor\!\lfloor h \rfloor\!\rfloor \rrbracket$ and $\llbracket \lfloor\!\lfloor h \rfloor\!\rfloor \rrbracket \mapsto^{\text{Norm}} \llbracket \lfloor\!\lfloor b \rfloor\!\rfloor [0] \rrbracket$*

*Proof.* Trivial based on the unfolding definition of the compiler, where there is a block containing sole instruction **moveexception**, which will be pointed by the exception

handler in DEX, between possibly throwing instruction and its handler for particular exception class $\tau$. The proof is then concluded by successor relation in DEX. □

**Lemma 6.2.3.** *Let $P$ be a JVM program and $P[a]$ = **invoke** and $P[b] = Ins_b$ be two of its instructions at program points $a$ and $b$. If $a \mapsto^{\text{Norm}} b$, then $\lceil\lfloor a \rfloor\rfloor[0]\rceil \mapsto^{\text{Norm}} \lceil\lfloor a \rfloor\rfloor[1]\rceil$ and $\lceil\lfloor a \rfloor\rfloor[1]\rceil \mapsto^{\text{Norm}} \lceil\lfloor b \rfloor\rfloor[0]\rceil$*

*Proof.* This is trivial based on the unfolding definition of the compiler since the primary successor of $\lceil\lfloor a \rfloor\rfloor[0]\rceil$ is $\lceil\lfloor a \rfloor\rfloor[1]\rceil$, where $\lceil P\rceil[\lceil\lfloor a \rfloor\rfloor[1]\rceil]$ = **moveresult**, and the primary successor of $\lceil\lfloor a \rfloor\rfloor[1]\rceil$ is $\lceil\lfloor b \rfloor\rfloor[0]\rceil$. The proof is then concluded by the definition of successor relation in DEX. □

**Lemma 6.2.4.** *Let $P$ be a JVM program and $P[a] = Ins_a$ and $P[b] = Ins_b$ be two of its instructions at program points $a$ and $b$. Suppose $Ins_b$ is translated to an empty sequence (e.g. $Ins_b$ is pop or goto). Let $s$ be the first in the successor chain of $b$ such that $\lfloor P[s] \rfloor$ is non-empty (we can justify this successor chain as an instruction that causes branching will never be translated into empty sequence). If $a \mapsto b$, then either*

$$\begin{gathered}\lceil\lfloor a \rfloor\rfloor[n-1]\rceil \mapsto^{\text{Norm}} \lceil\lfloor s \rfloor\rfloor[0]\rceil \\ \textit{or} \\ \left(\begin{array}{c}\lceil\lfloor a \rfloor\rfloor[n-1]\rceil \mapsto^{\text{Norm}} (\lceil\lfloor a \rfloor\rfloor[n-1]\rceil + 1) \\ \textit{and} \\ (\lceil\lfloor a \rfloor\rfloor[n-1]\rceil + 1) \mapsto^{\text{Norm}} \lceil\lfloor s \rfloor\rfloor[0]\rceil\end{array}\right)\end{gathered}$$

*Proof.* We use induction on the length of successor's chain. In the base case where the length is 0, we can use Lemma 6.2.1 to establish that this lemma holds. For the case where the length is $n+1$, there are two possibilities for the last instruction in the chain :

- the successor is the next instruction
  In this case, using the definition of successor relation, we know that it will be in the first case.
- the successor is not the next instruction Since there will be a **goto** appended, it will fall to the second case. Using the successor relation, we know that the latter property holds.

then use IH to conclude. □

**Property 6.2.5** (Region Translation and Compilation for **invoke**)**.** $\forall i.P_{DEX}[i]$ = **invoke**, $i+1 \in$ **region**$(i,)$ ($i+1$ will be the program point for **moveresult**).

**Property 6.2.6** (Region Translation and Compilation for handler)**.** $\forall i, j.j \in$ **region**$(i)$, let $i_e$ be the instruction in $\lfloor P[i] \rfloor$ that possibly throws, then

$$\begin{gathered}\textbf{handler}(i_e, \tau) \in \textbf{region}(i_e, \tau) \text{ in } \lfloor P \rfloor \\ \text{and} \\ \textbf{handler}(\lceil i_e \rceil, \tau) \in \textbf{region}(\lceil i_e \rceil, \tau) \text{ in } [\![P]\!]\end{gathered}$$

(note that the handler will point to **moveexception**).

**Lemma 6.2.5** (SOAP Preservation)**.** *The SOAP properties are preserved in the translation from JVM to DEX, i.e. if the JVM program satisfies the SOAP properties, so does the translated DEX program.*

*Proof.* We proceed by exhaustion, that is if the original JVM bytecode satisfies SOAP, then the resulting translation to DEX instructions will also satisfy each of the property.

**SOAP1.** Since the JVM bytecode satisfies SOAP, that means $i$ is a branching point which will also be translated into a sequence of instructions. Denote $i_b$ as the program point in the sequence and suppose it is a branching point. Let $\llbracket P[k] \rrbracket$ be the translation of instruction $P[k]$ and $k_1$ be the address of its first instruction ($\llbracket P[k] \rrbracket[0]$). Using the first case in the Lemma 6.2.1, Lemma 6.2.2, Lemma 6.2.3 and Lemma 6.2.4, we know that $i_b \mapsto k_1$. In the case that $k \in \mathbf{region}(i, \tau)$, we know that $k_1 \in \mathbf{region}(i_b, \tau)$ using Property 6.2.1. In the case that $k = \mathbf{jun}(i_b, \tau)$, we then will have $k_1 = \mathbf{jun}(i_b, \tau)$ using Property 6.2.3.

Special cases for the second case of Lemma 6.2.1, Lemma 6.2.2 and Lemma 6.2.3 in that they contain additional instructions in the lemma. We argue that the property still holds using Property 6.2.5 Property 6.2.6, and Property 6.2.2. Suppose $k'$ is the program point that points to the extra instruction, then we have $k' \in \mathbf{region}(i_b, \tau)$ from the three definitions we have mentioned. Following the argument from before, we can conclude that $k_1 \in \mathbf{region}(i, \tau)$ or $k_1 = \mathbf{jun}(i_b, \tau)$.

**SOAP2.** Let $j_n$ be the last instruction in $[\![P[j]]\!]$. Denote $i_b$ as the program point in the sequence $[\![i]\!]$ and suppose it is a branching point. Let $\llbracket P[k] \rrbracket$ be the translation of instruction $P[k]$ and $k_1$ be the address of its first instruction ($\llbracket P[k] \rrbracket[0]$). Using Property 6.2.1, we obtain $j_n \in \mathbf{region}(i_b, \tau)$. Using the first case of Lemma 6.2.1, Lemma 6.2.2, Lemma 6.2.3 and Lemma 6.2.4 we will get that $j_n \mapsto k_1$. Now since the JVM bytecode satisfies SOAP, we know that there are two cases we need to take care of and $k$ will fall to one case or the other. Assume $k \in \mathbf{region}(i, \tau)$. This means using Property 6.2.1 we will have $k_1 \in \mathbf{region}(i_n, \tau)$. Assume $k = \mathbf{jun}(i, \tau)$, we use Property 6.2.3 and obtain that $k_1 = \mathbf{jun}(i_n, \tau)$. Either way, the SOAP property is preserved for SOAP2. Similar argument as SOAP1 to establish the second case of Lemma 6.2.1, and that the property is still preserved in the presence of **moveresult** and **moveexception**.

**SOAP3.** Trivial

**SOAP4.** Let $k_1 = \mathbf{jun}(i, \tau_1)$ and $k_2 = \mathbf{jun}(i, \tau_2)$ (this may be a bit confusing, this program point here refers to the program point in JVM bytecode). Let $i_b$ be the instruction in $[\![i]\!]$ that branch and $k_{11}$ and $k_{21}$ be the first instruction in $[\![P[k_1]]\!]$ and $[\![P[k_2]]\!]$ respectively. We proceed by using Property 6.2.1 and the knowledge that the JVM bytecode satisfies SOAP4 to establish that when $k_{11} \neq k_{21}$, then $k_{11} \in \mathbf{region}(i_b, \tau_2)$ or $k_{21} \in \mathbf{region}(i_b, \tau_1)$ thus the DEX program will also satisfy SOAP4.

**SOAP5.** For any $\mathbf{jun}(i,\tau')$ such that it is defined, let program point $k = \mathbf{jun}(i,\tau')$. Using Property 6.2.3 we have $k_1 = \mathbf{jun}(i_n,\tau')$. Using Property 6.2.1, we know that $k_1 \in \mathbf{region}\ (i_n,\tau)$. If we then set $k_1$ to be such a point, where $\mathbf{jun}(i_n,\tau')$ and $\mathbf{jun}(i_n,\tau') \in \mathbf{region}\ (i_n,\tau)$ for any $\tau'$ with junction point defined, the property then holds.

**SOAP6.** Is similar to the way of proving SOAP5, with the addition of simple property where the size of a code and its translation is covariant in a sense that if a program $a$ has more code than $b$, then $[\![a]\!]$ also has more code than $[\![b]\!]$.

□

### 6.2.2 Compilation Preserves Typability

There are several assumptions we make for this compilation. Firstly, the JVM program will not modify its self-reference for an object. Secondly, since now we are going to work in blocks, the notion of $se, S$, and $RT$ will also be defined in term of this addressing. A new scheme for addressing $blockAddress$ is defined from sets of pairs $(bi,j)$, $bi \in blockIndex$, a set of all block indices (label of the first instruction in the block), where $\forall i \in \mathcal{PP}.\exists bi,j.$ s.t.$bi + j = i$. We also add additional relation $\Rightarrow^*$ to denote the reflexive and transitive closure of $\Rightarrow$ to simplify the typing relation between blocks.

We overload $\|.\|$ and $[\![.]\!]$ to also apply to stack types to denote the translation from stack type into typing for registers. This translation basically just maps each element of the stack to registers at the end of registers containing the local variables (with the top of the stack with larger index, i.e., stack expanding to the right). More formally, if there are $n$ local variables denoted by $v_1,\ldots,v_n$ and stack type with the height of $m$ (0 denotes the top of the stack), then $[\![st]\!] = \{r_0 \mapsto \vec{k_a}(v_1),\ldots,r_{n-1} \mapsto \vec{k_a}(v_n), r_n \mapsto st[m-1],\ldots,r_{n+m-1} \mapsto st[0]\}$. Lastly, the function $\lceil\!\lceil.\rceil\!\rceil$ is also overloaded for addressing $(bi,i)$ to denote abstract address in the DEX side which will actually be instantiated when producing the output DEX program from the blocks.

**Property 6.2.7** (Stack Type Translation)**.** $\forall i \in \mathcal{PP}, RT_{\|i\|[0]} = \|S_i\|$.

To prove the typability preservation of the compilation processes, we define an intermediate type system that closely resembles that of DEX, except that the addressing uses block addressing. The purpose of this intermediate addressing is to know the existence of register typing to satisfy typability and the constraint satisfaction for each instruction. We defer the details to Appendix A to avoid more clutter.

The following monotony lemma is useful in proving the relation $\sqsubseteq$ between registers typing obtained from compiling stack types.

**Lemma 6.2.6** (Monotonicity of Translation)**.** *Let $rt$ be a register types and $S_1$ and $S_2$ stack types. If we have $rt \sqsubseteq \|S_1\|$ and $S_1 \sqsubseteq S_2$, then $rt \sqsubseteq \|S_2\|$ as well.*

*Proof.* Trivial based on the definition of $\|.\|$ and the $\sqsubseteq$ for register types. □

The idea of the proof that compilation from JVM bytecode to DEX bytecode preserves typability is that any instruction that does not modify the block structure can be proved using Lemma 6.2.7 and Lemma 6.2.9 to prove the typability of register typing.

Initially, we state lemmas saying that typable JVM instructions will yield typable DEX instructions. Paired with each normal execution is the lemma for the exception throwing one. These lemmas are needed to handle the additional block of **moveexception** attached to each exception handler.

**Lemma 6.2.7** (Typeable Sequence)**.** *For any JVM program P with instruction Ins at address i , let the length of* $\llbracket Ins \rrbracket$ *be n. Let* $RT_{\llbracket i \rrbracket[0]} = \llbracket S_i \rrbracket$*. If according to the transfer rule for* $P[i] = Ins$ *there exists st s.t.* $i \vdash S_i \Rightarrow st$ *then*

$$\begin{array}{c} \left( \ \forall 0 \leq j < (n-1). \exists rt'. \llbracket i \rrbracket[j] \vdash RT_{\llbracket i \rrbracket[j]} \Rightarrow rt', rt' \sqsubseteq RT_{\llbracket i \rrbracket[j+1]} \ \right) \\ \textit{and} \\ \exists rt. \llbracket i \rrbracket[n-1] \vdash RT_{\llbracket i \rrbracket[n-1]} \Rightarrow rt, rt \sqsubseteq \llbracket st \rrbracket \end{array}$$

*according to the typing rule(s) of* $\llbracket Ins \rrbracket$*.*

*Proof.* It is a case by case instruction, although for most of the instructions they are straightforward as they only translate into one instruction. For the rest of the proof, using Property 6.2.4 to say that the translated $se(\llbracket i \rrbracket)$ have the same security level as $se(i)$.

- **Push**
  We appeal directly to both of the transfer rules of **Push** and **Const**. In **Push** case, it only appends top of the stack with $se(i)$. Let such $rt$ be

  $$rt = RT_{\llbracket i \rrbracket[0]} \oplus \{r(TS_i) \mapsto se(\llbracket i \rrbracket[0])\}$$

  referring to **Const** transfer rule. Since **Push** is translated into **Const**$(r(TS_i))$, where $TS_i$ corresponds to the top of the stack, we know that $\llbracket st \rrbracket = rt$ because $RT_{\llbracket i \rrbracket[0]} = \llbracket S_i \rrbracket$ and the $rt$ we have is the same as $\llbracket se(i) :: S_i \rrbracket$ thus $rt \sqsubseteq \llbracket st \rrbracket$

- **Pop**
  In this case, since the instruction does not get translated, this instruction does not affect the lemma.

- **Load** $x$
  Similar to **Push** except that the security value pushed on top of the stack is $se(\llbracket i \rrbracket[0]) \sqcup \vec{k_a}(x)$. And although there are several transfer rules for **move**, there is only one applicable because the source register comes from a local variable register, and the target register is one of the stack space. Using this transfer rule, we can trivially show that $rt = \llbracket st \rrbracket$ where $st = \left(se(i) \sqcup \vec{k_a}(x)\right) :: S_i$ and $rt = RT_{\llbracket i \rrbracket[0]} \oplus \{r(TS_i) \mapsto se(\llbracket i \rrbracket[0]) \sqcup \vec{k_a}(x)\}$, thus $rt \sqcup \llbracket st \rrbracket$.

- **Store** $x$
  This instruction is also translated as **move** except that the source register is the

top of the stack and the target register is one of the local variable registers. The $rt$, in this case, will be

$$rt = RT_{[\![i]\!][0]} \oplus \{\, r_x \mapsto se([\![i]\!][0]) \sqcup RT_{[\![i]\!][0]}\big(r(TS_i - 1)\big)\}$$

This $rt$ coincides with the transfer rule for **move** where the target register is a register used to contain local variable. Since we know that the $x$ is in the range of local variable, we will have that $rt \sqsubseteq [\![st]\!]$ based on the definition of $\sqsubseteq$, $[\![.]\!]$ of flattening a stack.

- **Goto**
  This instruction does not get translated just like **Pop**, so this instruction also does not affect the lemma.

- **Ifeq** $t$
  This instruction is translated to conditional branching in the DEX instruction. There are two changes applied to the stack types; one is that the removal of the top value of the stack which is justified by the definition of $\sqsubseteq$, and then lifting the value of the rest of the stack. Since there is no lift involved in DEX, we know that this assignment will preserve typability as the registers are assigned higher security levels.

- **Binop**
  Translated as a DEX instruction for specified binary operator with the source taken from the top two values from the stack, and then put the resulting value in the then would be top of the stack. Let $rt$, in this case, comes from

$$\begin{aligned} rt =& RT_{[\![i]\!][0]} \oplus \{r(TS_i - 1) \mapsto se([\![i]\!][0]) \sqcup \\ & RT_{[\![i]\!][0]}\big(r(TS_i - 1)\big) \sqcup \big(RT_{[\![i]\!][0]}\big(r(TS_i - 2)\big)\big)\} \end{aligned}$$

  This $rt$ corresponds to the scheme of DEX transfer rule for binary operation. Then we will have that $rt \sqsubseteq [\![st]\!]$ where $st = se(i) \sqcup k_a \sqcup k_b :: st'$ and $S_i = k_a :: k_b :: st'$

- **Swap**
  In dx tool, this instruction is translated into four **move** instructions. In this case, the $rt$ is

$$\begin{aligned} rt = RT_{[\![i]\!][0]} \oplus\{& \\ r(TS_i) \;\; &\mapsto se([\![i]\!][0]) \sqcup RT_{[\![i]\!][0]}\big(r(TS_i - 2)\big), \\ r(TS_i + 1) &\mapsto se([\![i]\!][1]) \sqcup RT_{[\![i]\!][1]}\big(r(TS_i - 1)\big), \\ r(TS_i - 2) &\mapsto se([\![i]\!][2]) \sqcup RT_{[\![i]\!][2]}\big(r(TS_i - 1)\big), \\ r(TS_i - 1) &\mapsto se([\![i]\!][3]) \sqcup RT_{[\![i]\!][3]}\big(r(TS_i - 2)\big)\} \end{aligned}$$

  justified by applying transfer rule for **move** four times. As before, appealing to the definition of $\sqsubseteq$ to establish that this $rt \sqsubseteq [\![st]\!]$ where $st = k_b :: k_a :: st'$ and $S_i = k_a :: k_b :: st'$.

