# Formal Certification of Android Bytecode

Alwen Tiu
School of Computer Engineering
Nanyang Technological University
Email: atiu@ntu.edu.sg

Hendra Gunadi
Research School of Computer Science
The Australian National University
Email: hendra.gunadi@anu.edu.au

Rajeev Gore
Research School of Computer Science
The Australian National University
Email: rajeev.gore@anu.edu.au

*Abstract*—**Android is an operating system that has been used in a majority of mobile devices. 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 then translated to DEX bytecode using the dx tool in the Android SDK. In this work, we aim to develop a type-based method for certifying non-interference properties of DEX bytecode, following a methodology that has been developed for Java bytecode certification by Barthe et al. 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 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 (which guarantees non-interference) 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.**

## I. INTRODUCTION

Android is an operating system that has been used in many mobile devices. According to [1], Android has the largest market share for mobile devices, making it an attractive target for malwares, so verification of the security properties of Android apps is crucial. 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). Each of these applications runs in an instance of a Dalvik virtual machine (VM) on top of the Linux operating system. Contained in each of these APKs is a DEX file containing specific instructions [2] 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, but this 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 (sensitive) information the apps use is not leaked outside the device without the user's consent. The framework should also provide a mean 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) [3], [4], applied to the Android setting. We follow a type-based approach for restricting information flow [5] in Android apps. Semantically, information flow properties of apps are specified via a notion of non-interference [6]. In this setting, typeable programs are guaranteed to be non-interferrent, with respect to a given policy, and typing derivations serve as certificates of non-interference. Our eventual goal is to produce a compiler tool chain that can help developers to develop Android applications that complies 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 will aggregate the Java classes and produce a DEX file to be bundled together with other resource files in the APK. Non-interference type systems exist for Java source code [7], JVM [8] and (abstracted) DEX bytecode without exception handling mechanism [9]. 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 in [10]. In this work, we fill the gap by showing that the connection between JVM and DEX type systems. Our contributions are the following:

- 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 VI details some of the translation processes.

- We provide a proof that the translation from JVM to DEX preserves typeability. That is, JVM programs typeable in the non-interference type system for JVM translates into typeable programs in the non-interference type system for DEX.

The development of the operational semantics and the type systems for DEX bytecode follows closely the framework set up in [8]. Although Dalvik is a registered-based machine and JVM is a stack-based machine, the translation from one instruction set to the other is for 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 [8], 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 from JVM to DEX, the dx tool simulates the operand stack using DEX registers. As the type system for JVM is parameterised 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) codes, so one needs to relate blocks of codes in the image of the translation back to the original codes (see Section VI) .

The rest of the paper is organized as follows : in the next section we describe some work done for Java bytecode security and the work on static analysis for Android bytecode. This is important as what we are doing is bridging the relationship between these two security measures. Then we review the work of Barthe et.al. on a non-interferent type system for JVM bytecode. In the remainder of the paper we will describe our work, namely providing the type system for DEX and proving the translation of typability. We also give examples to demonstrate how our methodology is able to detect interference by failure of typability. Before concluding, we provide our design of implementation for the proof of concept.

## II. RELATED WORK

As we already mentioned, our work is heavily influenced by the work of Barthe et. al. [11], [8] on enforcing non-interference via type systems. We discuss other related work in the following.

The cloest to our work is the Cassandra project [9], [12], that aims at developing certified app stores, where apps can be certified, using an information-flow type system similar to ours, for absence of specific information flow. Specifically, the authors of [9], [12] have developed an abstract Dalvik language (ADL), similar to Dalvik bytecode, and a type system for enforcing non-interference properties for ADL. 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 ADL since we aim to eventually integrate our certificate compilation into existing compiler tool chains for Android apps, without having to modify those tool chains.

Bian et. al. [13] targets 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 analyse 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, i.e., 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. [11] 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 stack-based virtual machine (JVM bytecode) to the actual language for register-based machine (DEX bytecode). There are also works that address this non-interference preservation from Java source code to JVM bytecode [10]. 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 [14], [15], [16], [17], [18], [19], [20], [21], [22], [23] although some of them are geared towards the privilege escalation problem. These works base their context of Android security studied in [23]. 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 static analysis for Android. The closest one to mention is ScanDroid [14], with the underlying type system and static analysis tool for security specification in Android [24]. Then along the line of type system there is also work by Bugliesi et. al. called Lintent that tries to address non-interference on the source code level [15]. The main difference with what we do lies in that the analysis itself is relying 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. Trust-Droid [17] and ScanDal [18]. TrustDroid is another static analysis tool on Android bytecode, trying to prevent infor-mation leaking. TrustDroid is more interested in doing taint analysis on the program, although different from TaintDroid [16] in that TrustDroid is 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 targeting 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.

Since the property that we are interested in is non-interference, it is also worth mentioning Sorbet, a run time

| | | |
|---|---|---|
| **binop** $op$ | : | binary operation on stack |
| **push** $c$ | : | push value on top of a stack |
| **pop** | : | pop value from top of a stack |
| **swap** | : | swap top two operand stack values |
| **load** $x$ | : | load value of $x$ on stack |
| **store** $x$ | : | store top of stack in variable $x$ |
| **ifeq** $j$ | : | conditional jump |
| **goto** $j$ | : | unconditional jump |
| **return** | : | return the top value of the stack |
| **new** $C$ | : | create new object in the heap |
| **getfield** $f$ | : | load value of field $f$ on stack |
| **putfield** $f$ | : | store top of stack in field $f$ |
| **newarray** $t$ | : | create 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 $m_{\mathrm{ID}}$ with arguments on top of the stack |
| **throw** | : | Throw exception 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 $m_{\mathrm{ID}} \in \mathcal{M}$.

Fig. 1: JVM Instruction List

enforcement of the property by modifying the Android operating system [19], [20]. 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.

## III. TYPE SYSTEM FOR JVM

In this section, we give an overview of Barthe et. al's type system for JVM. Due to space constraints, some details are omitted and the reader is referred to [8] 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.

A program $P$ is given by its list of instructions given in Figure 1. 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. We use the notation $^*$ to mean that for any set $X$, $X^*$ is 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 of Java types $\mathcal{T}_J$. The instructions listing can be seen in Figure 1.

**Operational Semantics** The operational semantics is given as a relation $\leadsto_{m,\tau} \subseteq \mathrm{State} \times (\mathrm{State} + (\mathcal{V}, \mathbf{heap}))$ where $m$ indicates the method under which the relation is considered and $\tau$ indicates whether the instruction is executing normally (indicated by $\mathrm{Norm}$) 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 the instruction is executing normally or not ). State here represents a set of JVM states, which is a tuple $\langle i, \rho, os, h \rangle$ where $\mathrm{i} \in \mathcal{PP}$ 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 $h \in \mathbf{heap}$ 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 $\mathbf{class}(o)$ and a partial function to access field values, which is denoted by $o.f$ 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. $\mathbf{Heap}$ is the set of heaps.

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 exception in the heap $h$.

$\overline{op}$ denotes here the standard interpretation of arithmetic operation of $op$ in the domain of values $\mathcal{V}$ (although there is no arithmetic operation on locations).

The instruction that may throw an exception primarily are method invocation and the object/array manipulation instructions. $\{np\}$ is used as the class for null pointer exceptions, with the associated exception handler being $\mathbf{RuntimeExceptionHandling}$. The transitions are also parameterized by a tag $\tau \in \{\mathrm{Norm}\} + \mathcal{C}$ to describe whether the transition occurs normally or some exception is thrown.

Some last remarks: firstly, because of method invocation, the operational semantics will also be mixed with a big step semantics style $\leadsto_m^+$ from method invocations of method $m$ and its associated result, to be more precise $\leadsto_m^+$ is a transitive closure of $\leadsto_m$. Then, for instructions that may not throw an exception, we remove the subscript $\{m, \mathrm{Norm}\}$ from $\leadsto$ because it is clear that they have no exception throwing operational semantic counterpart. A list of operational semantics are contained in Figure 2. We do not show the full list of operational semantics due to space limitations. However, the interested reader can see Figure 7 in Appendix C for the full list of JVM operational semantics.

**Successor Relation** The successor relation $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ of a program $P$ are tagged with whether the execution is normal or throwing an exception. According to the types of instructions at program point $i$, there are several possibilities:

- $P_m[i] = \mathbf{goto}\ t$. The successor relation is $i \mapsto^{\mathrm{Norm}} t$

- $P_m[i] = \mathbf{ifeq}\ t$. In this case, there are 2 successor relations denoted by $i \mapsto^{\mathrm{Norm}} i + 1$ and $i \mapsto^{\mathrm{Norm}} t$.

- $P_m[i] = \mathbf{return}$. In this case it is a return point denoted by $i \mapsto^{\mathrm{Norm}}$

- $P_m[i]$ 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 $i \mapsto^{\mathrm{np}} t$.

$$\frac{P_m[i] = \textbf{push } n}{\langle i, \rho, os \rangle \rightsquigarrow \langle i+1, \rho, n :: os \rangle} \qquad \frac{P_m[i] = \textbf{pop}}{\langle i, \rho, v :: os \rangle \rightsquigarrow \langle i+1, \rho, os \rangle} \qquad \frac{P_m[i] = \textbf{return}}{\langle i, \rho, v :: os \rangle \rightsquigarrow v, h} \qquad \frac{P_m[i] = \textbf{goto } j}{\langle i, \rho, os \rangle \rightsquigarrow \langle j, \rho, os \rangle}$$

$$\frac{P_m[i] = \textbf{store } x \quad x \in \textbf{dom}(\rho)}{\langle i, \rho, v :: os \rangle \rightsquigarrow \langle i+1, \rho \oplus \{x \mapsto v\}, os \rangle} \qquad \frac{P_m[i] = \textbf{load } x}{\langle i, \rho, os \rangle \rightsquigarrow \langle i+1, \rho, \rho(x) :: os \rangle} \qquad \frac{P_m[i] = \textbf{binop } op \quad n_2 \underline{op} \, n_1 = n}{\langle i, \rho, n_1 :: n_2 :: os \rangle \rightsquigarrow \langle i+1, \rho, n :: os \rangle}$$

$$\frac{P_m[i] = \textbf{swap}}{\langle i, \rho, v_1 :: v_2 :: os \rangle \rightsquigarrow \langle i+1, \rho, v_2 :: v_1 :: os \rangle} \qquad \frac{P_m[i] = \textbf{ifeq } j \quad n \neq 0}{\langle i, \rho, n :: os \rangle \rightsquigarrow \langle i+1, \rho, os \rangle} \qquad \frac{P_m[i] = \textbf{ifeq } j \quad n = 0}{\langle i, \rho, n :: os \rangle \rightsquigarrow \langle j, \rho, os \rangle}$$

Fig. 2: JVM Operational Semantic (Selected)

- $P_m[i]$ 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 $i \mapsto^{\text{np}}$.

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

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

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

- $P_m[i] = \textbf{invoke } m_{\text{ID}}$, throwing an exception $C \in \textbf{excAnalysis}(m_{\text{ID}})$, and $\textbf{Handler}(i, C) \uparrow$. 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 $i \mapsto^{norm} i+1$

**Typing Rules** The security level is defined as a partially ordered set $(\mathcal{S}, \leq)$ of security levels $\mathcal{S}$ that form a lattice. $\sqcup$ denotes the lub of two security levels, and for every $k \in \mathcal{S}$, $\textbf{lift}_k$ 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 level 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 of security levels $\mathcal{S}$. The stack type itself takes the form of a stack with corresponding indices from the operand stack, as shown below.
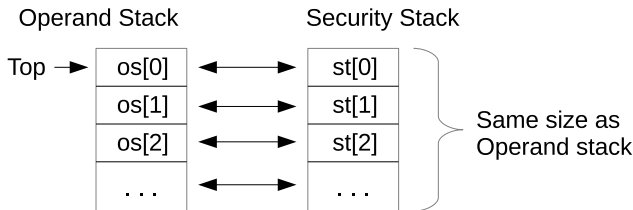


We assume that a method comes with its security policy of the form $\vec{k_a} \xrightarrow{k_h} \vec{k_r}$ where $\vec{k_a}$ represents a list $\{v_1 : k_1, \ldots, v_n : k_n\}$ with $k_i \in \mathcal{S}$ being the security level of local variables

$v_i \in \mathcal{R}$, $k_h$ is the effect of the method on the heap and $\vec{k_r}$ is the return signature, i.e. the security levels of the return value. The return signature is of 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 $\{\text{Norm} : k_n, e_1 : k_{e_1}, \ldots, e_n : k_{e_n}\}$ 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 $\vec{k_r}[n]$ to stand for $k_n$ 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 $L, H \in \mathcal{S}, L \leq k_{\text{obs}}, H \nleq k_{\text{obs}}$ which indicates 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.

Arrays have an extended security level than that of the usual object or value to cater for the security level of the contents. The security level of an array will be of the form $k[k_c]$ where $k$ represents the security level of an array and $k_c$ will represent the security level of its content (this implies that all array elements have the same security level $k_c$). Denote $\mathcal{S}^{\text{ext}}$ as 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 $k \leq k_{\text{obs}}$. Only $k_{\text{obs}}$ and $se$ (defined later) will stay in the form of $\mathcal{S}$, everything else will be extended to also include the extended level $\mathcal{S}^{\text{ext}}$.

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

$$\frac{P_m[i] = \textbf{load } x}{se, i \vdash st \Rightarrow \big(\vec{k_v}(x) \sqcup se(i)\big) :: st} \qquad \frac{P[i]_m = \textbf{store } x \quad se(i) \sqcup k \le \vec{k_a}(x)}{se, i \vdash k :: st \Rightarrow st} \qquad \frac{P_m[i] = \textbf{swap}}{i \vdash k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st}$$

$$\frac{P[i]_m = \textbf{ifeq } j \quad \forall j' \in \textbf{region}(i, \text{Norm}), k \le se(j')}{\textbf{region}, se, i \vdash k :: st \Rightarrow \textbf{lift}_\textbf{k}(st)} \qquad \frac{P_m[i] = \textbf{goto } j}{i \vdash st \Rightarrow st} \qquad \frac{P_m[i] = \textbf{push } n}{i \vdash st \Rightarrow se(i) :: st}$$

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

Fig. 3: JVM Transfer Rule (Selected)

signature $\Gamma_{m_{\text{ID}}}[k]$. The collection of security signatures of a method $m$ is defined as $\textbf{Policies}_\Gamma(m_{\text{ID}}) = \{\Gamma_{m_{\text{ID}}}[k] \mid k \in \mathcal{S}\}$.

A method is also parameterized by a control dependence region (CDR) which is defined in terms of two functions: **region** and **jun**. The function $\textbf{region} : \mathcal{PP} \to \wp(\mathcal{PP})$ 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 $\textbf{region}(i)$ the guard will be program point $i$. The function $\textbf{jun}(i)$ itself can be seen as the nearest program point which all instructions in $\textbf{region}(i)$ have to execute (junction point). A CDR is safe if it satisfies the following SOAP (Safe Over APproximation) properties.

**Definition III.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 $\tau$ such that $i \mapsto j$ and $i \mapsto^\tau k$ and $j \ne k$ ($i$ is hence a branching point), $k \in \textbf{region}(i, \tau)$ or $k = \textbf{jun}(i, \tau)$.*

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

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

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

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

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

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;

$\textbf{ft}$: a map from fields to their global policy level;

CDR: a structure consists of ($\textbf{region}, \textbf{jun}$).

$se$: security environment

$sgn$: method signature of the current method

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

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

although 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 (for the full list of transfer rules, see Figure 8 in Appendix C). Using these transfer rules, we can then define the notion of typability:

**Definition III.2** (Typable method)**.** *A method $m$ is typable w.r.t. a method signature table $\Gamma$, a global field policy $\textbf{ft}$, a policy $sgn$, and a CDR $\textbf{region}_\textbf{m} : \mathcal{PP} \to \wp(\mathcal{PP})$ 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 $\Gamma, \textbf{ft}, \textbf{region}, se, sgn, i \vdash^e S_i \Rightarrow st$ and $st \sqsubseteq S_j$;*

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

*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 whenever given indistinguishable inputs, it yields indistinguishable outputs. To cater for this definition, first there are definitions of indistinguishability.