There's a slight subtlety here in that the relation might not hold due to the presence of *se* in *rt* even though there is no such occurrence in *st*. But on closer inspection, we know that in the case of swap instruction, the effect of *se* will be nothing. There are two cases to consider:

- If the value in the operand stacks is already there before *se* is modified. We know that this can be the case only when there was a conditional branch before, which also means that the operand stacks will be lifted to the level of the guard and the level of *se* is determined by this level of the guard as well. So practically, they are the same thing
- If the value in the operand stacks is put after *se* is modified. Based on the transfer rules of the instructions that put a value on top of the stack, they will lub *se* with the values. Therefore, another lub with *se* will have no effect.

For the first property, we have the register typing below

$$\begin{array}{lcl} RT_{\llbracket i \rrbracket[1]} & = & RT_{\llbracket i \rrbracket[0]} \oplus \{r(TS_i) \mapsto se(\llbracket i \rrbracket[0]) \sqcup RT_{\llbracket i \rrbracket[0]}\big(r(TS_i - 2)\big)\} \\ RT_{\llbracket i \rrbracket[2]} & = & RT_{\llbracket i \rrbracket[1]} \oplus \{r(TS_i + 1) \mapsto se(\llbracket i \rrbracket[1]) \sqcup RT_{\llbracket i \rrbracket[1]}\big(r(TS_i - 1)\big)\} \\ RT_{\llbracket i \rrbracket[3]} & = & RT_{\llbracket i \rrbracket[2]} \oplus \{r(TS_i - 2) \mapsto se(\llbracket i \rrbracket[2]) \sqcup RT_{\llbracket i \rrbracket[3]}\big(r(TS_i - 1)\big)\} \end{array}$$

which satisfy the property.

- **New**
  The argument that goes for this instruction is exactly the same as that of **Push**, where the *rt*, in this case, is $\llbracket S_i \rrbracket \oplus \{r(TS_i) \mapsto se(\llbracket i \rrbracket[0])\}$.

- **Getfield**
  In this case, the transfer rule for the translated instruction coincides with the transfer rule for **Getfield**. Let

$$rt = RT_{\llbracket i \rrbracket[0]} \oplus \{r(TS_i - 1) \mapsto se(\llbracket i \rrbracket[0]) \sqcup \mathbf{ft}(f)\}$$

  Then we have $rt = \llbracket st \rrbracket$ which can be trivially shown with $st = se(i) \sqcup \mathbf{ft}(f) :: S_i$ thus giving us $rt \sqsubseteq \llbracket st \rrbracket$.

- **Putfield**
  Since the JVM transfer rule for the operation itself only removes the top 2 of the stack items, and the transfer rule for DEX keeps the register typing, when we have $rt = \llbracket S_i \rrbracket$, then by the definition of $\sqsubseteq$ we'll have $rt \sqsubseteq \llbracket st \rrbracket$ since $S_i = k_o :: k_v :: st$.

- **Newarray**
  Similar to the argument of **load**, we have

$$rt = RT_{\llbracket i \rrbracket[0]} \quad \oplus \{r(TS_i - 1) \mapsto RT_{\llbracket i \rrbracket[0]}\big(r(TS_i - 1)\big)[\mathbf{at}(\llbracket i \rrbracket[0])]\}$$

  , $rt = \llbracket st \rrbracket$, where $st = k[at(i)] :: st'$, $S_i = k :: st'$, which will give us $rt \sqsubseteq \llbracket st \rrbracket$.

- **Arraylength**
  Let $k[k_c] = RT_{[\![i]\!][0]}\big(r(TS_i - 1)\big) = S_i[0]$. In this case $rt = RT_{[\![i]\!][0]} \oplus \{r(TS_i - 1) \mapsto k\} = [\![st]\!]$ then we will have $rt = [\![st]\!]$ where $st = k :: st'$ and $S_i = k[k_c] :: st'$ which will give us $rt \sqsubseteq [\![st]\!]$.

- **Arrayload**
  Let $k[k_c] = RT_{[\![i]\!][0]}\big(r(TS_i - 2)\big) = S_i[1]$. In this case

$$rt = RT_{[\![i]\!][0]} \oplus \{r(TS_i - 2) \mapsto \big(se(i) \sqcup k \sqcup RT_{[\![i]\!][0]}(r(TS_i - 1))\big) \sqcup^{\text{ext}} k_c\}$$

  which coincides with $[\![st]\!]$ where $st = (k \sqcup k_i) \sqcup^{\text{ext}} k_c :: st'$ and $S_i = k_i :: k[k_c] :: st'$ except for lub with $se(i)$. The similar reasoning with **Swap** where lub with $se(i)$ in this case will have no effect.

- **Arraystore**
  Similar argument as **putfield** where the JVM instruction removes the top of the stack and the DEX instruction preserves the register typing for $rt$. Thus appealing to the definition of $\sqsubseteq$ we have that $rt \sqsubseteq [\![st]\!]$.

- **Invoke**
  This instruction itself yields 1 or 2 instructions depending on whether the function returns a value or not. Since the assumption for JVM type system is that functions always return a value, the translation will be that **invoke** and **moveresult** except that **moveresult** will always be in the region Norm. Let $\vec{k'_a} \xrightarrow{k'_h} \vec{k'_r}$ be the policy for method invoked. Type system wise, there will be three different cases for this instruction, normal execution, caught, and uncaught exception. For this lemma, the only one applicable is normal execution since it is the one tagged with Norm. There will be two resulting instructions since it will also contain the instruction **moveresult**. Let $st_1$ be the stack containing the function's arguments, $t$ be the top of the stack after popping the function arguments from the stack and the object reference $t = locN + (\textbf{length}(S_i) - \textbf{length}(st_1) - 1)$, where $locN$ is the number of local variables. Let $k$ be the security level of object referenced and $k_e = \bigsqcup \{\vec{k'_r}(e) \mid e \in \textbf{excAnalysis}(m_{\text{ID}})$. Since the method can also throw an exception, we must also include the lub of security level for possible exceptions, denoted by $k_e$. In this case, such $rt$ will be

$$RT_{[\![i]\!][0]} \oplus \{\quad ret \mapsto \big(\vec{k'_r}(\text{Norm}) \sqcup se([\![i]\!][0])\big), r_t \mapsto \big(\vec{k'_r}(\text{Norm}) \sqcup se([\![i]\!][1])\big)\}$$

  and by definition of $\sqsubseteq$ we will have that $rt \sqsubseteq [\![st]\!]$, where $st = \textbf{lift}_{\mathbf{k} \sqcup \mathbf{k_e}}(\big(\vec{k'_r}[n] \sqcup se(i)\big) :: st_2)$ and $S_i = st_1 :: k :: st2$. With that form of $rt$ in mind, then the registers typing for $[\![i]\!][1]$ can be

$$RT_{[\![i]\!][0]} \oplus \{ret \mapsto \big(\vec{k'_r}(\text{Norm}) \sqcup se([\![i]\!][0])\big)$$

  coming from the transfer rule of **invoke** in DEX.

- **Throw**
  This lemma will never apply to **Throw** since if the exception is caught, then the successor will be in the tag $\tau \neq$ Norm, but if the exception is uncaught, then the instruction is a return point.

□

**Lemma 6.2.8.** *For any JVM program P with instruction Ins at address i and tag* $\tau \neq$ Norm *with exception handler at address* $i_e$*, let the length of* $\llbracket Ins \rrbracket$ *until the instruction that throws an exception* $\tau$ *be denoted by n. Let* $(be, 0) = \llbracket i_e \rrbracket$ *be the address of the handler for that particular exception. If according to the transfer rule for Ins* $i \vdash^{\tau} S_i \Rightarrow st$*, then*

$$\begin{gathered}\left( \ \forall 0 \leq j < (n-1). \exists rt'. \llbracket i \rrbracket[j] \vdash^{\text{Norm}} RT_{\llbracket i \rrbracket[j]} \Rightarrow rt', rt' \sqsubseteq RT_{\llbracket i \rrbracket[j+1]} \ \right) \\ \textit{and} \\ \exists rt. \llbracket i \rrbracket[n-1] \vdash^{\tau} RT_{\llbracket i \rrbracket[n-1]} \Rightarrow rt, rt \sqsubseteq RT_{(be,0)} \\ \textit{and} \\ \exists rt. (be, 0) \vdash^{\text{Norm}} RT_{(be,0)} \Rightarrow rt, rt \sqsubseteq \llbracket st \rrbracket\end{gathered}$$

*according to the transfer rule(s) of first n instruction in* $\llbracket Ins \rrbracket$ *and* **moveexception**.

*Proof.* Case by case of possibly throwing instructions:

- **Invoke**
  We only need to take care of the case where the exception is caught, as an uncaught exception is a return point (there is no successor). In this case, $n = 1$ as the instruction that may throw is the **invoke** itself. Therefore, the first property trivially holds (**moveexception** can't possibly throw an exception). Let $locN$, in this case, be the number of local variables, and $e$ be the exception thrown. Let $k$ be the security level of the object referenced. In this case, the last $rt$ will take the form

  $$rt = \{\vec{k_a}, ex \mapsto (k \sqcup \vec{k'_r}(e)), r(locN) \mapsto (k \sqcup \vec{k'_r}(e))\}$$

  Again with this $rt$, we will have $rt \sqsubseteq \llbracket st \rrbracket$, where $st = (k \sqcup \vec{k'_r}[e]) :: \epsilon$. Such an $rt$ is obtained from the transfer rule for **invoke** where an exception of tag $\tau$ is thrown, and the transfer rule for **moveexception**. Then we have the register typing for $(be, 0)$ as

  $$RT_{(be,0)} = \{\vec{k_a}, ex \mapsto (k \sqcup \vec{k'_r}(e))\}$$

  which fulfills the second property (transfer rule from **invoke**) and the last property, which when joined with the transfer rule for **moveexception** will give us the $rt$ that we want.

- **Throw**
  The argument follows that of **Invoke** for the caught and uncaught exception. For an uncaught exception, there is nothing to prove here as there is no resulting $st$. For a caught exception, let $k$ be the security level of the exception and

$locN$ be the number of local variables. Such $rt$ can be

$$rt = \{\vec{k_a}, ex \mapsto \big(k \sqcup se(\llbracket i \rrbracket[0])\big), r(locN) \mapsto \big(k \sqcup se(\llbracket i \rrbracket[0])\big)\}$$

and it will make the relation $rt \sqsubseteq \llbracket st \rrbracket$ holds, where $st = (k \sqcup se(i)) :: \epsilon$ . This $rt$ comes from the transfer rules for **throw** and **moveexception** combined. Registers typing for $(be, 0)$ takes the form of

$$RT_{(be,0)}\{\vec{k_a}, ex \mapsto (k \sqcup se(\llbracket i \rrbracket[0])\}$$

which will give us the final $rt$ we want after the transfer rule for **moveexception**

- Other possibly throwing instruction
  Essentially they are the same as that of **throw** where the security level that we are concerned with is the security level of the object lub-ed with its security environment. The will also come from the transfer rule of each respective instruction throwing a null pointer exception combined with the rule for **moveexception**.

□

**Lemma 6.2.9** (Typeable Translation)**.** *Let $Ins$ be an instruction at address $i$, $i \mapsto j$, $st$, $S_i$ and $S_j$ are stack types such that $i \vdash S_i \Rightarrow st, st \sqsubseteq S_j$. Let $n$ be the length of $\llbracket Ins \rrbracket$. Let $RT_{\llbracket i \rrbracket[0]} = \llbracket S_i \rrbracket$, let $RT_{\llbracket j \rrbracket[0]} = \llbracket S_j \rrbracket$ and $rt$ be registers typing obtained from the transfer rules involved in $\llbracket Ins \rrbracket$. Then $rt \sqsubseteq RT_{\llbracket j \rrbracket[0]}$.*

*Proof.* Using Lemma 6.2.7 and Lemma 6.2.8 to establish that we have $rt \sqsubseteq [\![st]\!]$. Then we conclude by using Lemma 6.2.6 to establish that $rt \sqsubseteq RT_{\llbracket j \rrbracket[0]}$ because $st \sqsubseteq S_j$. □

We need an additional lemma to establish that the constraints in the JVM transfer rules are satisfied after the translation. This is because the definition of typability also relies on the constraint which can affect the existence of register typing.

**Lemma 6.2.10** (Constraint Satisfaction)**.** *Let $Ins$ be an instruction at program point $i$, $S_i$ its corresponding stack types, and let $RT_{\llbracket i \rrbracket[0]} = \llbracket S_i \rrbracket$. If $P[i]$ satisfy the typing constraint for $Ins$ with the stack type $S_i$, then $\forall (bj, j) \in \llbracket i \rrbracket . P_{DEX}[bj, j]$ will also satisfy the typing constraints for all instructions in $\llbracket Ins \rrbracket$ with the initial registers typing $RT_{\llbracket i \rrbracket[0]}$.*

*Proof.* We do a case by case examination of the instructions:

- **Push**
  This instruction is translated into **Const** which does not have a constraint.
- **Pop**: does not get translated.
- **Load** $x$
  This instruction will be translated to **Move** which does not have a constraint.
- **Store** $x$
  This instruction will be translated to **Move** which does not have a constraint.

- **Goto**: does not get translated
- **Ifeq** $t$
  This instruction will get translated to an **ifeq** instruction where the condition is based on top of the stack ($TS_i - 1$). There is only one constraint of the form $\forall j' \in \textbf{region}(i, \text{Norm}), rt\big(r(TS_i - 1)\big) \leq se(j')$, and we know that in the JVM bytecode the constraint $\forall j' \in \textbf{region}(i, \text{Norm}), k \leq se(j')$ is fulfilled. Based on the definition of $[\![.]\!]$, we will have $k = rt\big(r(TS_i - 1)\big)$. Thus we only need to prove that the difference in the region will still preserve the constraint. We do this by proof by contradiction. Suppose there exists an instruction at address $(bj, j) \in \textbf{region}([\![i]\!][n])$ such that $k \not\leq se(bj, j)$. But according to Property 6.2.1, such an instruction will come from an instruction at address $i'$ s.t. $i' \in \textbf{region}(i)$ thus it will satisfy $k \leq se(i')$. By Property 6.2.4, $se(bj, j) = se(i')$, thus we will have $k \leq se(bj, j)$. A plain contradiction.
- **Binop**
  This instruction will be translated to **Binop** or **BinopConst**, both of which do not have a constraint.
- **Swap**
  Trivially holds as well because all the four **move** instructions translated from **swap** do not have a constraint.
- **New**
  Trivially holds as the target instruction **New** does not have a constraint.
- **Getfield**
  There are different sets of constraints depending on whether the instruction executes normally, throws a caught exception, or throws an uncaught exception.

  In the case of **Getfield** executing normally, there are only two constraints that we need to track; one is that $rt(r_o) \in \mathcal{S}$ and the second is $\forall j \in \textbf{region}(i, \text{Norm})$, $rt(r_o) \leq se(j)$. The first constraint is trivial since we already have that in JVM the constraint $k \in \mathcal{S}$ is satisfied, where $S_i = k :: st$ for some stack type $st$. We know that based on the definition of $[\![S_i]\!]$ we have $rt(r_o) = k$. Therefore we can conclude that $rt(r_o) \in \mathcal{S}$. The second constraint follows, using a similar argument to the satisfaction of region constraint in **Ifeq**.

  In the case of **Getfield** throwing an exception, we then know that based on the compilation scheme, depending on whether the exception is caught or not, the same thing will apply to the translated instruction **iget**, i.e., if **Getfield** has a handler for np, so does **iget** and if **Getfield** does not have a handler for np, **iget** does not either. Thus we only need to take care of one more constraint in that if this instruction does throw an uncaught exception, then it will satisfy $se(i) \sqcup rt(r_o) \leq \vec{k_r}(\text{np})$. This constraint also trivially holds as the policy is translated directly, i.e. $\vec{k_r}(\text{np})$ is the same both in the JVM type system and the DEX type system, and that $rt(r_o) = k$. Since the JVM typing satisfy $k \leq \vec{k_r}(\text{np})$, then so does the DEX typing.

- **Putfield**
  To prove the constraint satisfaction for this instruction we appeal to the translation scheme and the definition of $[\![.]\!]$. We know from the translation scheme that the resulting instruction is $\mathbf{iput}(r(TS_i - 1), r(TS_i - 2), f)$, so the top of the stack $(TS_i - 1)$ corresponds to $r_s$ and the second to top of the stack $(TS_i - 2)$ corresponds to $r_o$. From the JVM transfer rule, we know that the security level of $S_i[0]$ (denoted by $k_1$) is in the set of $\mathcal{S}^{\text{ext}}$ and the security level of $S_i[1]$ is in the set of $\mathcal{S}$. Thus we then know that the constraints $rt(r_o) \in \mathcal{S}$ and $rt(r_s) \in \mathcal{S}^{\text{ext}}$ are fulfilled since we have $rt(TS_i - 1) = S_i[0]$ and $rt(TS_i - 2) = S_i[1]$.

  Now for constraints $k_h \leq \mathbf{ft}(f)$ and, $(rt(r_o) \sqcup se(i)) \sqcup^{\text{ext}} rt(r_s) \leq \mathbf{ft}(f)$ we know that the policies are translated directly. Thus the constraint $k_h \leq \mathbf{ft}(f)$ trivially holds. For the other constraint, we know that $k_1 = rt(r_s)$, $k_2 = rt(r_o)$, and $se$ stays the same, therefore the constraint $(rt(r_o) \sqcup se(i)) \sqcup^{\text{ext}} rt(r_s) \leq \mathbf{ft}(f)$ is also satisfied because $(k_2 \sqcup se(i)) \sqcup^{\text{ext}} k_1 \leq \mathbf{ft}(f)$ is assumed to be satisfied. Lastly, for the rest of the constraints refer to the proof in **Getfield** as they are essentially the same (the constraint for the region, handler's existence / non-existence, and constraint against $\vec{k_r}$ on uncaught exception).