To define the notions of location, object, and array indistinguishability itself 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 the first state to low objects allocated in the heap of the second state. The object might be indistiguishable, even if their locations are different during execution.

**Definition III.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 III.4** (Local variables indistinguishability). *For $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$, we have $\rho \sim_{k_{\mathrm{obs}}, \vec{k_a}, \beta} \rho'$ if $\rho$ and $\rho'$ have the same domain and $\rho(x) \sim_\beta \rho'(x)$ for all $x \in \mathbf{dom}(\rho)$ such that $\vec{k_a}(x) \leq k_{\mathrm{obs}}$.*

**Definition III.5** (Object indistinguishability). *Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (noted by $o_1 \sim_{k_{\mathrm{obs}}, \beta} o_2$) 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. $\mathbf{ft}(f) \leq k_{\mathrm{obs}}$.*

**Definition III.6** (Array indistinguishability). *Two arrays $a_1, a_2 \in \mathcal{A}$ are indistinguishable w.r.t. an attacker level $k_{\mathrm{obs}}$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (noted by $a_1 \sim_{k_{\mathrm{obs}}, \beta} o_2$) if and only if $a_1.\mathbf{length} = a_2.\mathbf{length}$ and, moreover, if $\mathbf{at}(a_1) \leq k_{\mathrm{obs}}$, then $a_1[i] \sim_\beta a_2[i]$ for all $i$ such that $0 \leq i < a_1.\mathbf{length}$.*

**Definition III.7** (Heap indistinguishability). *Two heaps $h_1$ and $h_2$ are indistinguishable, written $h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2$, with respect to an attacker level $k_{\mathrm{obs}}$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ iff:*

- *$\beta$ is a bijection between $\mathbf{dom}(\beta)$ and $\mathbf{rng}(\beta)$;*

- *$\mathbf{dom}(\beta) \subseteq \mathbf{dom}(h_1)$ and $\mathbf{rng}(\beta) \subseteq \mathbf{dom}(h_2)$;*

- *$\forall l \in \mathbf{dom}(\beta), h_1(l) \sim_{k_{\mathrm{obs}}, \beta} h_2(\beta(l))$ where $h_1(l)$ and $h_2(\beta(l))$ are either two objects or two arrays.*

**Definition III.8** (Output indistinguishability). *Given an attacker level $k_{\mathrm{obs}}$, 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 :*

$$\frac{h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2 \quad \vec{k_r}[n] \leq k_{\mathrm{obs}} \to v_1 \sim_\beta v_2}{(v_1, h_1) \sim_{k_{\mathrm{obs}}, \beta, \vec{k_r}} (v_2, h_2)}$$

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

$$\frac{h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2 \quad (\mathbf{class}(h_1(l_1)) : k) \in \vec{k_r} \quad k \nleq k_{\mathrm{obs}}}{(\langle l_1 \rangle, h_1) \sim_{k_{\mathrm{obs}}, \beta, \vec{k_r}} (v_2, h_2)}$$

$$\frac{h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2 \quad (\mathbf{class}(h_2(l_2)) : k) \in \vec{k_r} \quad k \nleq k_{\mathrm{obs}}}{(v_1, h_1) \sim_{k_{\mathrm{obs}}, \beta, \vec{k_r}} (\langle l_2 \rangle, h_2)}$$

$$\frac{h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2 \quad (\mathbf{class}(h_1(l_1)) : k_1) \in \vec{k_r} \quad k_1 \nleq k_{\mathrm{obs}}}{(\mathbf{class}(h_2(l_2)) : k_2) \in \vec{k_r} \quad k_2 \nleq k_{\mathrm{obs}}}{(\langle l_1 \rangle, h_1) \sim_{k_{\mathrm{obs}}, \beta, \vec{k_r}} (\langle l_2 \rangle, h_2)}$$

*where $\to$ indicates logical implication.*

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 $k_{\mathrm{obs}}$, instead of writing $o_1 \sim_{k_{\mathrm{obs}}, \beta} o_2$ we may drop $k_{\mathrm{obs}}$ and write $o_1 \sim_\beta o_2$ if $k_{\mathrm{obs}}$ is obvious. We may also drop $k_h$ from a policy $\vec{k_a} \xrightarrow{k_h} \vec{k_r}$ and write $\vec{k_a} \to \vec{k_r}$ if $k_h$ is irrelevant to the discussion.

**Definition III.9** (Non-interferent JVM method). *A method $m$ is non-interferent w.r.t. a policy $\vec{k_a} \to \vec{k_r}$, if for every attacker level $k_{\mathrm{obs}}$, 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.*

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

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

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

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

**Definition III.10** (Side effect preorder). *Two heaps $h_1, h_2 \in \mathbf{Heap}$ are side effect preordered (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 $k \nleq \mathbf{ft}(f)$.*

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

**Definition III.11** (Side effect safe). *A method $m$ is side-effect-safe with respect to a security level $k_h$ if for all local variables $x \in \mathbf{dom}(\rho), \rho \in \mathcal{X} \rightharpoonup \mathcal{V}$, all heaps $h, h' \in \mathbf{Heap}$ and value $v \in \mathcal{V}, \langle 1, \rho, \epsilon, h \rangle \leadsto_m^+ v, h'$ implies $h \preceq_{k_h} h'$.*

**Definition III.12** (Safe JVM method). *A method $m$ is safe w.r.t. a policy $\vec{k_a} \xrightarrow{k_h} \vec{k_r}$ 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 III.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 III.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. $\Gamma$.*

## IV. DEX TYPE SYSTEM

A program *P* is given by its list of instructions in Figure 4. The set $\mathcal{R}$ is the set of DEX virtual registers, $\mathcal{V}$ is the set of values, and $\mathcal{PP}$ is the set of program points. Since the DEX translation involves simulation of the JVM which uses a stack, we will also distinguish the registers :

- registers used to store the local variables,

- registers used to store parameters,

- and registers used to simulate the stack.

In practice, there is no difference between registers used to simulate the stack and those that are used to store local variables. The translation of a JVM method refers to code which assumes that the parameters are already copied to the local variables.

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

$$\frac{P_m[i] = \mathbf{const}(r,v) \quad r \in \mathbf{dom}(\rho)}{\langle i,\rho,h\rangle \rightsquigarrow \langle i+1, \rho \oplus \{r \mapsto v\}, h\rangle} \qquad \frac{P[i]_m = \mathbf{ifeq}(r,j) \quad \rho(r) = 0}{\langle i,\rho,h\rangle \rightsquigarrow \langle t,\rho,h\rangle} \qquad \frac{P_m[i] = \mathbf{ifeq}(r,t) \quad \rho(r) \neq 0}{\langle i,\rho,h\rangle \rightsquigarrow \langle i+1,\rho,h\rangle}$$

$$\frac{P[i]_m = \mathbf{return}(r_s) \quad r_s \in \mathbf{dom}(\rho)}{\langle i,\rho,h\rangle \rightsquigarrow \rho(r_s), h} \qquad \frac{P_m[i] = \mathbf{move}(r,r_s) \quad r \in \mathbf{dom}(\rho)}{\langle i,\rho,h\rangle \rightsquigarrow \langle i+1, \rho \oplus \{r \mapsto \rho(r_s)\}, h\rangle} \qquad \frac{P_m[i] = \mathbf{goto}(t)}{\langle i,\rho,h\rangle \rightsquigarrow \langle t,\rho,h\rangle}$$

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

Fig. 5: DEX Operational Semantic (Selected)

| | | |
|---|---|---|
| **binop** $op$ | $r, r_a, r_b$ | $\rho(r) = \rho(r_a) op \; \rho(r_b)$. |
| **const** | $r, v$ | $\rho(r) = v$ |
| **move** | $r, r_s$ | $\rho(r) = \rho(r_s)$ |
| **ifeq** | $r, t$ | conditional jump if $\rho(r) = 0$ |
| **ifneq** | $r, t$ | conditional jump if $s\rho(r) \neq 0$ |
| **goto** | $t$ | unconditional jump |
| **return** | $r_s$ | return the value of $\rho(r_s)$ |
| **new** | $r, c$ | $\rho(r) = $ new object of class $c$ |
| **iget** | $r, r_o, f$ | $\rho(r) = \rho(r_o).f$ |
| **iput** | $r_s, r_o, f$ | $\rho(r_o).f = \rho(r)$ |
| **newarray** | $r, r_l, t$ | $r = $ new array of type $t$ with $r_l$ number of elements |
| **arraylength** | $r, r_a$ | $\rho(r) = \rho(r_a).\mathbf{length}$ |
| **aput** | $r_s, r_a, r_i$ | $\rho(r_a)[\rho(r_i)] = \rho(r_s)$ |
| **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 $r$. Must be placed directly after invoke |
| **throw** | $r$ | throw the exception in $r$ |
| **move–exception** | $r$ | store exception in $r$. Have to be the first in the handler. |

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 $\rho : \mathcal{R} \to \mathbb{Z}$.