- **Newarray**
  Trivially holds as the translated instruction does not have a constraint.

- **Arraylength**
  We first deal with the constraints $k \in \mathcal{S}$ and $k_c \in \mathcal{S}^{\text{ext}}$. From the definition of $[\![.]\!]$, we know that $rt(r_a) = k[k_c]$. Since JVM typing satisfies these constraints, it follows that DEX typing also satisfies this constraint. For the rest of the constraints refer to the proof in **Getfield** as they are essentially the same (the constraint for the region, handler's existence / non-existence, and constraint against $\vec{k_r}$ on uncaught exception).

- **Arrayload**
  We first deal with the constraints $k, rt(r_i) \in \mathcal{S}$ and $k_c \in \mathcal{S}^{\text{ext}}$. From the definition of $[\![.]\!]$, we know that $rt(r_a) = k_2[k_c]$ and $rt(r_i) = k_1$. Since we know that JVM typing satisfies all the constraint, we know that $rt(r_i) \in \mathcal{S}$ since $k_1 \in \mathcal{S}$, $k \in \mathcal{S}$ since $k_2 \in \mathcal{S}$, and $k_c \in \mathcal{S}^{\text{ext}}$ since in JVM typing $k_c \in \mathcal{S}^{\text{ext}}$. For the rest of the constraints refer to the proof in **Getfield** as they are essentially the same (the constraint for the region, handler's existence / non-existence, and constraint against $\vec{k_r}$ on uncaught exception).

- **Arraystore**
  Similar to that of **Putfield**, where $rt(r_s) = k_1$, $rt(r_i) = k_2$, and $k_3[k_c] = rt(r_a) = k'[k'_c]$. $k_2, k_3 \in \mathcal{S}$ gives us $k', rt(r_i) \in \mathcal{S}$ and $k_1, k_c \in \mathcal{S}^{\text{ext}}$ gives us $k'_c, rt(r_s) \in \mathcal{S}^{\text{ext}}$. In this setting as well, it is easy to show that DEX typing satisfies $((k' \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k'_c$ because JVM typing satisfies $((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \leq^{\text{ext}} k_c$. For the rest of the constraints refer to the proof in **Getfield** as they are essentially the same (the constraint for the region, handler's existence / non-existence, and constraint against $\vec{k_r}$ on uncaught exception).

- **Invokevirtual**
  There will be three different cases for this instruction; the first case is when method invocation executes normally. According to the translation scheme, the object reference will be put in $\vec{p}[0]$ and the rest of the parameters are arranged to match the arguments to the method call. This way, we will have the correspondence that $rt(\vec{p}[0]) = k$, and $\forall i \in [0, \mathbf{length}(st_1) - 1].\vec{p}[i+1] = st_1[i]$. Since the policies and $se$ are translated directly, we will have $rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h$ since we know that the original JVM instruction satisfies $k \sqcup k_h \sqcup se(i) \leq k'_h$. We also know that $rt(\vec{p}[0]) \leq \vec{k'_a}[0]$ since $k \leq \vec{k'_a}[0]$. A similar argument applies to the rest of parameters to the method call to establish that $\forall i \in [1, \mathbf{length}(st_1) - 1].\vec{p}[i] \leq \vec{k'_a}[i]$ that in turn will give us $\forall 0 \leq i \leq n.rt(\vec{p}[i]) \leq \vec{k'_a}[i]$. For the last constraint, we know that **excAnalysis** also gets translated directly, thus yielding the same $k_e$ for both JVM and DEX. Following the argument of **Getfield** for the region constraint, we only need to make sure that $rt(\vec{p}[0]) \sqcup k_e = k \sqcup k_e$ which is the case in our setting. Therefore, we will have that constraint $\forall j \in \mathbf{region}(i, \text{Norm}).rt(\vec{p}[0]) \sqcup k_e \leq se(j)$ is satisfied.

  The second case is when method invocation throws a caught exception. Basically, the same arguments as that of normal execution, except that the region condition is based upon particular exception $\forall j \in \mathbf{region}(i, e)$. $rt(\vec{p}[0]) \sqcup k'_r[e] \leq se(j)$. Since the policy stays the same, JVM instruction satisfies this constraint will imply that the DEX instruction will also satisfy the constraint. Since now the method is throwing an exception, we also need to make sure that it is within the possible thrown exception defined in **excAnalysis**. Again as the class stays the same and that **excAnalysis** is the same, the satisfaction of $e \in \mathbf{excAnalysis}(m_{\text{ID}}) \sqcup \{\text{np}\}$ in the JVM side implies the satisfaction of $e \in \mathbf{excAnalysis}(m') \sqcup \{\text{np}\}$ in the DEX side.

  The last case is when method invocation threw an uncaught exception. Same argument as the caught exception with the addition that escaping exception is contained within the method's policy. Since we have $k \sqcup se(i) \sqcup \vec{k'_r}(e) \leq \vec{k_r}(e)$ in the JVM side, it will also imply that $rt(\vec{p}[0]) \sqcup se(i) \sqcup \vec{k'_r}(e) \leq \vec{k_r}(e)$ in the DEX side since $rt(\vec{p}[0]) = k$ and everything else is the same.

  Actually, there is a possibility that there is an addition of **moveresult** and/or **moveexception**, except that the target of this instruction will be in the stack space. Therefore there will be no constraint involved to satisfy.

- **Throw**
  We use similar arguments to that of **Invokevirtual** addressing the similar form of the constraints. In the case of caught exception case, the constraint $e \in \mathbf{classAnalysis}(i) \cup \{\text{np}\}$ is satisfied because, as before, **classAnalysis** and classes ($e$) are the same. So, if the JVM program satisfies the constraint, the translated DEX program will also satisfy it. The same with $\forall j \in \mathbf{region}(i, e) rt(r) \leq se(j)$ since $rt(r) = k$.

  The case where an exception is uncaught is the same as the caught case, with

an addition that the security level of the thrown exception must be contained within method's policy. In this case, we already have $rt(r) \leq \vec{k}_r(e)$ since $rt(r) = k$ and policies stay the same.

□

Using the above lemmas, we can prove the lemma that all the resulting blocks will also be typable in DEX.

**Lemma 6.2.11** (Translation Soundness)**.** *Let $P$ be a JVM program s.t.*

$$\forall i, j.i \mapsto j.\exists st.i \vdash S_i \Rightarrow st \quad \textit{and} \quad st \sqsubseteq S_j$$

*Then $[\![P]\!]$ will satisfy*

1. *for all blocks $bi, bj$ s.t. $bi \mapsto bj$, $\exists rt_b$. s.t. $RTs_{bi} \Rightarrow^* rt_b, rt_b \sqsubseteq RTs_{bj}$; and*
2. *$\forall bi, i, j \in bi$. s.t. $(bi, i) \mapsto (bi, j).\exists rt$. s.t. $(bi, i) \vdash RT_{(bi,i)} \Rightarrow rt, rt \sqsubseteq RT_{(bi,j)}$*

*where*

$$\begin{array}{llllll} RTs_{bi} = [\![S_i]\!] & \textit{with} & [\![i]\!] = (bi, 0) & , & RTs_{bj} = [\![S_j]\!] & \textit{with} \quad [\![j]\!] = (bj, 0), \\ RT_{(bi,i)} = [\![S_{i'}]\!] & \textit{with} & [\![i']\!] = (bi, i) & , & RT_{(bi,j)} = [\![S_{j'}]\!] & \textit{with} \quad [\![j']\!] = (bj, j). \end{array}$$

*Proof.* For the first property, they are mainly proved using Lemma 6.2.9 because we know that if a DEX instruction is at the end of a block, it is the last instruction in its translated JVM instruction, except for **invoke** and throwing instructions. Based on Lemma 6.2.9, we have that $rt \sqsubseteq RT_{(bj,0)}$, where $RT_{(bj,0)} = [\![S_j]\!]$. Since by definition $rt_b$ is such $rt$ and $RTs_{bj} = RT_{(bj,0)}$, the property holds. For **invoke** we use the first case of Lemma 6.2.7, and for throwing instructions, we use the first case of Lemma 6.2.8.

For the second property, it is only possible if the DEX instruction at address $i$ is non-invoke and non-throwing instruction. There are two possible cases here, whether $i$ and $j$ come from the same JVM instruction or not. If $i$ and $j$ come from the same JVM instruction, then we use the first case of Lemma 6.2.7. Otherwise, we use Lemma 6.2.9. □

After we established that the translation into DEX instructions in the form of blocks preserves typability, we also need to ensure that the next phases in the translation process also preserves typability. The next phases are ordering the blocks, output the DEX code, then fix the branching targets. Before we proceed to the proof of Lemma 6.2.12, we define a property which is satisfied after the ordering and output phase.

**Property 6.2.8.** For any block whose next order is not its primary successor, there are two possible cases. If the ending instruction is not **ifeq**, then there will be a **goto** instruction appended after the output of that particular block during the output phase. If the ending instruction is **ifeq**, check whether the next order is, in fact, the second branch. If it is the second branch, then we need to "swap" the **ifeq** instruction into **ifneq** instruction. Otherwise appends **goto** to the primary successor block.

**Lemma 6.2.12** (Order and Output Soundness)**.** *Let* $\llbracket P \rrbracket$ *be the typable basic blocks resulting from translation of JVM instructions still in the block form, i.e.*

$$\llbracket P \rrbracket = \textbf{Translate}(\textbf{TraceParentChild}(\textbf{StartBlock}(P))).$$

*Given the ordering scheme to output the block contained in* **PickOrder** *then the output* $[\![P]\!]$ *is also typable.*

*Proof.* The proof of this lemma is straightforward based on the definition of the property and typability. Assuming that initially, we have the blocks already typable, then what is left is to ensure that the successor relationship of the DEX instructions is preserved in the output as well. Since the output is based on the ordering, and the property ensures that for any ordering, all the block will have a correct successor, then the typability of the program is preserved.

To flesh out the proof, we go for each possible ending of a block and its output after the output phase.

- Sequential instruction
  There are two possible cases; the first case is that the successor block is the next block in order. Let $bi$ indicate the current block and $bj$ the successor block in question. Let $i_n$ be the last instruction in $bi$, then we know that $\exists rt.RT_{(bi,i_n)} \Rightarrow rt, rt \sqsubseteq RTs_{bj}$ where $RTs_{bj}$ will be the register typing for the next instruction (in other words $RT_{(bj,0)}$). Therefore, the typability property trivially holds.

  The second case is that the successor block is not the next block in order. According to the step performed in the **Output** phase, the property 6.2.8 will be satisfied. Thus there will be a **goto** appended after instructions in the block output targetting the successor block. Let such block be $bi$ and the successor block $bj$. Let $i_n$ be the last instruction in $bi$. From the definition of typability, we know that if $bj$ is the next block to output, then $\exists rt.RT_{(bi,i_n)} \Rightarrow rt, rt \sqsubseteq RTs_{bj}$. Now with additional **goto** in the horizon, we appeal to the transfer rule to establish that this instruction does not need to modify the register typing, i.e. $\exists rt.(bi, i_n) \vdash RT_{(bi,i_n)} \Rightarrow RT_{(bi,i_n+1)}, (bi, i_n + 1) \vdash RT_{(bi,i_n+1)} \Rightarrow rt, rt \sqsubseteq RTs_{bj}$ where $RT_{(bi,i_n+1)} = rt$.

- **ifeq**
  There are three possible cases here; the first case is that the next block to output is its primary successor. It is trivial as the relationship is preserved in that the next block to output is the primary successor.

  The next case is that the next block to output is its secondary successor. We switch the instruction to its complement, i.e., **ifneq**. Let $bi$ be the current block, $bj$ be the primary successor (which is directly placed after this block), and $bk$ the other successor. Let $i_n$ be the index to the last instruction in $bi$. If $bi$ ends with **ifneq**, then we know that it is originally from the instruction **ifeq** since it is impossible to explicitly code **ifneq** in the JVM. We also know that the blocks are typable, therefore we have that for the two successors of $bi$ the following

relation holds: $\exists rt_1.i_n \Rightarrow rt_1, rt_1 \sqsubseteq RTs_{bj}$ and $\exists rt_2.i_n \Rightarrow rt_2, rt_2 \sqsubseteq RTs_{bk}$, which defines the typability for the output instructions.

The last case is when the next block to output is not its successor. The argument is the same as the sequential instruction one, where we know that adding **goto** can maintain the register typing thus preserving the typability by fixing the successor relationship.

For the secondary successor (target of branching), we know that there is a step in the output that handles the branch addressing to maintain the successor relations.

- **invoke**, yet the next block to output is not **moveresult**
  Although superficially this seems like a possibility, the fact that **moveresult** is added corresponding to a unique **invoke** renders the case impossible. If the block containing sole **moveresult** is not yet ordered, we know that it will be the next block to output based on the ordering scheme. This is the only way that a **moveresult** can be given an order, so it is impossible to order a **moveresult** before ordering its unique **invoke**.

□

Finally, the main result of this paper in that the compilation of typable JVM bytecode will yield typable DEX bytecode which can be proved from Lemma 6.2.11 and Lemma 6.2.12. Typable DEX bytecode will also have the non-interferent property because it is based on a safe CDR (Lemma 6.2.5) according to DEX.

**Theorem 6.2.1** (Compilation Soundness)**.** *If $P$ is a typable JVM bytecode w.r.t. its safe CDR (***region**, **jun***), and method policies $\Gamma$, then according to the translation scheme $[\![P]\!]$ has the property that*

- $\forall i, j \in PP_{\text{DEX}}$. *if* $i \mapsto j$. *then* $\exists rt.i \vdash RT_i \Rightarrow rt, rt \sqsubseteq RT_j$
- $\forall i, j \in PP_{\text{DEX}}$. *if* $i \mapsto .$ *then* $i \vdash RT_i \Rightarrow$

*w.r.t. a safe CDR (*$[\![$**region**$]\!]$, $[\![$**jun**$]\!]$*).*

## 6.3 Implementation for Type-Preserving Compilation

Figure 6.1 shows the overall architecture of our work. There are a lot of components already fixed in place when we are talking about the compilation from Java source to Android program. The connection between Java and JVM type systems for non-interference has been studied by Barthe et al. [2006a]. Nevertheless we did not include Java source code itself as one of the components as we are more concerned about the Java classes compiled from Java source code (JVM Bytecode). In this setting, our contributions are highlighted and set in a bolded frame.

The whole Android compilation setting starts from a developer writing the application in Java. The components of the application will then get compiled to Java

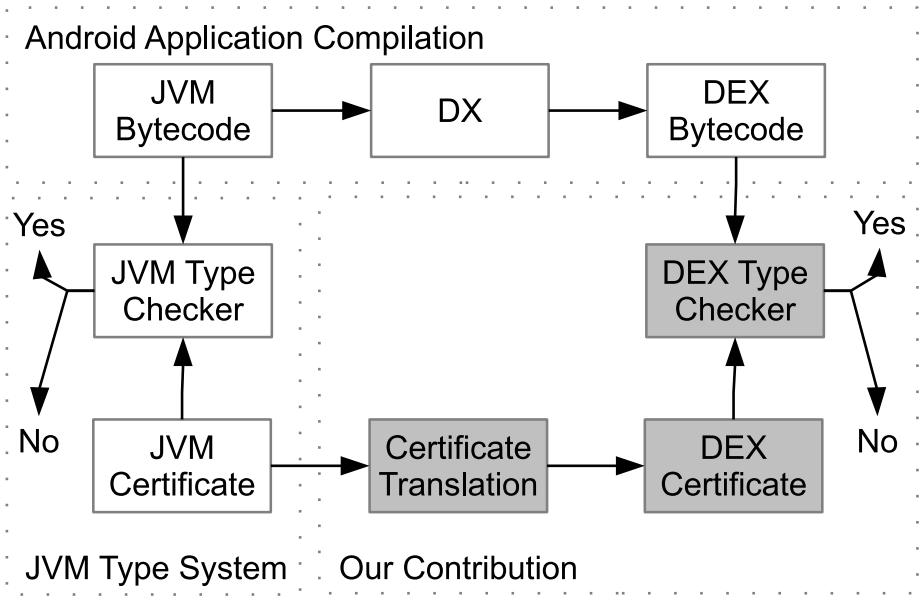



Figure 6.1: Overall architecture

classes. dx tool which is bundled with the Android SDK will then aggregate these classes into one (possibly many) DEX file, usually named "classes.dex". The rest of the toolchain will also bundle the manifest file, other resources, and assets into an APK. This APK is what we call an Android application, which can be installed on a mobile phone running Android OS.

The work by Barthe et al. comes into the picture in providing a type system to guarantee that a JVM bytecode is non-interferent. Crucial to the work are the components that check whether a given JVM bytecode is typable, and produce a certificate if it is (type inference). We have implemented our own JVM type inference since we do not have a type inference that suits our need. There is then another component which type checks whether a JVM bytecode matches its certificate. Since our aim is the translation of a certificate, we do not include the type inference component in the figure. Nevertheless, this component is crucial in providing us with the certificate for JVM bytecode which is necessary for the whole system to even start. We will detail this component in the next section.

Our contributions are mainly contained in two components. The first component, certificate translation, basically takes the certificates for the source Java classes and translates it into a certificate for the corresponding DEX file, independently of the non-optimizing compilation done by the dx tool. The other components parse the DEX file, take the certificate, and check whether they match. We will detail this component in the next section as well.