Fig. 4: DEX Instruction List

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 contains the list of DEX instructions.

**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. As for the JVM in handling the method invocation, operational semantics are also extended to have a big step semantics for the method invoked. Figure 5 shows some of the operational semantics for DEX instructions. Refer to Figure 9 in Appendix D for a full list of DEX operational semantics.

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.

**Type Systems** The transfer rules of DEX are defined in terms of registers typing $rt : (\mathcal{R} \to \mathcal{S})$ instead of stack typing. Note that this registers typing is total w.r.t. the registers used in the method. 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 65535 registers.

The transfer rules also come equipped with a security policy for fields $\mathbf{ft} : \mathcal{F} \to \mathcal{S}^{\text{ext}}$ and $\mathbf{at} : \mathcal{PP} \to \mathcal{S}^{\text{ext}}$. Some of the transfer rules for DEX instructions are contained in Figure 6. Full transfer rules are contained in Appendix D.

The typability of the DEX closely follows that of the JVM, except that the relation between program points is $i \vdash RT_i \Rightarrow rt, rt \sqsubseteq RT_j$. The definition of $\sqsubseteq$ is also defined in terms of point-wise registers. For now we assume the existence of 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. Formal definition of typable DEX method:

**Definition IV.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 $\mathbf{region_m} : \mathcal{PP} \to \wp(\mathcal{PP})$ if there exists a security environment $se : \mathcal{PP} \to \mathcal{S}$ and a function $\mathbf{RT} : \mathcal{PP} \to (\mathcal{R} \to \mathcal{S})$ 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 $rt \in (\mathcal{R} \to \mathcal{S})$ 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

$$\frac{P_m[i] = \mathbf{const}(r, v)}{i \vdash rt \Rightarrow rt \oplus \{r \mapsto se(i)\}} \qquad \frac{P_m[i] = \mathbf{return}(r_s) \quad se(i) \sqcup rt(r_s) \le \vec{k_r}[n]}{i \vdash rt \Rightarrow}$$

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

$$\frac{P_m[i] = \mathbf{goto}(j)}{i \vdash rt \Rightarrow rt} \qquad \frac{P_m[i] = \mathbf{ifeq}(r, t) \quad \forall_{j'} \in \mathbf{region}(i, \mathrm{Norm}), se(i) \sqcup rt(r) \le se(j')}{i \vdash rt \Rightarrow rt}$$

Fig. 6: DEX Transfer Rule (Selected)

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 IV.2** (Registers indistinguishability)**.** *For $\rho, \rho'$ : $(\mathcal{R} \rightharpoonup \mathcal{V})$ and $rt, rt' : (\mathcal{R} \rightharpoonup \mathcal{S})$, we have $\rho \sim_{k_{\mathrm{obs}}, rt, rt', \beta} \rho'$ iff $\forall x \notin locR$, $rt(x) = rt'(x) = k, k \not\le k_{\mathrm{obs}}$ or $rt(x) = k$, $rt'(x) = k'$, $k \le k_{\mathrm{obs}}$, $k' \le k_{\mathrm{obs}}$, $\rho(x) = \rho'(x) = v$. where $v \in \mathcal{V}$, and $k, k' \in \mathcal{S}$.*

**Definition IV.3** (Object indistinguishability)**.** *Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (noted by $o_1 \sim_{k_{\mathrm{obs}}, \beta} o_2$) 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. $\mathbf{ft}(f) \le k_{\mathrm{obs}}$.*

**Definition IV.4** (Array indistinguishability)**.** *Two arrays $a_1, a_2 \in \mathcal{A}$ are indistinguishable w.r.t. an attacker level $k_{\mathrm{obs}}$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (noted by $a_1 \sim_{k_{\mathrm{obs}}, \beta} o_2$) if and only if $a_1.\mathbf{length} = a_2.\mathbf{length}$ and, moreover, if $\mathbf{at}(a_1) \le k_{\mathrm{obs}}$, then $a_1[i] \sim_\beta a_2[i]$ for all $i$ such that $0 \le i < a_1.\mathbf{length}$.*

**Definition IV.5** (Heap indistinguishability)**.** *Two heaps $h_1$ and $h_2$ are indistinguishable with respect to an attacker level $k_{\mathrm{obs}}$ and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, written $h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2$, 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 $\mathbf{rng}(\beta) \subseteq \mathbf{dom}(h_2)$;*

- *$\forall l \in \mathbf{dom}(\beta), h_1(l) \sim_{k_{\mathrm{obs}}, \beta} h_2(\beta(l))$ where $h_1(l)$ and $h_2(\beta(l))$ are either two objects or two arrays.*

**Definition IV.6** (Output indistinguishability)**.** *Given an attacker level $k_{\mathrm{obs}}$, 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 :*

$$\frac{h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2 \quad \vec{k_r}[n] \le k_{\mathrm{obs}} \Rightarrow v_1 \sim_\beta v_2}{(v_1, h_1) \sim_{k_{\mathrm{obs}}, \beta, \vec{k_r}} (v_2, h_2)}$$

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

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

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

$$\frac{\begin{array}{c} h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2 \quad (\mathbf{class}(h_1(l_1)) : k_1) \in \vec{k_r} \quad k_1 \not\le k_{\mathrm{obs}} \\ (\mathbf{class}(h_2(l_2)) : k_2) \in \vec{k_r} \quad k_2 \not\le k_{\mathrm{obs}} \end{array}}{(\langle l_1 \rangle, h_1) \sim_{k_{\mathrm{obs}}, \beta, \vec{k_r}} (\langle l_2 \rangle, h_2)}$$

*where $\rightarrow$ indicates logical implication.*

**Definition IV.7** (Non-interferent DEX method)**.** *A method $m$ is non-interferent w.r.t. a policy $\vec{k_a} \rightarrow \vec{k_r}$, if for every attacker level $k_{\mathrm{obs}}$, every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}, h_1, h_2, h_1', h_2' \in \mathbf{Heap}, v_1, v_2 \in \mathcal{V} + \mathcal{L}$ s.t.*

$$\begin{array}{cc} \langle 1, \rho_1, h_1 \rangle \leadsto_m^+ v_1, h_1' & h_1 \sim_{k_{\mathrm{obs}}, \beta} h_2 \\ \langle 1, \rho_2, h_2 \rangle \leadsto_m^+ v_2, h_2' & \rho_1 \sim_{k_{\mathrm{obs}}, \vec{k_a}, \beta} \rho_2 \end{array}$$

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

$$(v_1, h_1') \sim_{k_{\mathrm{obs}}, \beta', \vec{k_a}} (v_2, h_2')$$

**Definition IV.8** (Side effect preorder)**.** *Two heaps $h_1, h_2 \in$ 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 $k \not\le \mathbf{ft}(f)$.*

**Definition IV.9** (Side effect safe)**.** *A method $m$ is side-effect-safe with respect to a security level $k_h$ if for all registers in $\rho \in \mathcal{R} \rightharpoonup \mathcal{V}, \mathbf{dom}(\rho) = locR$, for all heaps $h, h' \in \mathbf{Heap}$ and value $v \in \mathcal{V}, \langle 1, \rho, h \rangle \leadsto_m^+ v, h'$ implies $h \preceq_{k_h} h'$.*

**Definition IV.10** (Safe DEX method)**.** *A method $m$ is safe w.r.t. a policy $\vec{k_a} \xrightarrow{k_h} \vec{k_r}$ 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 IV.11** (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 $Policies_\Gamma(m)$.*

**Theorem IV.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. $\Gamma$.*

## V. EXAMPLES

Throughout our examples we will use two security levels $L$ and $H$ to indicate low and high security level respectively. We start with a simple example where a high guard is used to determine the value of a low variable.