The prototype for our implementation is tested functionally, i.e., we write some apps without some of the capabilities that are not included in the discussion (e.g., monitor and filled-new-array) which are then compiled with the dx tool with the no-optimize flag. We then take the intermediate Java classes and feed them into our

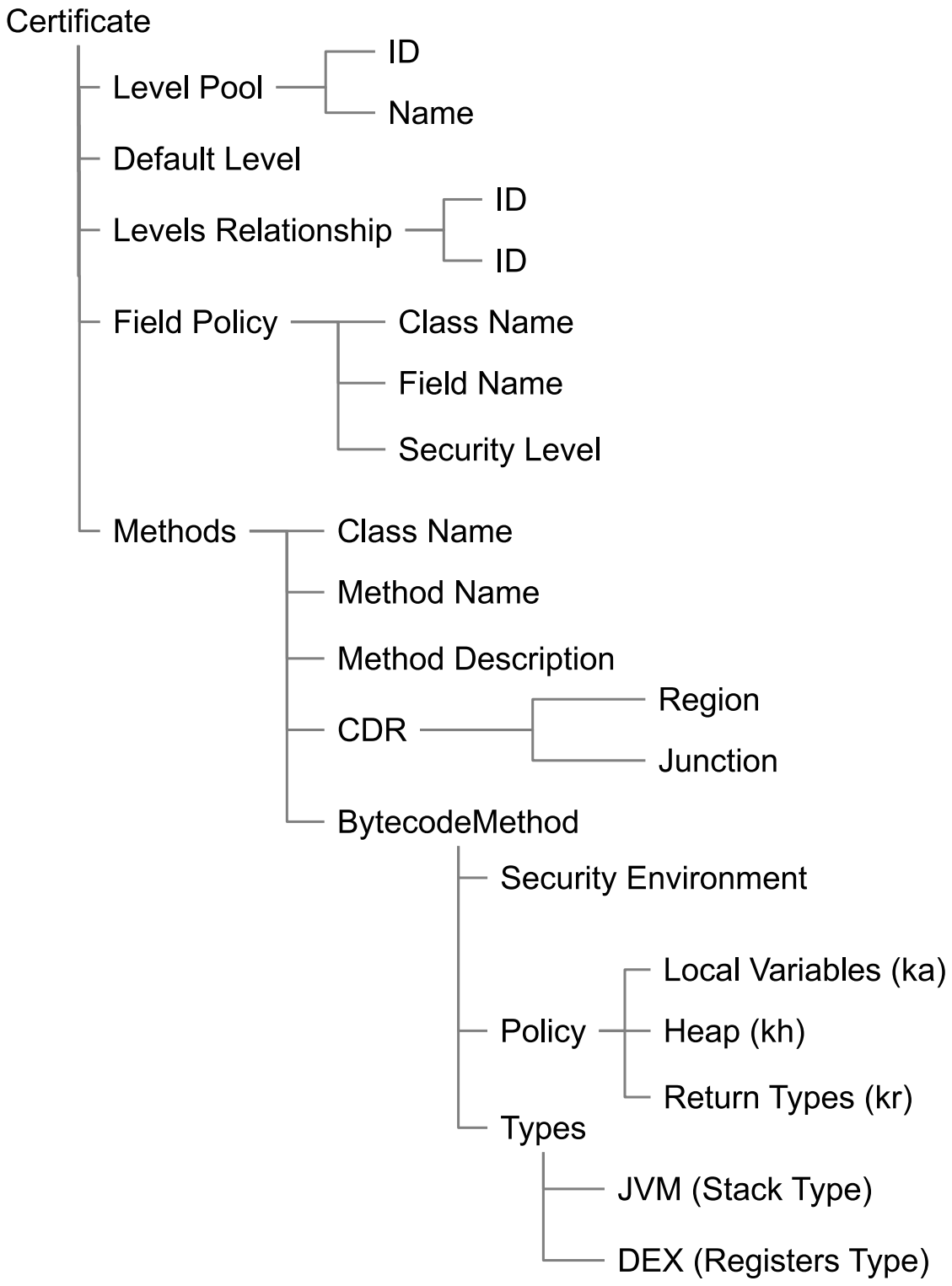


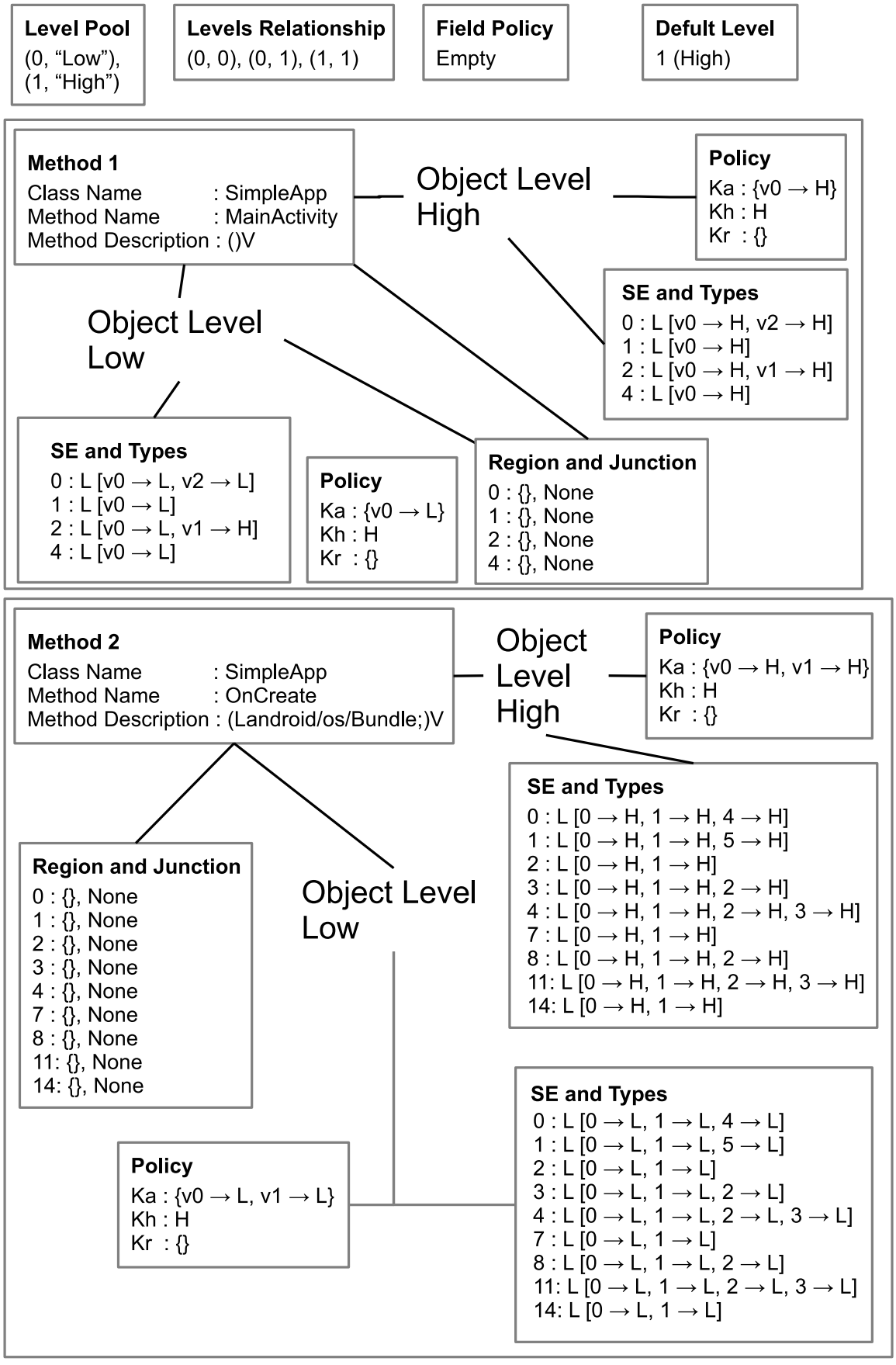

Figure 6.2: Certificate Structure

```
MainActivity()
0 : move-object v0, v2
1 : move-object v1, v0
2 : invoke-direct {v1}, method@27f3
4 : return-void
```

```
OnCreate(Landroid/os/Bundle;)
0 : move-object v0, v4
1 : move-object v1, v5
2 : move-object v2, v0
3 : move-object v3, v1
4 : invoke-super {v2, v3}, method@27fe
7 : move-object v2, v0
8 : const v3, #7f040019
11: invoke-virtual {v2, v3}, method@3ba5
14: return-void
```

```
btnCalculate_click(Landroid/view/View;)
0 : move-object v0, v9
1 : move-object v1, v10
2 : move-object v6, v0
3 : const v7, #7f0c0053
6 : invoke-virtual {v6, v7}, method@3ba3
9: move-result-object v6
10: check-cast v6, type@709
12: move-object v2, v6
13: move-object v6, v0
14: const v7, #f0c0054
17: invoke-virtual {v6, v7}, method@3ba3
20: move-result-object v6
21: check-cast v6, type@709
23: move-object v3, v6
24: move-object v6, v0
25: const v7, #7f0c0050
28: invoke-virtual {v6, v7}, method@3ba3
31: move-result-object v6
32: check-cast v6, @type732
34: move-object v4, v6
35: new-instance v6, type@77c
37: move-object v8, v6
38: move-object v6, v8
39: move-object v7, v8
40: invoke-direct {v7}, method@3c5a
43: move-object v7, v2
44: invoke-virtual {v7}, method@3c3c
47: move-result-object v7
48: invoke-virtual {v7}, method@3c3c
51: move-result-object v7
52: invoke-virtual {v6, v7}, method@3c63
55: move-result-object v6
56: move-object v7, v3
57: invoke-virtual {v7}, method@3a1c
60: move-result-object v7
61: invoke-virtual {v7}, method@3c3c
64: move-result-object v7
65: invoke-virtual {v6, v7}, method@3c63
68: move-result-object v6
69: invoke-virtual {v6}, method@3c68
72: move-result-object v6
73: move-object v5, v6
74: move-object v6, v4
75: move-object v7, v5
76: invoke-virtual {v6, v7}, method@3b94
79: return-void
```

Figure 6.3: Content of the Methods for the Sample Application


Level Pool
(0, "Low"),
(1, "High")
Levels Relationship
(0, 0), (0, 1), (1, 1)
Field Policy
Empty
Defult Level
1 (High)
Method 1
Class Name : SimpleApp
Method Name : MainActivity
Method Description : ()V
Object Level High
Policy
Ka : {v0 → H}
Kh : H
Kr : {}
SE and Types
0 : L [v0 → H, v2 → H]
1 : L [v0 → H]
2 : L [v0 → H, v1 → H]
4 : L [v0 → H]
Object Level Low
SE and Types
0 : L [v0 → L, v2 → L]
1 : L [v0 → L]
2 : L [v0 → L, v1 → H]
4 : L [v0 → L]
Policy
Ka : {v0 → L}
Kh : H
Kr : {}
Region and Junction
0 : {}, None
1 : {}, None
2 : {}, None
4 : {}, None
Method 2
Class Name : SimpleApp
Method Name : OnCreate
Method Description : (Landroid/os/Bundle;)V
Object Level High
Policy
Ka : {v0 → H, v1 → H}
Kh : H
Kr : {}
SE and Types
0 : L [0 → H, 1 → H, 4 → H]
1 : L [0 → H, 1 → H, 5 → H]
2 : L [0 → H, 1 → H]
3 : L [0 → H, 1 → H, 2 → H]
4 : L [0 → H, 1 → H, 2 → H, 3 → H]
7 : L [0 → H, 1 → H]
8 : L [0 → H, 1 → H, 2 → H]
11: L [0 → H, 1 → H, 2 → H, 3 → H]
14: L [0 → H, 1 → H]
Region and Junction
0 : {}, None
1 : {}, None
2 : {}, None
3 : {}, None
4 : {}, None
7 : {}, None
8 : {}, None
11: {}, None
14: {}, None
Object Level Low
Policy
Ka : {v0 → L, v1 → L}
Kh : H
Kr : {}
SE and Types
0 : L [0 → L, 1 → L, 4 → L]
1 : L [0 → L, 1 → L, 5 → L]
2 : L [0 → L, 1 → L]
3 : L [0 → L, 1 → L, 2 → L]
4 : L [0 → L, 1 → L, 2 → L, 3 → L]
7 : L [0 → L, 1 → L]
8 : L [0 → L, 1 → L, 2 → L]
11: L [0 → L, 1 → L, 2 → L, 3 → L]
14: L [0 → L, 1 → L]

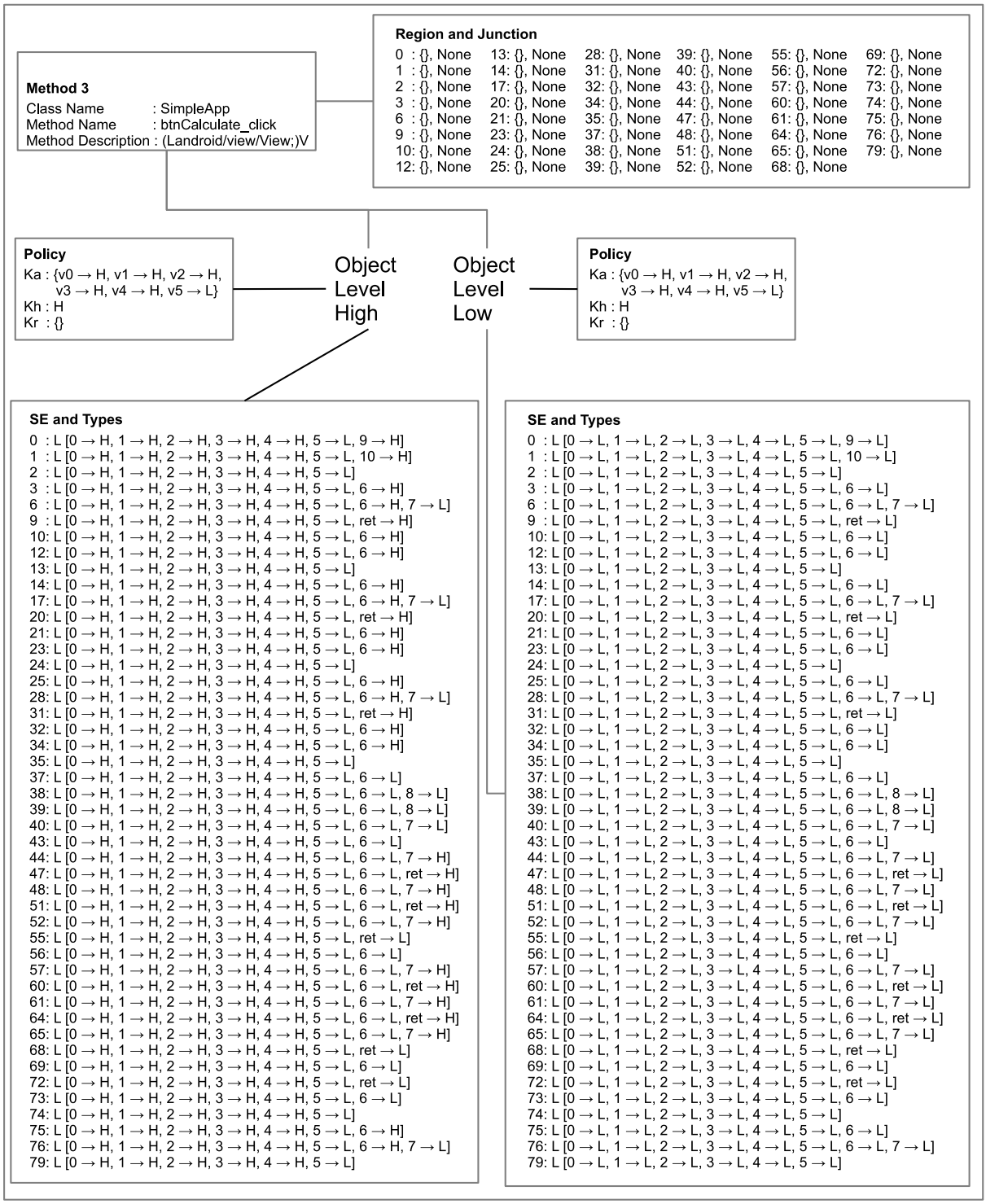



Figure 6.4: Certificate for the Sample Application

JVM type inference tool, which then provides the certificates for the Java classes. The output certificates are further processed by the certificate translation tool to produce the certificate for DEX, which then inserted into the APK. Since this is only a proof of concept, we did not make an effort to do a thorough testing.

### 6.3.1 Component Details

In this section, we will go into more depth about each of the components that provide us with the setting of this experimentation.

#### 6.3.1.1 Certificate Structure

Here we will describe how we structure our certificate, both for JVM and DEX. They mainly differ in that JVM uses a stack type while DEX uses a register type. Figure 6.2 shows the high-level structure of the certificate.

For the proof of concept, we implement a simple Android application with two text input and one button, which will concatenate the strings in the two boxes. The application itself has three main methods, a method for initialization, the method to handle button click, and a method to handle application creation. The rest are methods added by default. We will focus on this three methods and its certificate. Figure 6.3 shows the excerpt from the DEX file. For this particular application, its certificate is shown in Figure 6.4. Note that in this sample certificate we do not show the entirety of the register typing and only show those that may be of interest. The rest of the register is filled with a default level.

Below, we describe how we represent the certificate as a file. Since we will infer the type for JVM, we do not provide the instruction typing for JVM certificate, just instruction typing for DEX. We used the notation u*n* to denote *n* bytes of data for each type. We first describe how we represent a string and extended security level; then we describe the certificate and its details.

```
string {
        u1      string_size;
        u1      char[string_size];
}
```

**string_size:** the value of this item is equal to the number of characters in the string. The index to char is valid if it is greater than zero and less than string_size. The string does not have to be ended by '\0'.