**Example V.1.** Assume that local variable 1 is $H$ and local variable 2 is $L$. 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.

|  |  |  |  |
|---|---|---|---|
| | . . . | | . . . |
| | **push** 0 | | **const**$(r_3, 0)$ |
| | **store** 2 | | **move**$(r_2, r_3)$ |
| | **load** 1 | | **move**$(r_3, r_1)$ |
| | **ifeq** $l_1$ | | **ifeq**$(r_3, l_1)$ |
| | **push** 1 | | **const**$(r_3, 1)$ |
| | **store** 2 | | **move**$(r_2, r_3)$ |
| $l_1 :$ | . . . | $l_1 :$ | . . . |

In this case, the type system for the JVM bytecode will reject this example because there is a violation in the last **store** 2 constraint. The reasoning is that $se(i)$ for **push** 1 will be $H$, thus the constraint will be $H \sqcup H \le L$ which can not be satisfied. The same goes for the DEX instructions. Since $r_3$ gets its value from $r_1$ which is $H$, the typing rule for **ifeq**$(r_3, l_1)$ states that $se$ in the last instruction will be $H$. Since the last **move** instruction is targetting variables in the local variables side, the constraint applies, which states that $H \sqcup H \le L$ which also can not be satisfied, thus this program will be rejected by the DEX type system.

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 V.2.** Assume that $\vec{k_a} = \{r_1 \mapsto H, r_2 \mapsto H, r_3 \mapsto L\}$ (which means local variable 1 is high, local variable 2 is high and local variable 3 is low). Also the field $f$ is low ($\mathbf{ft}(f) = L$).

|  |  |  |  |
|---|---|---|---|
| | . . . | | . . . |
| | **new** $C$ | | **new**$(r_4, C)$ |
| | **store** 3 | | **move**$(r_3, r_4)$ |
| | **load** 2 | | **move**$(r_4, r_2)$ |
| $l_1 :$ | **ifeq** $l_2$ | $l_1 :$ | **ifeq**$(r_4, l_2)$ |
| | **new** $C$ | | **new**$(r_4, C)$ |
| | **goto** $l_3$ | | **goto**$(l_3)$ |
| $l_2 :$ | **load** 3 | $l_2 :$ | **move**$(r_4, r_3)$ |
| $l_3 :$ | **store** 1 | $l_3 :$ | **move**$(r_1, r_4)$ |
| | **load** 1 | | **move**$(r_4, r_1)$ |
| | **push** 1 | | **const**$(r_5, 1)$ |
| | **putfield** $f$ | | **iput**$(r_5, r_4, f)$ |
| | . . . | | . . . |

The above JVM bytecode and its translation will be rejected by the type system for the JVM bytecode because for **putfield** $f$ at the last line there is a constraint with the security level of the object. In this case, the **load** 1 instruction will push a reference of the object with high security level, therefore, the constraint that $L \sqcup H \sqcup L \le L$ can not 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 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 $L \sqcup H \sqcup L \le L$.

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

**Example V.3.** Assume that $\vec{k_a} = \{r_1 \mapsto H, r_2 \mapsto L, r_3 \mapsto H\}$ and $\vec{k_r} = \{n \mapsto L, e \mapsto H, \mathrm{np} \mapsto H\}$. **Handler**$(l_2, \mathrm{np}) = l_h$, and for any $e$, **Handler**$(l_2, e) \uparrow$. The following JVM bytecode and its translation will be rejected by the typing system for the JVM bytecode.

|  |  |  |  |
|---|---|---|---|
| | . . . | | . . . |
| | **load** 1 | | **move**$(r_4, r_1)$ |
| $l_1 :$ | **ifeq** $l_2$ | $l_1 :$ | **ifeq**$(r_4, l_2)$ |
| | **new** $C$ | | **new**$(r_4, C)$ |
| | **store** 3 | | **move**$(r_3, r_4)$ |
| | **load** 3 | | **move**$(r_4, r_3)$ |
| $l_2 :$ | **invokevirtual** $m$ | $l_2 :$ | **invoke**$(1, m, r_4)$ |
| | **push** 0 | | **const**$(r_4, 0)$ |
| | **return** | $ret$ | **move**$(r_0, r_4)$ |
| | | | **return**$(r_0)$ |
| $l_h :$ | **push** 1 | $l_h :$ | **const**$(r_4, r_1)$ |
| | **return** | | **move**$(r_0, r_4)$ |
| | | | **goto**$(ret)$ |
| | . . . | | . . . |

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** 2 (line 8 and 10) the constraint 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 store the value in local variable 2 (constraint $H \le L$ is violated, where $H$ comes from lub with $se$).

## VI. TRANSLATION PHASE

Before we continue to describe the translation processes, we find it helpful to first define a construct called the 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 parents list, successors list, primary succesor, and its order in the output phase. There are also some auxilliary functions :

| | |
|---|---|
| BMap | is a mapping function from program pointer in JVM bytecode to a DEX basic block. |
| SBMap | Similar to BMap, this function takes a program pointer in JVM bytecode and returns whether that instruction is the start of a DEX basic block. |
| TSMap | A function that maps a program pointer in JVM bytecode to an integer denoting the index to the top of the stack. Initialized with the number of local variables as that index is the index which will be used by DEX to simulate the stack. |
| NewBlock | A function to create a new Block and associate it with the given instruction. |

The DEX bytecode is simulating the JVM bytecode although they have different infrastructure. DEX is register-based

whereas JVM is stack-based. To bridge this gap, DEX uses registers to simulate the stack. The way it works :

- $l$ number of registers are used to hold local variables. $(1, \ldots, l)$. We denote these registers with $locR$.

- Immediately after $l$, there are $s$ number of registers which are used to simulate the stack $(l + 1, \ldots, l + s)$.

Note that although in principal stack can grow indefinitely, it is impossible to write a program that does so in Java, due to the strict stack discipline in Java. Given a program in JVM bytecode, it is possible to statically determine the height of the operand stack at each program point. This makes it possible to statically map each operand stack location to a register in DEX (cf. TSMap above and Appendix A-C); see [25] for a discussion on how this can be done.

There are several phases involved to translate JVM bytecode into DEX bytecode:

**StartBlock**: 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.

**TraceParentChild**: Resolves the parents successors (and primary successor) relationship between blocks. 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 in time, 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. Then **return** will just return $r_0$.

**Translate**: Translate JVM instructions into DEX instructions.

**PickOrder**: Order blocks according to "trace analysis".

**Output**: Output the instructions in order. During this phase, **goto** will be added for each block whose next block to output is not its successor. After the compiler has output all blocks, it will then read the list of DEX instructions and fix up the targets of jump instructions. Finally, all the information about exception handlers is collected and put in the section that deals with exception handlers in the DEX file structure.

**Definition VI.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 $\lfloor P \rfloor$, where*

$$\lfloor P \rfloor = \textbf{Translate}(\textbf{TraceParentChild}(\textbf{StartBlock}(P))).$$

**Definition VI.2** (Output Translated Program). *The output of the translated JVM program $\lfloor P \rfloor$ in which the blocks are ordered and then output into DEX program is denoted by $\lceil \lfloor P \rfloor \rceil$, where*

$$\lceil \lfloor P \rfloor \rceil = \textbf{Output}(\textbf{PickOrder}(\lfloor P \rfloor)).$$

**Definition VI.3** (Compiled JVM Program). *The compilation of a JVM program $P$ is denoted by $\llbracket P \rrbracket$, where*

$$\llbracket P \rrbracket = \lceil \lfloor P \rfloor \rceil.$$

Details for each of the phase can be seen in appendix A.

## VII. Proof that Translation Preserves Typability

### A. 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 VII.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 translation of **region** and **jun** since we assume that the JVM bytecode comes equipped with **region** and **jun**.