**char:** this item contains a list of characters in the order of their appearance in the string. Each character is represented by its ASCII which is in the range of 0-255.

A sample of string in this format is "03 4C 6F 77" to represent "Low", where the first byte is the length of the string, "4C 6F 77" is the 3 byte ASCII value for "Low".

```
security_level {
        u1      level_depth;
        u1      security_level_ID[level_depth];
}
```

**level_depth:** the value of this item is equal to the depth of the extended security level. The depth is valid if it is greater than zero because zero is reserved for a security level of $\perp$ which should never exist in a valid certificate except for pseudo-register *ret*.

**security_level_ID:** this item constitutes a representation of an extended level, in the form of bytes in the order they appear in the security level. A security level with depth 1 denotes a simple security level (is an element of $\mathcal{S}$) whereas depth more than 1 represents extended security level ($\mathcal{S}^{\text{ext}}$). A depth of more than one will have the following form: depth of 2 will be $b_1[b_2]$, depth of 3 will be $b_1[b_2[b_3]]$, and so on, where $b_n$ indicates the $n$-th byte.

A sample security level for this (referring to the sample certificate) is "01 00" to represent a simple low security level. A value of "02 00 01" will represent an extended security level "Low[High]".

```
Certificate {
        u1              level_pool_size;
        string          level_pool[level_pool_size];
        u1              default_level;
        u1              levels_relationship_size;
        u1*u1           levels_relationship[levels_relationship_size];
        u2              field_size;
        field_info      fields[field_size];
        u2              methods_size;
        method_info     methods[method_size];
}
```

The structure of the certificate is the following :

**level_pool_size:** the value of this item is equal to the number of entries in the level_pool. The index to level_pool is valid if it is greater or equal than zero and less than level_pool_size

**level_pool:** this item contains a list of the name of the security levels. The security level itself will be represented as a pair of ID and its name. The list of names here is expected to follow the order of the ID of security level, i.e., the first item in the list will have ID 0, the second item will have ID 1, and so on.

**default_level:** the value of this item will denote the lowest level in the level_pool.

**levels_relationship_size:** the value of this item is equal to the number of entries in the levels_relationship.

**levels_relationship:** this item contains a list of a pair of ID $x, y$ to denote that $x \leq y$. The value in this item does not have to be sorted because they are represented as a relation map.

**field_size:** the value of this item is equal to the number of entries in the fields.

**fields:** this item contains the global security policy for the field ($ft$). Each field is represented as a string of class name, a string of field name, and a security_level_info of security level. The list of this fields does not have to be sorted according to some order because we represent the policy as a map from a pair of strings to its security level.

**methods_size:** the value of this item is equal to the number of entries in the methods.

**methods:** this item contains methods (the structure of each method is described in method_info), where each item is a string of class name, a string of method name, a string of method description, and the content of the method. The entries in this item do not have to be sorted as they will be represented as a map from a triple of string to a method_info structure

For our particular example, the level pool size will have the value of "02" because there are only two security level ("Low" and "High"). The level pool itself will contain the two strings represented with the byte value "03 4C 6F 77 04 48 69 67 68", where "03 4C 6F 77" represents "Low" and "04 48 69 67 68" represents "High". The default level has a byte value of "01", the levels relationship size is "03", and the content itself will be "00 00 00 01 01 01" representing the three elements "(0,0), (0,1), (1,1)". The field size will be "00 00" because we do not have any fields in our program, and the method size will be "00 03" because there are three methods we are concerned about. For each of the fields and methods, the structure will be detailed further.

```
field_info {
        string          class_name;
        string          field_name;
        security_level  field_policy;
}
```

This entry basically says that the global policy ($ft$) for this particular field identified with class_name and field_name has the security level of field_policy. The structure of the field is the following:

**class_name:** the value of this item identifies the field with the class name. There are no rules regarding what should be put here. What we need to be concerned with is that if the class name does not match with the one in the original program, then the program and its certificate will not type check.

**field_name:** the value of this item identifies the field with its particular name. There are no rules regarding what should be put here. What we need to be concerned with is that if the field name does not match with the one in the original program, then the program and its certificate will not type check.

**field_policy:** this item will have the structure of security_level and will indicate the extended security level assigned to this particular field.

We do not refer to the sample certificate for this field item since the sample application does not have a field. For an example of this field, assume that we have a field "objectField" defined for a class "objectClass". Assume also that the field policy for "objectClass.objectField" is high. Then the entry in the certificate will be "0B 6F 62 6A 65 63 74 43 6C 61 73 73 0B 6F 62 6A 65 63 74 46 69 65 6C 64 01 01" where "0B 6F 62 6A 65 63 74 43 6C 61 73 73" represents the string "objectClass", "0B 6F 62 6A 65 63 74 46 69 65 6C 64" represents the string "objectField" and "01 01" represent the high security level.

```
method_info {
        string                  class_name;
        string                  method_name;
        string                  method_description;
        u2                      instruction_size;
        CDR_info                CDR[instruction_size];
        bytecodeMethod_info     bytecodeMethod[level_pool_size];
}
```

For this particular method identified with the class_name, method_name, and method_description, this entry basically describes its CDR structure and policy and types of bytecodeMethod for each level in the level_pool. The structure of the method is the following:

**class_name:** the value of this item identifies the field with the class name. There are no rules regarding what should be put here. What we need to be concerned with is that if the class name does not match with the one in the original program, then the program and its certificate will not type check.

**method_name:** the value of this item identifies the method with its particular name. There are no rules regarding what should be put here. What we need to be concerned with is that if the method name does not match with the one in the original program, then the program and its certificate will not type check.

**method_description:** the value of this item identifies the method with its particular name. There are no rules regarding what should be put here. What we need to be concerned with is that if the method description does not match with the one in the original program, then the program and its certificate will not type check.

**instruction_size:** the value of this item identifies how many instructions are there in the method, mainly to identify how many CDRs we are expecting. A note need to be made that this instruction_size can be different from the one in the bytecodeMethod to cater for default methods that do not need a detailed instructions typing.

**CDR:** Each element this item will have the structure of CDR_info, which will indicate the labels, regions, and junctions for this method. The elements of this CDR does not have to be ordered according to the labels because we represent the CDR information as a mapping from program points to regions or junction.

**bytecodeMethod:** this item will contain some policies and typings according to the number of security levels there are in the level pool. The main purpose of this different bytecodeMethod is to cater for the $\Gamma$ policy which can be different according to the security level of the object that the method is invoked from. For the program to type check, all the methods (and all security level of the object) need to be type-checked.

An example for this entry is "09 53 69 6D 70 6C 65 41 70 70 06 3C 69 6E 69 74 3E 01 56 00 04 ..." where "09 53 69 6D 70 6C 65 41 70 70" represents the string "SimpleApp" "06 3C 69 6E 69 74 3E" represents the string "<init>", "01 56" represents the string "V", and "00 04" is the instruction size. The CDR structure and the content of the bytecode method will be detailed further.

```
CDR_info {
        u2      program_label
        u2      region_size;
        u2      region[region_size];
        u2      junction;
}
```

The structure of the CDR is the following:

**program_label:** the value of this item indicates the label of the program for which has the region information and junction.

**region_size:** the value of this item is equal the number of entries in region. A valid value for this region is greater or equal to zero and less than instruction_size.

**region:** the value of this item shows which program labels are under the region of program_label, it may be empty.

**junction:** the value of this item shows which program point serves as the junction of program_label. This item has the value of zero if there is no such junction for program_label.

For each of the program point in the method, we attach an entry for the CDR. An example of a CDR entry for program point 1 where $\mathbf{region}(1) = \{2, 4, 5\}$ and $\mathbf{jun}(1) =$

7 is “00 01 00 03 00 02 00 04 00 05 00 07” where “00 01” is the program point, “00 03” represents the number of program points in the region, “00 02 00 04 00 05” represents the 3 program points in the region of 1, and “00 07” represents the junction point of 1 which is 7. If the program point does not have any region, then the region size will simply have the value of zero.

```
bytecodeMethod_info {
        u1                      object_level;
        u1                      ka_size;
        security_level          ka[ka_size];
        u1                      kh;
        u1                      kr_size;
        security_level          kr[kr_size];
        u2                      bm_instruction_size;
        bm_instruction_info     instructions[bm_instruction_size];
}
```

The structure of the bytecodeMethod_info is the following:

**object_level:** the value of this item indicates the security of the object level that invokes this particular method.

**ka_size:** the value of this item denotes how many locals variable are there. Even though there is no such thing as local variable in the DEX itself, this value is carried over from the JVM certificate where there is a concept of local variable.

**ka:** this item will denote the security level for each of the local variables. The order in this item matters, i.e., the first security_level denotes the security level of $\vec{k_a}[0]$, the second security_level denotes the security level of $\vec{k_a}[1]$, and so on.

**kh:** the value of this item refers to the level_pool, which will denote the method policy for heap.

**kr_size:** the value of this item denotes how many return types are there. On a normal method without exception returning a value, the value will be 1. The value of kr_size is the 1 + the number of exceptions possibly thrown by the method. At the current implementation, we have not implemented exception handling yet, so the value will either be 0 or 1.

**kr:** this item will denote the security level for each of the return type. We reserve kr(0) to be the security level of the normal return value.

**bm_instruction_size:** the value of this item indicates the number of instructions in the method. This value will be either 0 or the same as instruction_size in the method structure.

**instructions:** this item describes the registers type and security environment for each instruction in the method.

A sample of this entry from the sample program for method "<init>" and has low object security level is "00 01 00 00 01 00 01 00 00 04 ..." where "00" signify the low object security level, "01" represents the size of $\vec{k_a}$, "00 00 01 00" represents a low security level ("01 00") for register 0 ("00 00"). "00 04" represents the size of the instruction, and then for each of the instruction, the structure is detailed in bm_instruction_info.

```
bm_instruction_info {
        u2                  program_label;
        u1                  security_environment;
        u2                  register_type_size;
        u2*security_level   register_level[register_type_size];
        security_level      ret;
}
```

The structure of the instruction_info is the following:

**program_label:** the value of this item indicates the program label which will be typed by the register type and the security environment.

**security_environment:** the value of this item will refer to a basic security level contained in the level_pool. The valid value will be greater or equal than 0 and less than level_pool_size.

**register_type_size:** the value of this item indicates how many registers are there in the register type for the current program label.

**register_level:** this item describes the security level of the value held by a particular register. This item is represented as a pair of register number and its security level, and we represent them directly as a map.

**ret:** the value of this item will indicate what is the security level of the pseudo-register *ret*. In the case where it is a $\perp$, we can say that *ret* does not hold any value

An example for this entry is "00 02 00 00 03 00 02 01 01 00 01 01 01 00 00 01 01 00" where "00 02" is the program point 2, "00" is the low security level for *se*, "00 03" represents the size of the register typing, "00 02 01 01" means register 2 has high security level, "00 01 01 01" means register 1 has high security level, "00 00 01 01" means register 0 has high security level, and the last byte "00" means the pseudo register *ret* is not used.

To collect everything together, a certificate for the sample DEX program is given in Figure 6.5.

#### 6.3.1.2 Naive JVM Type Inference

This component takes an input file which contains the certificate without the typing for each instruction (stacktype and security environment) and reconstructs the

```
02 03 4C 6F 77 04 48 69 67 68                          } Level pool
01                                                     } default level
03 00 00 00 01 01 01                                   } levels relationship
00 00                                                  } Field
00 03                                                  } Method size
09 53 69 6D 70 6C 65 41 70 70                          } Method 1
06 3C 69 6E 69 74 3E 01 56
00 04                                                  } CDR for Method 1
   00 00 00 00 00 00
   00 01 00 00 00 00
   00 02 00 00 00 00
   00 04 00 00 00 00
00 01 00 00 01 00 01 00                                } Policy for Low object level
00 04                                                  } bytecode method typing
00 00 00
   00 03                                               } register type for program point 0
   00 02 01 00
   00 01 01 01
   00 00 01 00
   00
00 01 00
   00 03                                               } register type for program point 1
   00 02 01 01
   00 01 01 01
   00 00 01 00
   00
00 02 00
   00 03                                               } register type for program point 2
   00 02 01 01
   00 01 01 01
   00 00 01 00
   00
00 04 00
   00 03                                               } register type for program point 4
   00 02 01 01
   00 01 01 01
   00 00 01 00
   00
...
the rest of the certificate is omitted
```

Figure 6.5: Sample Certificate

**Algorithm 10** Flow_Tracer($bm$)

```
order := [1]; // always start with the first instruction
workingSet := [1];
while workingSet is not empty do
  (i :: workingSet') := workingSet;
  if bm.instructionAt(i) ∈ {Nop, Push c, Load x, Store x, Binop, New, Getfield,
  Putfield, Invoke m'} then
    order.append(i + 1);
    if (i + 1) ∉ order then workingSet'.append(i + 1);
  else if bm.instructionAt(i) = Goto t then
    order.append(t);
    if t ∉ order then workingSet'.append(t);
  else if bm.instructionAt(i) ∈ {If t, Ifz t} then
    order.append(i + 1); order.append(t)
    if (i + 1) ∉ order then workingSet'.append(i + 1);
    if t ∉ order then workingSet'.append(t);
  workingSet := workingSet';
return order;
```

certificate for the JVM bytecode. The inference itself (described in Algorithm 12 is simple in nature, we just repeatedly infer the stack type and security environment for the successor instruction, starting from label 0, until they reach a fixpoint (no more change either in stack type or security environment). Algorithm 11, describes how we implement the naive type inference. A little note to be made is that we abstract from actual program counter and use the index to the list instead. We also overload the symbol $\sqsubseteq$ to be operable between stack types as well to mean pairwise lub between the stacktype elements. Before we invoke the Algorithm 11, we first do a simple program flow tracing described in Algorithm 10. The purpose of this program flow tracing is to ease the burden of type inference so that it just needs to traverse the order produced by the flow tracer without having to deal with the case where an instruction still has not been assigned a stack type and security environment yet.

We implemented this component in OCaml, in conjunction with the component to do a certificate translation. Our main consideration for doing so is because it is much easier to transfer the resulting JVM certificate directly to the next component within the program itself, rather than having a temporary output file.

#### 6.3.1.3 Non-Optimizing Certificate Translation

The translation component basically implements what we have highlighted before in the translation section. We implemented the five steps in translating the JVM bytecode into DEX bytecode while carrying over the type from JVM for each instruction. The translation of the stack type also follows in that we just flatten the stack type and assign an index (starting from the number of local variables) from the bottom of the stack. The details for the implementation itself is straightforward if it just

concerns the translation of the bytecode. When we translate the type, however, some complications arise.

Firstly, since the translation steps only concern a specific bytecode under a specific policy, we have to extend that to include each security level for the object that contains this method. We then have to incorporate all the methods in the class. Since dx aggregates all the files together, we also have to extend the scope to include all the classes to be compiled to DEX bytecode. This peripheral information is not necessarily difficult in itself because it is mostly straightforward. For example, the policy for the field and methods can be directly transferred from JVM to DEX.

That said, the convention for the class identifier is different between JVM and DEX, so we need a bit of adjustment here. In particular, for JVM the class identifier is a "/" delimited string where the last string is the particular class name, and the string before that is the package name. For DEX, however, the class identifier takes the form of "L" + class identifier + ";", e.g., if a class in JVM is identified with "java/lang/Object" then in DEX it is identified as "Ljava/lang/Object;". For the current implementation, we also use the short method descriptor in DEX as opposed to the full method descriptor.

Secondly, there are some instructions that do not fit nicely with the framework of just bringing the types from the JVM. The instruction pair of **Invoke** and **MoveResult** in DEX use the pseudo-register *ret*, which obviously is not captured in JVM. For **MoveResult**, firstly we just take the stack type of the successor of the **Invoke** instruction and translate it into the register type. Then we need to check whether the Invoke instruction immediately before is returning a value or not, and if so, we remove one value from the top of the stack and put it into *ret*. The **Goto** instruction is also a tricky bit that we have to handle, in that it only ever appears in the translation phase if the next block to output is not a successor. To handle this, we maintain the register type for the start of each block and assign them to any **Goto** instruction that points to that block.

Thirdly, the translation of regions and junction are not really straightforward either. We need to know the relation between the source program points and their translation. In particular, a program point in the JVM may be related to 0 or more program points in the DEX, but each program point in the DEX will only have one relation to the JVM program points (one to many) with the exception of a **Goto** instruction. The simplest part of the region translation is that for a program point $i$, all the program points in the DEX that are related to any program points in $\mathbf{region}(i)$ will also be in $\mathbf{region}(j)$ where $j$ is a DEX program point that is related to $i$. Again, **MoveResult** and **Goto** complicate this process. For **MoveResult**, because it does not have its corresponding instruction in the JVM we have to fix it so that it relates to its associated **Invoke** instruction in the JVM. A similar case can be made with the **Goto** instruction in that we tag the source instruction for **Goto** to be the same as the start of the block which becomes the target of this **Goto** instruction. This decision of relating the **Goto** instruction with the target instead of the previous instruction is mainly to handle the case where the **Goto** instruction is generated by a branching instruction, in which case we need to include **Goto** in the region as well.

Finally, there are also some instructions that do not directly follow the translation outlined above. In the JVM, all **Binop** instructions use the values on the stack as the operands, regardless of whether or not they are constants. In the DEX, **BinopConst** is dedicated to deal with a binary operation with constant. The modifications to the values themselves will not change, but it has impact on the size of the instructions, i.e., the size of **Binop** instruction can be one short or two shorts, but the size of **BinopConst** is always two shorts. The DEX translation of **Dup** is also not straightforward. Instead of copying the value from the top of the stack and push it on top, it copies the value on top of the stack, creates a register containing the copy, and then copies the value from this register to both the top of the stack and the new top of the stack.