**Definition VII.2** (Region Translation and Compilation). *Given a JVM $\textbf{region}(i, \tau)$ and $P[i]$ is a branching instruction, let $i_b$ be the program point in $\lfloor i \rfloor$ such that $P_{\text{DEX}}[i_b]$ is a branching instruction, then*

$$\lfloor \textbf{region}(i, \tau) \rfloor = \textbf{region}(i_b, \lfloor \tau \rfloor) = \bigcup_{j \in \textbf{region}(i, \tau)} \lfloor j \rfloor$$
*and*
$$\llbracket \textbf{region}(i, \tau) \rrbracket = \textbf{region}(i_b, \llbracket \tau \rrbracket) = \bigcup_{j \in \textbf{region}(i, \tau)} \llbracket j \rrbracket$$

**Definition VII.3** (Region Translation and Compilation for **invoke**). *$\forall i. P_{DEX}[i] = \textbf{invoke}$, $i + 1 \in \textbf{region}(i, \text{Norm})$ ($i + 1$ will be the program point for **moveresult**).*

**Definition VII.4** (Region Translation and Compilation for handler). *$\forall i, j. j \in \textbf{region}(i, \tau)$, let $i_e$ be the instruction in $\lfloor P[i] \rfloor$ that possibly throws, then*

$$\textbf{handler}(i_e, \tau) \in \textbf{region}(i_e, \tau) \text{ in } \lfloor P \rfloor$$
*and*
$$\textbf{handler}(\llbracket i_e \rrbracket, \tau) \in \textbf{region}(\llbracket i_e \rrbracket, \tau) \text{ in } \llbracket P \rrbracket$$

*(note that the handler will point to **moveexception**).*

**Definition VII.5** (Region for appended **goto** instruction).

$$\forall b \in \lfloor P \rfloor. \quad P_{DEX}[\llbracket b.\textbf{lastAddress} \rrbracket + 1] = \textbf{goto}$$
$$\rightarrow \big(\forall.i \in \mathcal{PP}_{DEX}.b.\textbf{lastAddress} \in \textbf{region}(i, \tau)$$
$$\rightarrow (\llbracket b.\textbf{lastAddress} \rrbracket + 1) \in \textbf{region}(i, \tau)\big)$$

*where $\rightarrow$ indicates logical implication.*

**Definition VII.6** (Junction Translation and Compilation). *$\forall i, j. j = \textbf{jun}(i, \tau)$, let $i_b$ be in $\lfloor P[i] \rfloor$ that branch then*

$$\lfloor j \rfloor[0] = \textbf{jun}(\lfloor i \rfloor[i_b], \tau) \text{ in } \lfloor P \rfloor$$
*and*
$$\llbracket \lfloor j \rfloor[0] \rrbracket = \textbf{jun}(\llbracket \lfloor i \rfloor[i_b] \rrbracket, \tau) \text{ in } \llbracket P \rrbracket.$$

**Definition VII.7** (Security Environment Translation and Compilation). *$\forall i \in \mathcal{PP}, j \in \lfloor i \rfloor. se(j) = se(i) \text{ in } \lfloor P \rfloor$ and $\forall i \in \mathcal{PP}, j \in \lfloor i \rfloor. se(\llbracket j \rrbracket) = se(i) \text{ in } \llbracket P \rrbracket.$*

**Lemma VII.1** (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.*

### B. Compilation Preserves Typability

There are several assumption 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 $\lfloor . \rfloor$ and $\llbracket . \rrbracket$ to also apply to stack type to denote 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), and fill the rest with high security level. 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), and the method has $o$ registers (which corresponds to the maximum depth of the stack), then $\llbracket st \rrbracket = \{r_0 \mapsto \vec{k}_a(v_1), \ldots, r_{n-1} \mapsto \vec{k}_a(v_n), r_n \mapsto st[m-1], \ldots, r_{n+m-1} \mapsto st[0], r_{n+m} \mapsto H, \ldots, r_o \mapsto H\}$. Lastly, the function $\llbracket . \rrbracket$ 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.

Due to the way stack type is translated to registers typing, we find it beneficial to introduce a simple lemma that can be proved trivially (by structural induction and definition) in regards to the $rt_1 \sqsubseteq rt_2$ relation. In particular this lemma will relates the registers metioned in $rt_1$ but are not mentioned in $rt_2$.

**Lemma VII.2** (Registers Not in Stack Less Equal). *Let the number of local variables be $locN$. For any two stack types $st_1, st_2, \mathbf{length}(st_1) = n, \mathbf{length}(st_2) = m, m < n$, any register $x \in \{r_{locN+m+1}, \ldots, r_{locN+m+n}\}$, and register types $rt_1 = \lfloor st_1 \rfloor, rt_2 = \lfloor st_2 \rfloor$ we have that $rt_1(x) \leq rt_2(x)$.*

**Definition VII.8** ($\lfloor \mathbf{excAnalysis} \rfloor$ and $\llbracket \mathbf{excAnalysis} \rrbracket$).

$$\forall m \in \mathcal{M}. \lfloor \mathbf{excAnalysis}(m) \rfloor = \mathbf{excAnalysis}(m) \text{ in } \lfloor P \rfloor$$
*and*
$$\forall m \in \mathcal{M}. \llbracket \mathbf{excAnalysis}(m) \rrbracket = \mathbf{excAnalysis}(m) \text{ in } \llbracket P \rrbracket.$$

**Definition VII.9** ($\lfloor \mathbf{classAnalysis} \rfloor$ and $\llbracket \mathbf{classAnalysis} \rrbracket$). *Let $e$ be the index of the throwing instruction from $\lfloor i \rfloor$.*

$$\left( \begin{array}{c} \forall m \in \mathcal{M}, i \in \mathcal{PP}. \lfloor \mathbf{classAnalysis}(m, \lfloor i \rfloor[e]) \rfloor = \\ \mathbf{classAnalysis}(m, i) \text{ in } \lfloor P \rfloor \end{array} \right)$$
*and*
$$\left( \begin{array}{c} \forall m \in \mathcal{M}, i \in \mathcal{PP}. \lfloor \mathbf{classAnalysis}(m, \llbracket \lfloor i \rfloor[e] \rrbracket) \rfloor = \\ \mathbf{classAnalysis}(m, i) \text{ in } \llbracket P \rrbracket \end{array} \right).$$

**Definition VII.10** ($\lfloor \Gamma \rfloor$ and $\llbracket \Gamma \rrbracket$). $\forall m \in \mathcal{M}. \lfloor \Gamma[\lfloor m \rfloor] \rfloor = \Gamma[m] \text{ in } \lfloor P \rfloor$ and $\forall m \in \mathcal{M}. \llbracket \Gamma[\llbracket m \rrbracket] \rrbracket = \Gamma[m] \text{ in } \llbracket P \rrbracket.$

**Definition VII.11.** $\forall i \in \mathcal{PP}, RT_{\lfloor i \rfloor[0]} = \lfloor S_i \rfloor.$

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 VII.3, Lemma VII.4 and Lemma VII.5 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 for each exception handler.

**Lemma VII.3.** *For any JVM program $P$ with instruction $Ins$ at address $i$ and tag $\mathrm{Norm}$, let the length of $\lfloor Ins \rfloor$ be $n$. Let $RT_{\lfloor i \rfloor[0]} = \lfloor S_i \rfloor$. If according to the transfer rule for $P[i] = Ins$ there exists $st$ s.t. $i \vdash^{\mathrm{Norm}} S_i \Rightarrow st$ then*

$$\left( \begin{array}{c} \forall 0 \leq j < (n-1). \exists rt'. \lfloor i \rfloor[j] \vdash^{\mathrm{Norm}} \\ RT_{\lfloor i \rfloor[j]} \Rightarrow rt', rt' \sqsubseteq RT_{\lfloor i \rfloor[j+1]} \end{array} \right)$$
*and*
$$\exists rt. \lfloor i \rfloor[n-1] \vdash^{\mathrm{Norm}} RT_{\lfloor i \rfloor[n-1]} \Rightarrow rt, rt \sqsubseteq \lfloor st \rfloor$$

*according to the typing rule(s) of $\lfloor Ins \rfloor$.*

**Lemma VII.4.** *For any JVM program $P$ with instruction $Ins$ at address $i$ and tag $\tau \neq \mathrm{Norm}$ with exception handler at address $i_e$. Let the length of $\lfloor Ins \rfloor$ until the instruction that throws exception $\tau$ be denoted by $n$. Let $(be, 0) = \lfloor i_e \rfloor$ be the address of the handler for that particular exception. If $i \vdash^\tau S_i \Rightarrow st$ according to the transfer rule for $Ins$, then*

$$\left( \begin{array}{c} \forall 0 \leq j < (n-1). \exists rt'. \lfloor i \rfloor[j] \vdash^{\mathrm{Norm}} \\ RT_{\lfloor i \rfloor[j]} \Rightarrow rt', rt' \sqsubseteq RT_{\lfloor i \rfloor[j+1]} \end{array} \right)$$
*and*
$$\exists rt. \lfloor i \rfloor[n-1] \vdash^\tau RT_{\lfloor i \rfloor[n-1]} \Rightarrow rt, rt \sqsubseteq RT_{(be, 0)}$$
*and*
$$\exists rt. (be, 0) \vdash^{\mathrm{Norm}} RT_{(be, 0)} \Rightarrow rt, rt \sqsubseteq \lfloor st \rfloor$$

*according to the typing rule(s) of the first $n$ instructions in $\lfloor Ins \rfloor$ and* **moveexception**.

**Lemma VII.5.** *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 $\lfloor Ins \rfloor$. Let $RT_{\lfloor i \rfloor[0]} = \lfloor S_i \rfloor$, let $RT_{\lfloor j \rfloor[0]} = \lfloor S_j \rfloor$ and $rt$ be registers typing obtained from the transfer rules involved in $\lfloor Ins \rfloor$. Then $rt \sqsubseteq RT_{\lfloor j \rfloor[0]}$.*

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 VII.6.** *Let $Ins$ be an instruction at program point $i$, $S_i$ its corresponding stack types, and let $RT_{\lfloor i \rfloor[0]} = \lfloor S_i \rfloor$. If $P[i]$ satisfy the typing constraint for $Ins$ with the stack type $S_i$, then $\forall (bj, j) \in \lfloor i \rfloor. P_{DEX}[bj, j]$ will also satisfy the typing constraints for all instructions in $\lfloor Ins \rfloor$ with the initial registers typing $RT_{\lfloor i \rfloor[0]}$.*

Using the above lemmas, we can prove the lemma that all the resulting blocks will also be typable in DEX.

**Lemma VII.7.** *Let $P$ be a Java program such that*

$$\forall i, j. i \mapsto j. \exists st. i \vdash S_i \Rightarrow st \quad \text{and} \quad st \sqsubseteq S_j$$

*Then $\lfloor P \rfloor$ 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*

$$\begin{array}{lll} RTs_{bi} = \|S_i\| & \text{with} & \|i\| = (bi, 0) \\ RTs_{bj} = \|S_j\| & \text{with} & \|j\| = (bj, 0), \\ RT_{(bi,i)} = \|S_{i'}\| & \text{with} & \|i'\| = (bi, i) \\ RT_{(bi,j)} = \|S_{j'}\| & \text{with} & \|j'\| = (bj, j). \end{array}$$

After we established that the translation into DEX instructions in the form of blocks preserves typability, we also need 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.

**Lemma VII.8.** *Let $\|P\|$ be typable basic blocks resulting from translation of JVM instructions still in the block form, i.e.*

$$\|P\| = \textbf{Translate}(\textbf{TraceParentChild}(\textbf{StartBlock}(P))).$$

*Given the ordering scheme to output the block contained in* **PickOrder***, if the starting block starts with flag 0 ($F_{(0,0)} = 0$) then the output $\|P\|$ is also typable.*

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 VII.7 and Lemma VII.8. Typable DEX bytecode will also have the non-interferent property because it is based on a safe CDR (Lemma VII.1) according to DEX.

**Theorem VII.1.** *Let $P$ be a typable JVM bytecode according to its safe CDR (***region***, ***jun***), PA-Analysis (***classAnalysis*** and ***excAnalysis***), and method policies $\Gamma$, then $\|P\|$ according to the translation scheme has the property that*

$$\forall i, j \in PP_{\text{DEX}}. \text{ s.t. } i \mapsto j.\exists rt. \text{ s.t. } RT_i \Rightarrow rt, rt \sqsubseteq RT_j$$

*according to a safe CDR ($\|$***region***\|$, $\|$***jun***\|$), $\|PA - Analysis\|$, and $\|\Gamma\|$.*

## VIII. Conclusion and Future Work

We presented the design of a type system for DEX programs and showed that the non-optimizing compilation done by the dx tool preserves the typability of JVM bytecode. Furthermore, the typability of the DEX program also implies its non-interference. We provide a proof-of-concept implementation illustrating the feasibility of the idea. This opens up the possibility of reusing analysis techniques applicable to Java bytecode for Android. As an immediate next step for this research, we plan to also take into account the optimization done in the dx tool to see whether typability is still preserved by the translation.

Our result is quite orthogonal to the Bitblaze project [26], 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 DEX bytecode can be unified 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 to our currentwork as well in that we still have

not identified special object and method invocation for this message passing mechanism in the bytecode.

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. This allows us to look at several sublanguages of DEX bytecode in isolation. The output of our custom compiler resembles the output from the dx compiler up to some details such as the size of register addressing. Following the Compcert project [27], [28], we would ultimately like to have a fully certified end to end compiler. We leave this as future work.

## References

[1] "Stat counter global stats," http://gs.statcounter.com/#mobile_os-ww-monthly-201311-201411, accessed: 2014-12-31.

[2] "DEX bytecode instructions," http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html, accessed: 2014-12-31.

[3] G. C. Necula, "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.* ACM Press, 1997, pp. 106–119. [Online]. Available: http://doi.acm.org/10.1145/263699.263712

[4] A. W. Appel, "Foundational proof-carrying code," in *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings.* IEEE Computer Society, 2001, pp. 247–256. [Online]. Available: http://dx.doi.org/10.1109/LICS.2001.932501

[5] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 1, pp. 5–19, 2003.

[6] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy.* IEEE Computer Society, 1982, pp. 11–11.

[7] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *J. Funct. Program.*, vol. 15, no. 2, pp. 131–177, Mar. 2005. [Online]. Available: http://dx.doi.org/10.1017/S0956796804005453

[8] G. Barthe, D. Pichardie, and T. Rezk, "A certified lightweight non-interference java bytecode verifier," *Mathematical Structures in Computer Science*, vol. 23, pp. 1032–1081, 10 2013. [Online]. Available: http://journals.cambridge.org/article_S0960129512000850

[9] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber, "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.* ACM, 2014, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2666620.2666631

[10] G. Barthe, D. Naumann, and T. Rezk, "Deriving an information flow checker and certifying compiler for java," in *Security and Privacy, 2006 IEEE Symposium on.* IEEE, 2006, pp. 13–pp.

[11] G. Barthe, T. Rezk, and A. Saabas, "Proof obligations preserving compilation," in *Formal Aspects in Security and Trust.* Springer, 2006, pp. 112–126.

[12] S. Lortz, H. Mantel, A. Starostin, and A. Weber, "A sound information-flow analysis for Cassandra," TU Darmstadt, Tech. Rep., 2014, technical Report TUD-CS-2014-0064.

[13] G. Bian, K. Nakayama, Y. Kobayashi, and M. Maekawa, "Java bytecode dependence analysis for secure information flow." *IJ Network Security*, vol. 4, no. 1, pp. 59–68, 2007.

[14] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[15] M. Bugliesi, S. Calzavara, and A. Spanò, "Lintent: towards security type-checking of android applications," in *Formal Techniques for Distributed Systems.* Springer, 2013, pp. 289–304.

[16] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones," *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, 2014.

[17] Z. Zhao and F. C. C. Osono, "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*. IEEE, 2012, pp. 135–143.

[18] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "Scandal: Static analyzer for detecting privacy leaks in android applications," *MoST*, 2012.