#### 6.3.1.4 DEX Type Checker

This last component primarily just does a simple type checking on the Android phone. It takes the DEX file, parses the DEX file, and checks it against the certificate generated by the certificate translator whether all the instructions in the bytecode satisfy the typability definition. This means that our system works independently of the Android OS, and as such, there is no modification at all to the overall Android structure.

We implemented the DEX type checker as an Android application which takes an input string from the user which is the package name for the application that the user wants to type check. Several notes need to be made for this component:

- The DEX file in the package is contained in the predefined file name "classes.dex". We are aware of the possibility that there are multiple DEX files for a single application (e.g., when there are so many methods that it can not be contained within one DEX file), but we decided to ignore this because it is not the main focus of our work.

- We also take the certificate from a predefined file name "Certificate.cert" contained in the "assets" directory. The type checker will just check for this particular file for the certificate, will skip the type checking should a valid certificate not be found.

- There are a lot of generated additional methods and classes for Android application, e.g., "Build.config" and "R" (and its subclasses). Obviously, we could analyze each of those additional things and provide a proper certificate for each of them, but we decided to ignore them instead because they are not the focus of this work.

- A similar thing can be said about the methods and classes contained in the Android library. Although for this particular case, since programs will inevitably reference them, we decided to inject the certificate with these methods except that they are stripped off their bytecode instructions. This decision is mainly

due to the transfer rule of method invocation which requires the policy of the target method.

- Since we need to parse the whole DEX file, the process takes a significant amount of time

Algorithmically, Algorithm 13 describe how we implement the DEX type checker for a particular bytecode instructions under a particular policy. Since we need to type check all the methods for all the policy, we just apply this algorithm repeatedly for different methods and policy.

### 6.3.2 Compact Certificate

The certificate structure that we have given above is still within reasonable size. In the worst case, it is within polynomial size of the bytecode ($O(n^2)$). Although practically the size of this certificate is much smaller than the actual DEX bytecode because we skip default methods, the question of whether we can make the certificate much more compact is still an interesting question.

In order to make the certificate size smaller, we modify the representation of the instruction's typing because it is the bottleneck. As opposed to representing all the registers typing explicitly, we follow the idea from JVM representation of stack, in that we represent the current typing based on the previous typing and any changing values. We introduce two functions with the following symbols to represent our registers typing: $\sqcup^{cert}$ and $\oplus^{cert}$. $\sqcup^{cert}$ is a function that takes two registers typing, and produce a register typing which every register holds the least upper bound of the security level of the input registers typing. More formally,

$$\forall rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S}).rt_1 \sqcup^{cert} rt_2 = rt, \text{ where } \forall r \in rt.rt(r) = rt_1(r) \sqcup rt_2(r).$$

$\oplus^{cert}$ is a function that takes a registers typing and a pair of index and security level, and produce a registers typing with the security level of the register at index is modified to be the input security level. More formally,

$$\forall rt \in (\mathcal{R} \to \mathcal{S}), r \in rt, k \in \mathcal{S}^{\text{ext}}.rt \oplus^{cert} (r,k) = rt',$$
$$\text{where } \forall r' \in rt'.rt'(r') = \begin{cases} rt(r') & \text{if} \quad r' \neq r \\ k & \text{if} \quad r' = r \end{cases}.$$

Using these two functions, as opposed to representing the whole registers typing as a register and security level pair, we represent them using the operations between parent instructions' typing and their modifications. For example, if an instruction $i$ has two parent $j$ and $k$, where $rt_i = \{0 \mapsto H, 1 \mapsto H\}$, $rt_j = \{0 \mapsto L, 1 \mapsto L\}$, and $rt_k = \{0 \mapsto L, 1 \mapsto H\}$, then as opposed to represent the typing as explicit mapping, we represent the typing as $rt_i = (rt_j \sqcup^{cert} rt_k) \oplus^{cert} (0, H)$.

To obtain the certificate, we first take the naive certificate that has been produced and the DEX bytecode corresponding to the certificate. We scan the DEX bytecode and gather the list of parent instructions for each of the instruction. After we have

the parents list, we combine all the registers typing of the parents using the $\sqcup^{cert}$ operation. If any of the parents does not modify the result, then that particular parent is removed from the list. After we aggregated the registers typing from the parent, then we compare the aggregate with its actual registers typing. For each of the register contained in the actual registers typing that is different than the one in the aggregate, we fix the value by using $\oplus^{cert}$ operation. We argue that both the naive representation and the compact form of the representation is the same.

**Lemma 6.3.1.** *The compact form of register typing represents the same register typing as the naive representation.*

*Proof.* We first note that there are two steps in making the compact register typing; first is aggregating the parents' register typing, and then we add $+^{cert}$ for each register that is different from the actual register typing. The crux of the lemma is that the second step enforces the compact representation to be the same as the actual representation. Therefore no matter what register typing the first step give us it will be irrelevant to the proof of the lemma. We also know that the naive representation is a plain representation of the actual register typing. Therefore since both of them are representing the actual register typing, the lemma holds. □

Intuitively, what we have done so far is just shifting the burden from the size of the certificate to the type checker, i.e., the type checker has to do extra work to check the certificate. This is obvious in that before the type checker can check whether the program is typable given the certificate, the type checker now need to process both $\sqcup^{cert}$ and $\oplus^{cert}$ operations to obtain the actual registers typing before proceeding. After we get the actual registers typing, the type checker proceeds as before. So practically the only difference for the type checker itself is the additional functionality to unravel the compact certificate.

In terms of the overall certificate representation, there is not much difference with the naive representation of the certificate. The difference only lies in the structure of each instruction, in particular, the representation of the registers typing which now consists of the list of parents involved in the $\sqcup^{cert}$ operations and list of register and security level pairs involved in the $\oplus^{cert}$ operations.

```
bm_instruction_info {
        u2                  program_label;
        u1                  security_environment;
        u2                  parents_size;
        u2                  parent_label[parents_size];
        u2                  changes_size;
        u2*security_level   modifications[changes_size];
        security_level      ret;
}
```

The structure of the instruction_info is the following:

**program_label:** the value of this item indicates the program label which will be typed by the register type and the security environment.

**security_environment:** the value of this item will refer to a basic security level contained in the level_pool. The valid value will be greater or equal than 0 and less than level_pool_size.

**parents_size:** the value of this item indicates how many parents are there for the current program label. The only instruction whose parents_size is 0 is the first instruction in the method (instruction 0).

**parent_label:** the value of this item are the labels of the parent instruction. This is used to indicate which parents are involved in the $\sqcup^{cert}$ operations.

**changes_size:** the value of this item indicate how many pairs of register and security level there are which modify the registers typing. Value 0 indicates that there is no need for any modification from the aggregation of parents registers typing.

**modifications:** this item describes pairs of register and security level involved in the $\oplus^{cert}$ operations.

**ret:** the value of this item will indicate what is the security level of the pseudo-register *ret*. In the case where it is a $\perp$, we can say that *ret* does not hold any value

An example of this entry is "00 05 00 00 01 00 02 00 01 00 00 01 01 00", where "00 05" is the program point 5, "00" represents the low security environment, "00 01 00 02" represents the list of parent of the current program point involved in $\sqcup^{cert}$, which is only program point 2, "00 01 00 00 01 01" means there is one $\oplus^{cert}$ to modify the security level of register 0 to be high.

We have implemented this scheme by extending the OCaml program to produce the certificate and the type checker app. Referring to the sample application, we note that the size of the certificate is reduced to less than half, while there is a negligible additional overhead time to check the certificate. This is due to the fact that the main bulk of the process is not in type checking itself, but rather parsing the DEX file. In the sample certificate, for a particular representation of the bytecode typing for method 1, a naive representation will require 72 bytes to represent the typing while the compact representation only requires 54 bytes to represent the typing. Even in this small example it already exhibits a 25% reduction in the space. The reduction will be even more for more instructions and more registers in the typing.

The more compact certificate for the sample DEX program is given in Figure 6.6.

```
02 03 4C 6F 77 04 48 69 67 68                          } Level pool
01                                                     } default level
03 00 00 00 01 01 01                                   } levels relationship
00 00                                                  } Field
00 03                                                  } Method size
09 53 69 6D 70 6C 65 41 70 70                          } Method 1
06 3C 69 6E 69 74 3E 01 56                             } Method 1
00 04                                                  } CDR for Method 1
   00 00 00 00 00 00                                   } CDR for Method 1
   00 01 00 00 00 00                                   } CDR for Method 1
   00 02 00 00 00 00                                   } CDR for Method 1
   00 04 00 00 00 00                                   } CDR for Method 1
00 01 00 00 01 00 01 00                                } Policy for Low object level
00 04                                                  } bytecode method typing
00 00 00                                               } bytecode method typing
   00 00                                               } compact typing for program point 0
   00 03                                               } compact typing for program point 0
   00 02 01 00                                         } compact typing for program point 0
   00 01 01 01                                         } compact typing for program point 0
   00 00 01 00                                         } compact typing for program point 0
   00                                                  } compact typing for program point 0
00 01 00                                               } bytecode method typing
   00 01 00 00                                         } compact typing for program point 1
   00 01                                               } compact typing for program point 1
   00 02 01 01                                         } compact typing for program point 1
   00                                                  } compact typing for program point 1
00 02 00                                               } bytecode method typing
   00 01 00 01                                         } compact typing for program point 2
   00 00                                               } compact typing for program point 2
   00                                                  } compact typing for program point 2
00 04 00                                               } bytecode method typing
   00 01 00 02                                         } compact typing for program point 4
   00 00                                               } compact typing for program point 4
   00                                                  } compact typing for program point 4
...
the rest of the certificate is omitted
```

Figure 6.6: Sample Compact Certificate

**Algorithm 11** Type_Inference $(bm, bmc, mc, lvt, ft, order)$

$bmc.ST(1) = []$; // Initial stack type is always empty
$bmc.ST(1) = mc.default_level$; // Initial se is always Low
**for** $i = 1$ to $bm$.length **do**
  $idx := order(i)$;
  **switch** $(bm.instructionAt(idx))$
  **case** (Nop)**:**
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup bmc.ST(i)$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
  **case** (Push $c$)**:**
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup (bmc.se(i) :: bmc.ST(i))$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
  **case** (Load $x$)**:**
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup ((bmc.se(i) \sqcup lvt.ka(x)) :: bmc.ST(i))$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
  **case** (Store $x$)**:**
    $(k :: st) := bmc.ST(i)$;
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup st$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
    **if** $bmc.se(i) \sqcup k \not\leq lvt.ka(x)$ **then return** false;
  **case** (Binop)**:**
    $(k_1 :: k_2 :: st) := bmc.ST(i)$;
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup ((k_1 \sqcup k_2 \sqcup bmc.se(i)) :: st)$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
  **case** (Goto $t$)**:**
    $bmc.ST(t) := bmc.ST(t) \sqcup bmc.ST(i)$;
    $bmc.se(t) := bmc.se(t) \sqcup bmc.se(i)$;
  **case** (If $t$)**:**
    $(k_a :: k_b :: st) = bmc.ST(i)$;
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup lift(st, k_a \sqcup k_b)$;
    $bmc.ST(t) := bmc.ST(t) \sqcup lift(st, k_a \sqcup k_b)$;
    **for** $j$ in $bmc$.region(i) **do**
      $bmc.se(j) = bmc.se(j) \sqcup k_a \sqcup k_b$;
  **case** (Ifz $t$)**:**
    $(k :: st) = bmc.ST(i)$;
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup lift(st, k)$;
    $bmc.ST(t) := bmc.ST(t) \sqcup lift(st, k)$;
    **for** $j$ in $bmc$.region(i) **do**
      $bmc.se(j) = bmc.se(j) \sqcup k$;
  **case** (Return)**: continue**;
  **case** (IReturn)**:**
    $(k :: st) := bmc.ST(i)$;
    **if** $bmc.se(i) \sqcup k \not\leq lvt.kr(0)$ **then return** false;

**case** (New $c$)**:**
$bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup (bmc.se(i) :: bmc.ST(i))$;
$bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
**case** (Getfield $f$)**:**
$(k_o :: st) = bmc.ST(i)$;
$bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup ((bmc.se(i) \sqcup k_o \sqcup ft(f)) :: bmc.ST(i))$;
$bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
**case** (Putfield $(r, r_o, f)$)**:**
$(k_s :: k_o :: st) = bmc.ST(i)$;
$bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup st$;
$bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
**if** $bmc.se(i) \sqcup k_s \sqcup k_o \not\leq ft(f)$ **then return** false;
**if** $k_h \not\leq ft(f)$ **then return** false;
**case** (Invoke $m'$)**:**
$length(st_1) := nbArguments(m')$;
$(st_1 :: k :: st_2) := bmc.ST(i)$;
$lvt' := mc.find(m', k).policy$;
$bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup ((lvt'.kr(0) \sqcup bmc.se(i) \sqcup k) :: st_2)$;
$bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
**if** $k \sqcup lvt.kh \sqcup bmc.se(i) \not\leq lvt'.kh$ **then return** false
**if** $k \not\leq lvt'.ka(0)$ **then return** false
**for** $j := 1$ to $length(st_1)$ **do**
**if** $st_1[j] \not\leq lvt'.ka[j]$) **then return** false
**end for**
**return** *bmc*;

**Algorithm 12** Infer_Type($bm, bmc, mc, lvt, ft$)

*order* := Flow_Tracer($bm$);
*new_bmc* := Type_Inference($bm, bmc, mc, lvt, ft$);
**while** *bmc* <> *new_bmc* **do**
*bmc* := *new_bmc*;
*new_bmc* := Type_Inference($bm, bmc, mc, lvt, ft$);
**return** *new_bmc*;

**Algorithm 13** Type_check($bm, bmc, mc, lvt, ft$)

```
for i = 1 to bm.length do
  rt := bmc.RT(i);
  switch (bm.instructionAt(i))
  case (Nop):
    if bmc.RT(i) ⋢ bmc.RT(i+1) then return false;
  case (Const (r, c)):
    if rt ⊕ {r ↦ bmc.se(i)} ⋢ bmc.RT(i+1) then return false;
  case (Move (r, r_s)):
    if rt ⊕ {r ↦ bmc.se(i) ⊔ rt(r_s)} ⋢ bmc.RT(i+1) then return false;
  case (Binop (r, r_a, r_b)):
    if rt ⊕ {r ↦ bmc.se(i) ⊔ rt(r_a) ⊔ rt(r_b)} ⋢ bmc.RT(i+1) then return false;
  case (Binop2Addr (r_a, r_b)):
    if rt ⊕ {r_a ↦ bmc.se(i) ⊔ rt(r_a) ⊔ rt(r_b)} ⋢ bmc.RT(i+1) then return false;
  case (BinopConst (r, r_a, c)):
    if rt ⊕ {r ↦ bmc.se(i) ⊔ rt(r_a)} ⋢ bmc.RT(i+1) then return false;
  case (MoveResult (r)):
    if rt ⊕ {r ↦ bmc.se(i) ⊔ rt(ret)} ⋢ bmc.RT(i+1) then return false;
  case (Goto (t)):
    if rt ⋢ bmc.RT(t) then return false;
  case (If (r_a, r_b, t)):
    k := bmc.se(i) ⊔ rt(r_a) ⊔ rt(r_b)
    if lift(rt, k)} ⋢ bmc.RT(i+1) then return false;
    if lift(rt, k)} ⋢ bmc.RT(t) then return false;
    for j in bmc.region(i) do
      if k ⋢ bmc.se(j) then return false;
  case (Ifz (r, t)):
    k = bmc.se(i) ⊔ rt(r)
    if lift(rt, k)} ⋢ bmc.RT(i+1) then return false;
    if lift(rt, k)} ⋢ bmc.RT(t) then return false;
    for j in bmc.region(i) do
      if k ⋢ bmc.se(j) then return false;
  case (Return): continue;
  case (Return (r)):
    if bmc.se(i) ⊔ rt(r) ⋢ lvt.kr(0) then return false;
  case (NewInstance (r, c)):
    if rt ⊕ {r ↦ bmc.se(i)} ⋢ bmc.RT(i+1) then return false;
  case (Iget (r, r_o, f)):
    if rt ⊕ {r ↦ bmc.se(i) ⊔ rt(r_o) ⊔ ft(f)} ⋢ bmc.RT(i+1) then return false;
  case (Iput (r, r_o, f)):
    k := bmc.se(i) ⊔ rt(r_o) ⊔ rt(r)
    if rt ⋢ bmc.RT(i+1) then return false;
    if k ⋢ ft(f) then return false;
    if k_h ⋢ ft(f) then return false;
  case (Invoke (n, m', p⃗)):
    lvt' := mc.find(m', p⃗[0]).policy;
    if rt ⊕ {ret ↦ lvt'.kr(0) ⊔ bmc.se(i)} ⋢ bmc.RT(i+1) then return false;
    if rt(p⃗[0]) ⊔ lvt.kh ⊔ bmc.se(i) ⋢ lvt'.kh then return false
    for j := 1 to n do
      if p⃗[j] = 0 then
      if rt(p⃗[j]) ⋢ lvt'.ka[j] then return false
return true;
```

Chapter 7

# Conclusion

Chapter 2 shows a policy language design based on MTL that can effectively describe various scenarios of privilege escalation in Android. Moreover, we can effectively enforce any policy written in this language. The key to the latter is the fact that this enforcement procedure is trace-length independent. We have also given a proof-of-concept implementation on actual Android devices and show that our implementation can effectively enforce RMTL policies. To demonstrate the effectiveness of our framework, we demonstrate how it prevents a previously unknown exploit in the system component `com.android.phone` to gain privileges.

In Chapter 3 we presented the design of a type system for DEX programs and showed that in Chapter 6 that the non-optimizing compilation done by the official dx tool preserves the typability of JVM bytecode. Furthermore, Section 4.2 shows that the typability of the DEX program also implies its non-interference. This soundness proof is further reinforced by the formalization in Coq for a subset of DEX (Chapter 6). This opens up the possibility of reusing analysis techniques applicable to Java bytecode for Android. We also provided a proof-of-concept framework from the certificate (non-optimized) translation to the type checker in the form of Android app which can be used to type check actual apps with the corresponding certificate.

## 7.1 Future Work

There are several future directions that we can take from this project. To start with, we already hinted that we are also interested to formalize DEX C and DEX G. It will be good if we also have confidence in the soundness of the type system in the presence of exception since most of the applications make use of this feature.

As an immediate next step for this research of type-preserving compilation, we plan to also take into account the optimization done by the dx tool to see whether the translation preserves typability. This is crucial as real-world apps are compiled full feature of the dx tool, which includes optimization. For this we first need to analyze step by step optimization done by the compiler, then formalize the notion.

We currently do not deal with quantifiers directly in our algorithm for centralized monitoring. Such quantifiers are expanded into purely propositional connectives (when the domain is finite), which is exponential in the number of variables

in the policy. As an immediate future work, we plan to investigate whether techniques using *spawning automata* a la Bauer et al. [2013] can be adapted to our setting to allow a "lazy" expansion of quantifiers as needed. It is not possible to design trace-length-independent monitoring algorithms in unrestricted first-order LTL (Bauer et al. [2013]), so the challenge here is to find a suitable restriction that we can enforce efficiently.

Our work could also complement existing work (Lortz et al. [2014a,b]) on certifying non-interference properties of Android bytecode, by using its language construct (ADL) as a target language for the certificate translation. We plan to investigate how this could be done. Following the spirit of ARTist by Backes et al. [2016], it is also interesting to analyze *dex2oat* translation and optimization to see whether the compilation process preserves typability. Since newer versions of Android is using ART, it is important to be able to leverage current work to be useful for the end users.

Our result is quite orthogonal to the Bitblaze project (Song et al. [2008]), where they aim to unify different bytecodes into a common intermediate language, and then analyze this intermediate language instead. At this moment, we still do not see yet how we can unify DEX bytecode with this intermediate language as there is a quite different approach in programming Android's applications, namely the use of the message passing paradigm which has to be built into the Bitblaze infrastructure. This problem with message passing paradigm is essentially a limitation of our current work as well in that we still have not identified special object and method invocation for this message passing mechanism in the bytecode.

Following the Compcert project (Leroy [2006]; Blazy et al. [2006]), we would ultimately like to have a fully certified end to end compiler. For that reason, in this study, we have not worked directly with the dx tool; rather, we have written our own DEX compiler in OCaml based on our understanding of how the actual dx tool works since it is much easier to reason about OCaml program compared to Java program in which the dx tool is written. Nevertheless, we just use our custom compiler to instrument the bytecode with a certificate, so the actual Android app itself is not affected. Since the final DEX bytecode has a certificate, it does not matter if the compiler is not guaranteed to be correct, as long as the certificate is correct. Furthermore, with the PCC infrastructure, we do not have to trust the certificate generator or even the certificate itself, we just need to trust that the type checker is correct. The good thing about this infrastructure is that the trusted computing base becomes really small.

Appendix A

# Intermediate Type System

In Chapter 6, we have mentioned that there is an intermediate type system which is used to prove the typability. In essence, it really is the same as DEX type system except it has different addressing, i.e., it is indexed by the label of the block and the position of the instruction in the block as opposed to program point. As such, there is no need for us to prove its correctness as we are only interested in the typability (the typability of the target DEX type system is proven separately in Section 4.2). Even though we do not have to prove the correctness, we still have to define several things separately since the mode of addressing is different. We have to define the successor relations, the CDR, and the transfer rule itself.

Throughout the appendix, we will assume that $bi, bj, ...$ is the addressing for block and $i, j, ...$ is the addressing for the instruction's position in the block. All of the functions that take program point will now use the new addressing mode (a pair of block and position).

## A.1 Successor Relations

In the block addressing mode, we have to take care of the last instruction in a block. Apart from that, the rest is the same as the successor relations in DEX.

- $P_m[bi,i]$ = **goto** $(bj,0)$. The successor relation is $(bi,i) \mapsto^{\text{Norm}} (bj,1)$
- $P_m[bi,i]$ = **ifeq** $(bj,0)$ or $P_m[bi,i]$ = **ifneq** $t$. In this case, there are 2 successor relations denoted by $(bi,i) \mapsto^{\text{Norm}} (bj,0)$ and $(bi,i) \mapsto^{\text{Norm}} (bi,i+1)$ and if $i$ is not the last instruction in the block. Otherwise $(bi,i) \mapsto^{norm} (bj,0)$ where $b(bi,i) \mapsto^{norm} bj$.
- $P_m[bi,i]$ = **return**. In this case it is a return point denoted by $(bi,i) \mapsto^{\text{Norm}}$
- $P_m[bi,i]$ is an instruction throwing a null pointer exception, and there is a handler for it (**Handler**$((bi,i),\text{np}) = t$). In this case, the successor is $t$ denoted by $(bi,i) \mapsto^{\text{np}} t$.
- $P_m[bi,i]$ is an instruction throwing a null pointer exception, and there is no handler for it (**Handler**$((bi,i),\text{np}) \uparrow$). In this case it is a return point denoted by $(bi,i) \mapsto^{\text{np}}$.

- $P_m[bi,i]$ = **throw**, throwing an exception $C \in$ **classAnalysis**$(m,(bi,i))$, and the handler is **Handler**$((bi,i),C) = t$. The successor relation is $(bi,i) \mapsto^C t$.
- $P_m[bi,i]$ = **throw**, throwing an exception $C \in$ **classAnalysis**$(m,(bi,i))$, and the handler is **Handler**$((bi,i),C) = t$. It is a return point and the successor relation is $(bi,i) \mapsto^C$.
- $P_m[bi,i]$ = **invoke** $m_{\text{ID}}$, throwing an exception $C \in$ **excAnalysis**$(m_{\text{ID}})$, and the handler is **Handler**$((bi,i),C) = t$. The successor relation is $(bi,i) \mapsto^C t$.
- $P_m[bi,i]$ = **invoke** $m_{\text{ID}}$, throwing an exception $C \in$ **excAnalysis**$(m_{\text{ID}})$, and the handler is **Handler**$((bi,i),C) \uparrow$. It is a return point and the successor relation is $(bi,i) \mapsto^C$.
- $P_m[bi,i]$ is any other cases. The successor is its immediate instruction denoted by $(bi,i) \mapsto^{norm} (bi,i+1)$ if $i$ is not the last instruction in the block. Otherwise $(bi,i) \mapsto^{norm} (bj,0)$ where $b(bi,i) \mapsto^{norm} bj$.

## A.2 Control Dependence Region

Even though the successor relation requires special handling for the last instruction in a block, we do not have to do the same for CDR since it is inherent in the successor relation used in CDR.

**SOAP1.** $\forall (bi,i),(bj,j),(bk,k) \in \mathcal{PP}$ and tag $\tau$ if $(bi,i) \mapsto (bj,j)$ and $(bi,i) \mapsto^\tau (bk,k)$ and $(bj,j) \neq (bk,k)$ ($(bi,i)$ is hence a branching point) then $(bk,k) \in$ **region**$((bi,i),\tau)$ or $(bk,k) =$ **jun**$((bi,i),\tau)$.

**SOAP2.** $\forall (bi,i),(bj,j),(bk,k) \in \mathcal{PP}$ and tag $\tau$, if $(bj,j) \in$ **region**$((bi,i),\tau)$ and $(bj,j) \mapsto (bk,k)$, then either $(bk,k) \in$ **region**$((bi,i),\tau)$ or $(bk,k) =$ **jun**$((bi,i),\tau)$.

**SOAP3.** $\forall (bi,i),(bj,j) \in \mathcal{PP}$ and tag $\tau$, if $(bj,j) \in$ **region**$((bi,i),\tau)$ and $(bj,j)$ is a return point then **jun**$((bi,i),\tau)$ is undefined.

**SOAP4.** $\forall (bi,i) \in \mathcal{PP}$ and tags $\tau_1,\tau_2$ if **jun**$((bi,i),\tau_1)$ and **jun**$((bi,i),\tau_2)$ are defined and **jun**$((bi,i),\tau_1) \neq$ **jun**$((bi,i),\tau_2)$ then **jun**$((bi,i),\tau_1) \in$ **region**$((bi,i),\tau_2)$ or **jun**$((bi,i),\tau_2) \in$ **region**$((bi,i),\tau_1)$.

**SOAP5.** $\forall (bi,i),(bj,j) \in \mathcal{PP}$ and tag $\tau$, if $(bj,j) \in$ **region**$((bi,i),\tau)$ and $(bj,j)$ is a return point then for all tags $\tau'$ if **jun**$((bi,i),\tau')$ is defined then **jun**$((bi,i),\tau') \in$ **region**$((bi,i),\tau)$.

**SOAP6.** Let $\forall (bi,i) \in \mathcal{PP}$ and tag $\tau_1$, if $(bi,i) \mapsto^{\tau_1}$ then for all tags $\tau_2$, **region**$((bi,i),\tau_2) \subseteq$ **region**$((bi,i),\tau_1)$. If we have **jun**$((bi,i),\tau_2)$ is defined then **jun**$((bi,i),\tau_2) \in$ **region**$((bi,i),\tau_1)$.

## A.3 Transfer Rules

$$\frac{P_m[(bi,i)] = \mathbf{const}(r,v)}{se,(bi,i) \vdash rt \Rightarrow rt \oplus \{r \mapsto se(bi,i)\}} \qquad \frac{P_m[(bi,i)] = \mathbf{move}(r,r_s)}{se,(bi,i) \vdash rt \Rightarrow rt \oplus \{r \mapsto \big(rt(r_s) \sqcup se(bi,i)\big)\}}$$

$$\frac{P_m[bi,i] = \mathbf{ifeq}(r,t) \quad \forall (bj,j') \in \mathbf{region}((bi,i),\mathrm{Norm}), se(bi,i) \sqcup rt(r) \leq se(bj,j')}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash rt \Rightarrow rt}$$

$$\frac{P_m[bi,i] = \mathbf{ifneq}(r,t) \quad \forall (bj,j') \in \mathbf{region}((bi,i),\mathrm{Norm}), se(bi,i) \sqcup rt(r) \leq se(bj,j')}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash rt \Rightarrow rt}$$

$$\frac{P_m[bi,i] = \mathbf{binop}(op,r,r_a,r_b)}{se,(bi,i) \vdash rt \Rightarrow rt \oplus \{r \mapsto \big(rt(r_a) \sqcup rt(r_b) \sqcup se(bi,i)\big)\}}$$

$$\frac{P_m[bi,i] = \mathbf{return}(r_s) \quad se(bi,i) \sqcup rt(r_s) \leq \vec{k_r}(\mathrm{Norm})}{\vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, (bi,i) \vdash rt \Rightarrow}$$

$$\frac{P_m[bi,i] = \mathbf{new}(r,c)}{se,(bi,i) \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(bi,i)\}}$$

$$\frac{P_m[bi,i] = \mathbf{iget}(r,r_o,f) \quad rt(r_o) \in \mathcal{S} \quad \forall (bj,j) \in \mathbf{region}((bi,i),\mathrm{Norm}), rt(r_o) \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto \big((rt(r_o) \sqcup se(bi,i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f)\big)\}}$$

$$\frac{\begin{array}{c}P_m[bi,i] = \mathbf{iget}(r,r_o,f) \quad rt(r_o) \in \mathcal{S} \quad \forall (bj,j) \in \mathbf{region}((bi,i),\mathrm{np}), rt(r_o) \leq se(bj,j) \\ \mathbf{Handler}((bi,i),\mathrm{np}) = t\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\mathrm{np}} rt \Rightarrow rt \oplus \{ex \mapsto (rt(r_o) \sqcup se(bi,i))\}}$$

$$\frac{\begin{array}{c}P_m[bi,i] = \mathbf{iget}(r,r_o,f) \quad rt(r_o) \in \mathcal{S} \quad \forall (bj,j) \in \mathbf{region}((bi,i),\mathrm{np}), rt(r_o) \leq se(bj,j) \\ \mathbf{Handler}((bi,i),\mathrm{np}) \uparrow \quad se(bi,i) \sqcup rt(r_o) \leq \vec{k_r}(\mathrm{np})\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\mathrm{np}} rt \Rightarrow}$$

$$\frac{\begin{array}{c}P_m[bi,i] = \mathbf{iput}(r,r_o,f) \quad rt(r) \in \mathcal{S}^{\mathrm{ext}} \quad rt(r_o) \in \mathcal{S} \quad (rt(r_o) \sqcup se(bi,i)) \sqcup^{\mathrm{ext}} rt(r_s) \leq \mathbf{ft}(f) \\ k_h \leq \mathbf{ft}(f) \quad \forall (bj,j) \in \mathbf{region}((bi,i),\mathrm{Norm}), rt(r_o) \leq se(bj,j)\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\mathrm{Norm}} rt \Rightarrow rt}$$

$$\frac{\begin{array}{c}P_m[bi,i] = \mathbf{iput}(r_s,r_o,f) \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad rt(r_o) \in \mathcal{S} \quad (rt(r_o) \sqcup se(bi,i)) \sqcup^{\mathrm{ext}} rt(r_s) \leq \mathbf{ft}(f) \\ \forall (bj,j) \in \mathbf{region}((bi,i),\mathrm{np}), rt(r_o) \leq se(bj,j) \quad \mathbf{Handler}((bi,i),\mathrm{np}) = t\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\mathrm{np}} rt \Rightarrow rt \oplus \{ex \mapsto rt(r_o) \sqcup se(bi,i)\}}$$

$$\frac{P_m[bi,i] = \mathbf{iput}(r_s, r_o, f) \quad rt(r_s) \in \mathcal{S}^{\text{ext}} \quad rt(r_o) \in \mathcal{S} \quad (rt(r_o) \sqcup se(bi,i)) \sqcup^{\text{ext}} rt(r_s) \leq \mathbf{ft}(f) \quad \forall (bj,j) \in \mathbf{region}((bi,i),\text{np}), rt(r_o) \leq se(bj,j) \quad \mathbf{Handler}((bi,i),\text{np}) \uparrow \quad se(bi,i) \sqcup rt(r_o) \leq \vec{k_r}(\text{np})}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{np}} rt \Rightarrow}$$

$$\frac{P_m[bi,i] = \mathbf{newarray}(r, r_l, t) \quad rt(r_l) \in \mathcal{S} \quad rt(r_l)[\mathbf{at}((bi,i))] \leq \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto rt(r_l)[\mathbf{at}((bi,i))]\}}$$

$$\frac{P_m[bi,i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall (bj,j) \in \mathbf{region}((bi,i),\text{Norm}), k \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto k\}}$$

$$\frac{P_m[bi,i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad k \leq \vec{k_a}(r) \quad \forall (bj,j) \in \mathbf{region}((bi,i),\text{np}), k \leq se(bj,j) \quad \mathbf{Handler}((bi,i),\text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{np}} rt \Rightarrow rt \oplus \{ex \mapsto (k \sqcup se(bi,i))\}}$$

$$\frac{P_m[bi,i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad k \leq \vec{k_a}(r) \quad \forall (bj,j) \in \mathbf{region}((bi,i),\text{np}), k \leq se(bj,j) \quad \mathbf{Handler}((bi,i),\text{np}) \uparrow \quad se(bi,i) \sqcup k \leq \vec{k_a}[\text{np}]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{np}} rt \Rightarrow}$$

$$\frac{P_m[bi,i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall (bj,j) \in \mathbf{region}((bi,i),\text{Norm}), k \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto \left((se(bi,i) \sqcup k \sqcup rt(r_i)\right) \sqcup^{\text{ext}} k_c)\}}$$

$$\frac{P_m[bi,i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall (bj,j) \in \mathbf{region}((bi,i),\text{np}), k \leq se(bj,j) \quad \mathbf{Handler}((bi,i),\text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{np}} rt \Rightarrow rt \oplus \{ex \mapsto (k \sqcup se(bi,i))\}}$$

$$\frac{P_m[bi,i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall (bj,j) \in \mathbf{region}((bi,i),\text{np}), k \leq se(bj,j) \quad \mathbf{Handler}((bi,i),\text{np}) \uparrow \quad se(bi,i) \sqcup k \leq \vec{k_r}(\text{np})}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{np}} rt \Rightarrow}$$

$$\frac{P_m[bi,i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad \forall (bj,j) \in \mathbf{region}((bi,i), \text{Norm}), k \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{Norm}} rt \Rightarrow rt}$$

$$\frac{P_m[bi,i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad \forall (bj,j) \in \mathbf{region}((bi,i), \text{np}), k \leq se(bj,j) \quad \mathbf{Handler}((bi,i), \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{np}} rt \Rightarrow rt \oplus \{ex \mapsto (k \sqcup se(bi,i))\}}$$

$$\frac{P_m[bi,i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \quad se(bi,i) \sqcup k \leq \vec{k_r}(\text{np}) \quad ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \leq^{\text{ext}} k_c \quad \forall (bj,j) \in \mathbf{region}((bi,i), \text{np}), k \leq se(bj,j) \quad \mathbf{Handler}((bi,i), \text{np}) \uparrow}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{np}} rt \Rightarrow}$$

$$\frac{P_m[bi,i] = \mathbf{moveresult}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(bi,i) \sqcup rt(ret)\}}$$

$$\frac{P_m[bi,i] = \mathbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\to} \vec{k'_r} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(bi,i) \leq k'_h \quad \forall 0 \leq i < n.rt(\vec{p}[i]) \leq \vec{k'_a}[i] \quad k_e = \bigsqcup \{\vec{k'_r}(e) \mid e \in \mathbf{excAnalysis}(m')\} \quad \forall (bj,j) \in \mathbf{region}((bi,i), \text{Norm}), rt(\vec{p}[0]) \sqcup k_e \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\text{Norm}} rt \Rightarrow \left(rt \oplus \{ret \mapsto \vec{k'_r}(\text{Norm}) \sqcup se(bi,i)\}\right)}$$

$$\frac{P_m[bi,i] = \mathbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\to} \vec{k'_r} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(bi,i) \leq k'_h \quad \forall 0 \leq i < n.rt(\vec{p}[i]) \leq \vec{k'_a}[i] \quad \mathbf{Handler}((bi,i), e) = t \quad e \in \mathbf{excAnalysis}(m') \cup \{\text{np}\} \quad \forall (bj,j) \in \mathbf{region}((bi,i), e), rt(\vec{p}[0]) \sqcup k'_r[e] \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{e} rt \Rightarrow rt \oplus \{ex \mapsto (rt(\vec{p}[0]) \sqcup \vec{k'_r}(e))\}}$$

$$\frac{P_m[bi,i] = \mathbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\to} \vec{k'_r} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(bi,i) \leq k'_h \quad \forall 0 \leq i < n.rt(\vec{p}[i]) \leq \vec{k'_a}[i] \quad rt(\vec{p}[0]) \sqcup se(bi,i) \sqcup \vec{k'_r}(e) \leq \vec{k_r}(e) \quad \mathbf{Handler}((bi,i), e) \uparrow \quad e \in \mathbf{excAnalysis}(m') \cup \{\text{np}\} \quad \forall (bj,j) \in \mathbf{region}((bi,i), e), rt(\vec{p}[0]) \sqcup k'_r[e] \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{e} rt \Rightarrow}$$

$$\frac{P_m[bi,i] = \mathbf{throw}(r) \quad e \in \mathbf{classAnalysis}((bi,i)) \cup \{\mathrm{np}\} \quad \mathbf{Handler}((bi,i),e) = t \quad \forall (bj,j) \in \mathbf{region}((bi,i),e), rt(r) \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{e} rt \Rightarrow rt \oplus \{ex \mapsto (rt(r) \sqcup se(bi,i))\}}$$

$$\frac{P_m[bi,i] = \mathbf{throw}(r) \quad e \in \mathbf{classAnalysis}((bi,i)) \cup \{\mathrm{np}\} \quad se(bi,i) \sqcup rt(r) \leq \vec{k_r}(e) \quad \mathbf{Handler}((bi,i),e) \uparrow \quad \forall (bj,j) \in \mathbf{region}((bi,i),e), rt(r) \leq se(bj,j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{e} rt \Rightarrow}$$

$$\frac{P_m[bi,i] = \mathbf{moveexception}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \mathbf{region}, se, (bi,i) \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto (rt(ex) \sqcup se(bi,i))\}}$$

Figure A.1: DEX Intermediate Transfer Rule

# Bibliography


2016. DEX Bytecode instructions. http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. Accessed: 2016-11-8. (cited on page 37)

2017. Source Android. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. Accessed: 2017-3-25. (cited on page 67)

2017. Stat Counter Global Stats. http://gs.statcounter.com/os-market-share/mobile/chart.php?device=Mobile&device_hidden=mobile&statType_hidden=os_combined&region_hidden=ww&granularity=monthly&statType=Operating%20System&region=Worldwide&fromInt=201608&toInt=201708&fromMonthYear=2016-08&toMonthYear=2017-08&csv=1. Accessed: 2017-9-23. (cited on page 1)

Alur, R. and Henzinger, T. A., 1990. Real-time Logics: Complexity and Expressiveness. In *LICS*, 390–401. IEEE Computer Society. (cited on page 8)

Appel, A. W., 2001. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, 247–256. IEEE Computer Society. doi:10.1109/LICS.2001.932501. http://dx.doi.org/10.1109/LICS.2001.932501. (cited on page 37)

Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Le Traon, Y.; Octeau, D.; and McDaniel, P., 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49, 6 (2014), 259–269. (cited on page 39)

Backes, M.; Bugiel, S.; and Gerling, S., 2014. Scippa: System-centric ipc provenance on android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14 (New Orleans, Louisiana, USA, 2014), 36–45. ACM, New York, NY, USA. doi:10.1145/2664243.2664264. http://doi.acm.org/10.1145/2664243.2664264. (cited on pages 10 and 11)

Backes, M.; Bugiel, S.; Schranz, O.; von Styp-Rekowsky, P.; and Weisgerber, S., 2016. ARTist: The Android Runtime Instrumentation and Security Toolkit. *arXiv preprint arXiv:1607.06619*, (2016). (cited on pages 40 and 200)

Banerjee, A. and Naumann, D. A., 2005. Stack-based Access Control and Secure Information Flow. *J. Funct. Program.*, 15, 2 (Mar. 2005), 131–177. doi:10.1017/S0956796804005453. http://dx.doi.org/10.1017/S0956796804005453. (cited on page 37)

Barthe, G.; Naumann, D.; and Rezk, T., 2006a. Deriving an information flow checker and certifying compiler for Java. In *Security and Privacy, 2006 IEEE Symposium on*, 13–pp. IEEE. (cited on pages 38, 39, and 175)

Barthe, G.; Pichardie, D.; and Rezk, T., 2007. A Certified Lightweight Non-Interference Java Bytecode Verifier. http://hal.inria.fr/inria-00106182. Submitted to TOPLAS in September 2007 Work partially supported by IST Project MOBIUS, by the RNTL Castles and by the ACI Sécurité SPOPS. (cited on pages 55 and 101)

Barthe, G.; Pichardie, D.; and Rezk, T., 2013. A certified lightweight non-interference Java bytecode verifier. *Mathematical Structures in Computer Science*, 23 (10 2013), 1032–1081. doi:10.1017/S0960129512000850. http://journals.cambridge.org/article_S0960129512000850. (cited on pages 37, 38, 41, 55, and 59)

Barthe, G.; Rezk, T.; and Saabas, A., 2006b. Proof obligations preserving compilation. In *Formal Aspects in Security and Trust*, 112–126. Springer. (cited on pages 38 and 39)

Basin, D. A.; Klaedtke, F.; and Muller, S., 2010. Policy Monitoring in First-order Temporal Logic. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, 1–18. (cited on page 12)

Basin, D. A.; Klaedtke, F.; Müller, S.; and Pfitzmann, B., 2008. Runtime Monitoring of Metric First-order Temporal Properties. In *FSTTCS*, vol. 2 of *LIPIcs*, 49–60. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. (cited on pages 8, 9, and 11)

Basin, D. A.; Klaedtke, F.; and Zalinescu, E., 2012. Algorithms for Monitoring Real-Time Properties. In *RV*, vol. 7186 of *LNCS*, 260–275. Springer. (cited on pages 9, 11, and 24)

Bauer, A.; Gore, R.; and Tiu, A., 2009. A first-order policy language for history-based transaction monitoring. In *Proc. 6th Intl. Colloq. Theoretical Aspects of Computing (ICTAC)*, vol. 5684 of LNCS, 96–111. Springer. (cited on pages 12 and 19)

Bauer, A.; Küster, J.-C.; and Vegliach, G., 2013. From Propositional to First-order Monitoring. In *RV*, vol. 8174 of *LNCS*, 59–75. (cited on pages 9, 10, 19, and 200)

Bian, G.; Nakayama, K.; Kobayashi, Y.; and Maekawa, M., 2007. Java Bytecode Dependence Analysis for Secure Information Flow. *IJ Network Security*, 4, 1 (2007), 59–68. (cited on page 38)

Blackburn, P.; van Benthem, J. F.; and Wolter, F., 2007. *Handbook of Modal Logic*. Elsevier. (cited on page 11)

Blazy, S.; Dargaye, Z.; and Leroy, X., 2006. Formal verification of a C compiler front-end. In *FM 2006: Formal Methods*, 460–475. Springer. (cited on page 200)

Bradfield, J. and Stirling, C., 2007. Modal mu-calculi. In *HANDBOOK OF MODAL LOGIC*, 721–756. Elsevier. (cited on page 9)

Brewer, D. F. C. and Nash, M. J., 1989. The Chinese Wall Security Policy. In *IEEE Symposium on Security and Privacy*. IEEE. (cited on page 25)

Bugiel, S.; Davi, L.; Dmitrienko, A.; Fischer, T.; Sadeghi, A.-R.; and Shastry, B., 2012. Towards Taming Privilege-Escalation Attacks on Android. In *NDSSâĂŹ12*. (cited on pages 7, 8, and 10)

Bugliesi, M.; Calzavara, S.; and Spanò, A., 2013. Lintent: towards security type-checking of Android applications. In *Formal Techniques for Distributed Systems*, 289–304. Springer. (cited on page 39)

Chan, P. P. F.; Hui, L. C. K.; and Yiu, S.-M., 2012. Droidchecker: analyzing Android applications for capability leak. In *WISEC*, 125–136. ACM. (cited on page 7)

Chaudhuri, A., 2009. Language-based security on Android. In *Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security*, 1–7. ACM. (cited on page 39)

Chowdhury, O.; Jia, L.; Garg, D.; and Datta, A., 2014. Temporal mode-checking for runtime monitoring of privacy policies. In *International Conference on Computer Aided Verification*, 131–149. Springer. (cited on page 11)

Davi, L.; Dmitrienko, A.; Sadeghi, A.-R.; and Winandy, M., 2011. Privilege Escalation Attacks on Android. In *ISC 2010*, vol. 6531 of *LNCS*, 346–360. (cited on page 7)

Davis, B.; Beatty, A.; Casey, K.; Gregg, D.; and Waldron, J., 2003. The Case for Virtual Register Machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03 (San Diego, California, 2003), 41–49. ACM, New York, NY, USA. doi:10.1145/858570.858575. http://doi.acm.org/10.1145/858570.858575. (cited on page 143)

Denning, D. E. and Denning, P. J., 1977. Certification of programs for secure information flow. *Commun. ACM*, 20, 7 (Jul. 1977), 504–513. doi:10.1145/359636.359712. http://doi.acm.org.ezlibproxy1.ntu.edu.sg/10.1145/359636.359712. (cited on page 8)

Dietz, M.; Shekhar, S.; Pisetsky, Y.; Shu, A.; and Wallach, D. S., 2011. QUIRE: Lightweight provenance for smartphone operating systems. In *20th USENIX Security Symposium*. (cited on pages 8 and 10)

Dnyaneshwar, R. A., 2017. Custom privacy guards in android. (cited on page 30)

Enck, W., 2011. A study of android application security. (cited on page 4)

Enck, W.; Gilbert, P.; Chun, B.-G.; Cox, L. P.; Jung, J.; McDaniel, P.; and Sheth, A. N., 2014. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57, 3 (2014), 99–106. (cited on pages 5, 39, and 40)

Enck, W.; Gillbert, P.; Chun, B.-G.; Cox, L. P.; Jung, J.; McDaniel, P.; and Sheth, A. N., 2010. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*. (cited on pages 8 and 11)

Enck, W.; Ongtang, M.; and McDaniel, P., 2009a. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, 235–245. ACM. (cited on page 39)

Enck, W.; Ongtang, M.; and McDaniel, P. D., 2009b. Understanding Android Security. *IEEE Security & Privacy*, 7, 1 (2009), 50–57. (cited on pages 2 and 39)

Felt, A. P.; Wang, H.; Moschuk, A.; Hanna, S.; and Chin, E., 2011. Permission Re-delegation: Attacks and Defenses. In *20th USENIX Security Symposium*. (cited on pages 8, 10, and 39)

Fitting, M., 1996. *First-Order Logic and Automated Theorem Proving*. Springer. (cited on page 14)

Fragkaki, E.; Bauer, L.; Jia, L.; and Swasey, D., 2012. Modeling and enhancing Android's permission system. In *Computer Security—ESORICS 2012: 17th European Symposium on Research in Computer Security*, vol. 7459 of *Lecture Notes in Computer Science*, 1–18. doi:10.1007/978-3-642-33167-1_1. http://www.ece.cmu.edu/~lbauer/papers/2012/esorics2012-android.pdf. (cited on pages 39 and 40)

Fuchs, A. P.; Chaudhuri, A.; and Foster, J. S., 2009. Scandroid: Automated security certification of Android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/~ avik/projects/scandroidascaa*, (2009). (cited on page 39)

Goguen, J. A. and Meseguer, J., 1982. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 11–11. IEEE Computer Society. (cited on page 37)

Grace, M.; Zhou, Y.; Wang, Z.; and Jiang, X., 2012. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSSâĂŹ12*. (cited on page 7)

Havelund, K. and Rosu, G., 2002. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, vol. 2280 of LNCS, 342–356. Springer. (cited on pages 20 and 22)

Jia, L.; Aljuraidan, J.; Fragkaki, E.; Bauer, L.; Stroucken, M.; Fukushima, K.; Kiyomoto, S.; and Miyake, Y., 2013. Run-Time Enforcement of Information-Flow Properties on Android (extended abstract). In *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security*, 775–792. Springer. doi:10.1007/978-3-642-40203-6_43. http://www.ece.cmu.edu/~lbauer/papers/2013/esorics2013-android.pdf. (cited on pages 39 and 40)

Kim, J.; Yoon, Y.; Yi, K.; Shin, J.; and Center, S., 2012. Scandal: Static analyzer for detecting privacy leaks in Android applications. *MoST*, (2012). (cited on pages 39 and 40)

Leroy, X., 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *ACM SIGPLAN Notices*, 41, 1 (2006), 42–54. (cited on page 200)

Lichtenstein, O.; Pnueli, A.; and Zuck, L. D., 1985. The Glory of the Past. In *Logic of Programs*, vol. 193 of *LNCS*, 196–218. Springer. (cited on page 8)

Lindholm, T.; Yellin, F.; Bracha, G.; and Buckley, A., 2013. Jvm bytecode verifier. https://docs.oracle.com/javase/specs/jvms/se7/html/. (cited on page 143)

Lineberry, A.; Richardson, D. L.; and Wyatt, T., 2010. These aren't the permissions you're looking for. In *DefCon 18*. (cited on pages 7 and 8)

Lortz, S.; Mantel, H.; Starostin, A.; Bähr, T.; Schneider, D.; and Weber, A., 2014a. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03 - 07, 2014*, 93–104. ACM. doi:10.1145/2666620.2666631. http://doi.acm.org/10.1145/2666620.2666631. (cited on pages 38 and 200)

Lortz, S.; Mantel, H.; Starostin, A.; and Weber, A., 2014b. A Sound Information-Flow Analysis for Cassandra. Technical report, TU Darmstadt. Technical Report TUD-CS-2014-0064. (cited on pages 38 and 200)

McCarty, B., 2004. *Selinux: Nsa's open source security enhanced linux*. O'Reilly Media, Inc. (cited on page 1)

Necula, G. C., 1997. Proof-Carrying Code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, 106–119. ACM Press. doi:10.1145/263699.263712. http://doi.acm.org/10.1145/263699.263712. (cited on pages 37 and 40)

Octeau, D.; Jha, S.; and McDaniel, P., 2012. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, 6. ACM. (cited on page 40)

Octeau, D.; McDaniel, P.; Jha, S.; Bartel, A.; Bodden, E.; Klein, J.; and Le Traon, Y., 2013. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis*, (2013). (cited on page 40)

Pnueli, A., 1977. The Temporal Logic of Programs. In *FOCS*, 46–57. IEEE Computer Society. (cited on page 14)

Reaves, B.; Bowers, J.; Gorski III, S. A.; Anise, O.; Bobhate, R.; Cho, R.; Das, H.; Hussain, S.; Karachiwala, H.; Scaife, N.; et al., 2016. * droid: Assessment and evaluation of android application analysis tools. *ACM Computing Surveys (CSUR)*, 49, 3 (2016), 55. (cited on page 40)

Reinbacher, T.; Függer, M.; and Brauer, J., 2013. Real-Time Runtime Verification on Chip. In *RV*, vol. 7687 of *LNCS*, 110–125. Springer. (cited on pages 9 and 11)

Sabelfeld, A. and Myers, A. C., 2003. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21, 1 (2003), 5–19. (cited on page 37)

Schlegel, R.; Zhang, K.; Zhou, X.; Intwala, M.; Kapadia, A.; and Wang, X., 2011. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *18th Annual Network and Distributed System Security Symposium (NDSS)*. (cited on pages 7, 8, and 30)

Song, D.; Brumley, D.; Yin, H.; Caballero, J.; Jager, I.; Kang, M. G.; Liang, Z.; Newsome, J.; Poosankam, P.; and Saxena, P., 2008. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*, 1–25. Springer. (cited on page 200)

Thati, P. and Rosu, G., 2004. Monitoring Algorithms for Metric Temporal Logic Specifications. In *Proc. of RVâĂŹ04*. (cited on pages 13 and 15)

Thati, P. and Rosu, G., 2005. Monitoring Algorithms for Metric Temporal Logic Specifications. *Electr. Notes Theor. Comput. Sci.*, 113 (2005), 145–162. (cited on pages 8, 9, 11, and 17)

Yagemann, C. and Du, W., 2016. Intentio ex machina: Android intent access control via an extensible application hook. In *European Symposium on Research in Computer Security*, 383–400. Springer. (cited on page 11)

Zhang, D.; Wang, R.; Lin, Z.; Guo, D.; and Cao, X., 2016. Iacdroid: Preventing inter-app communication capability leaks in android. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*, 443–449. IEEE. (cited on page 11)

Zhao, Z. and Osono, F. C. C., 2012. Trustdroid: Preventing the use of Smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, 135–143. IEEE. (cited on page 39)