[19] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing Android's permission system," in *Computer Security—ESORICS 2012: 17th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 7459, Sep. 2012, pp. 1–18. [Online]. Available: http://www.ece.cmu.edu/~lbauer/papers/2012/esorics2012-android.pdf

[20] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake, "Run-time enforcement of information-flow properties on Android (extended abstract)," in *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security*. Springer, Sep. 2013, pp. 775–792. [Online]. Available: http://www.ece.cmu.edu/~lbauer/papers/2013/esorics2013-android.pdf

[21] A. P. Felt, H. Wang, A. Moschuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *20th USENIX Security Symposium*, 2011.

[22] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 235–245.

[23] W. Enck, M. Ongtang, P. D. McDaniel *et al.*, "Understanding android security." *IEEE security & privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[24] A. Chaudhuri, "Language-based security on android," in *Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security*. ACM, 2009, pp. 1–7.

[25] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron, "The case for virtual register machines," in *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, ser. IVME '03. New York, NY, USA: ACM, 2003, pp. 41–49. [Online]. Available: http://doi.acm.org/10.1145/858570.858575

[26] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Information systems security*. Springer, 2008, pp. 1–25.

[27] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," *ACM SIGPLAN Notices*, vol. 41, no. 1, pp. 42–54, 2006.

[28] S. Blazy, Z. Dargaye, and X. Leroy, "Formal verification of a c compiler front-end," in *FM 2006: Formal Methods*. Springer, 2006, pp. 460–475.

This section details compilation phases described in Section VI in more details. We first start this section by detailing the structure of basic block. $BasicBlock$ is a structure of

$$\{parents;\ succs;\ pSucc;\ order;\ insn\}$$

which denotes a structure of a basic block 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 $insn \cup DEXins$ is the DEX instructions contained in the block. The set of $BasicBlock$ is denoted as $BasicBlocks$. When instatiating the basic block, we denote the default object $NewBlock$, which will be a basic block with

$$\{parents = \varnothing;\ succs = \varnothing;\ pSucc = -1;\ order = -1;\ insn = \varnothing\}$$

Throughout the compilation phases, we also make use of two mappings $\mathbf{PMap} : \mathcal{PP} \to \mathcal{PP}$ and $\mathbf{BMap} : \mathcal{PP} \to BasicBlocks$. The mapping $\mathbf{PMap}$ is a mapping from a program point in JVM to a program point in JVM which starts a particular block, e.g. if we have a set of program points $\{2, 3, 4\}$ forming a basic block, then we have that $\mathbf{PMap}(2) = 2$, $\mathbf{PMap}(3) = 2$, and $\mathbf{PMap}(4) = 2$. $\mathbf{BMap}$ itself will be used to map a program point in JVM to a basic block.

### A. Indicate Instructions starting a Block (**StartBlock**)

This phase is done by sweeping through the JVM instructions (still in the form of a list). In the implementation, this phase will update the mapping startBlock. 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.

Case by case translation behaviour :

- $P[i]$ is Unconditional jump (goto $t$) : the target instruction will be a block starting point. There is implicit in this instruction that the next instruction should also be a start of the block, but this will be handled by another jump. We do not take care of the case where no jump instruction addresses this next instruction (the next instruction is a dead code), i.e.
  - $\mathbf{BMap} \oplus \{t \mapsto NewBlock\}$; and
  - $\mathbf{PMap} \oplus \{t \mapsto t\}$
- $P[i]$ is Conditional jump (ifeq $t$) : both the target instruction and the next instruction will be the start of a block, i.e.
  - $\mathbf{BMap} \oplus \{t \mapsto NewBlock, (i + 1) \mapsto NewBlock\}$; and
  - $\mathbf{PMap} \oplus \{t \mapsto t, (i + 1) \mapsto (i + 1)\}$.
- $P[i]$ is Return: the next instruction will be the start of a block. This instruction will update the mapping of the next instruction for $\mathbf{BMap}$ and $\mathbf{SBMap}$ 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, so the next instruction must be part of some execution path. To be more explicit, if there is a next instruction $i + 1$ then
  - $\mathbf{BMap} \oplus \{(i + 1) \mapsto NewBlock\}$; and
  - $\mathbf{PMap} \oplus \{(i + 1) \mapsto (i + 1)\}$
- $P[i]$ is an instruction which may throw an exception : just like return instruction, the next instruction will be the start of a new block. 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, its:

startPC  program counter (pc) which serves as the starting point (inclusive) of which the exception handler is active;

endPC  program counter which serves as the ending point (exclusive) of which the exception handler is active; and

handlerPC  program counter which points to the start of the exception handler

are indicated as starting of a block. For handler $h$, the intermediary block will have label $intPC = maxLabel + h.\mathbf{handlerPC}$. To reduce clutter, we write $sPC$ to stand for $h.\mathbf{startPC}$, $ePC$ to stand for $h.\mathbf{endPC}$, and $hPC$ to stand for $h.\mathbf{handlerPC}$.
  - $\mathbf{BMap} \oplus \{(i + 1) \mapsto NewBlock\}$;
  - $\mathbf{BMap} \oplus \{sPC \mapsto NewBlock\}$;
  - $\mathbf{BMap} \oplus \{ePC \mapsto NewBlock\}$;
  - $\mathbf{BMap} \oplus \{hPC \mapsto NewBlock\}$;
  - $\mathbf{BMap} \oplus \{int \mapsto NewBlock\}$;
  - $\mathbf{PMap} \oplus \{(i + 1) \mapsto (i + 1)\}$;
  - $\mathbf{PMap} \oplus \{sPC \mapsto sPC\}$;
  - $\mathbf{PMap} \oplus \{ePC \mapsto ePC\}$;
  - $\mathbf{PMap} \oplus \{hPC \mapsto hPC\}$;
  - $\mathbf{PMap} \oplus \{int \mapsto int\}$;
- $P[i]$ is any other instruction : no changes to $\mathbf{BMap}$ and $\mathbf{PMap}$.

### B. Resolve Parents Successors Relationship (**TraceParentChild**)

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'll abstract away from the detail and define the function as getting a fresh label which will not conflict with any existing label and labels for additional blocks before handler. These additional blocks before handlers 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, it means that adding this additional block as successor a of the mentioned block, 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. Case by case translation behaviour :

- $P[i]$ 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.
  - $\circ$ **BMap(PMap($i$)).succs** $\cup \{t\}$;
  - $\circ$ **BMap(PMap($i$)).pSucc** $= t$; and
  - $\circ$ **BMap($t$).parents** $\cup \{$**PMap($i$)**$\}$

- $P[i]$ is Conditional jump (**ifeq** $t$) : since there will be 2 successors from this instruction, the current block will have additional 2 successors block and both of the blocks will also update their parents list to include the current block, i.e.
  - $\circ$ **BMap(PMap($i$)).succs** $\cup \{i + 1, t\}$;
  - $\circ$ **BMap(PMap($i$)).pSucc** $= i + 1$;
  - $\circ$ **BMap($i + 1$).parents** $\cup \{$**PMap($i$)**$\}$; and
  - $\circ$ **BMap($t$).parents** $\cup \{$**PMap($i$)**$\}$

- $P[i]$ is **Return** : just add the return block as the current block successors, and also update the parent of return block to include the current block, i.e.
  - $\circ$ **BMap(PMap($i$)).succs** $\cup \{ret\}$;
  - $\circ$ **BMap(PMap($i$)).pSucc** $= ret$; and
  - $\circ$ **BMap($ret$).parents** $\cup \{$**PMap($i$)**$\}$

- $P[i]$ is one of the object manipulation instruction. 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 simplification here where we add the next instruction as the block's successor directly, i.e.
  - $\circ$ **BMap(PMap($i$)).succs** $\cup \{i + 1\}$;
  - $\circ$ **BMap(PMap($i$)).pSucc** $= i + 1$;
  - $\circ$ **BMap($i + 1$).parents** $\cup \{$**PMap($i$)**$\}$; and
  - $\circ$ for each exception handler $j$ associated with $i$, let $intPC = maxLabel + j.$**handlerPC** and $hPC = j.$**handlerPC**:
    - $\blacksquare$ **BMap(PMap($i$)).succs** $\cup \{intPC\}$;
    - $\blacksquare$ **BMap(PMap($i$)).handlers** $\cup \{j\}$;
    - $\blacksquare$ **BMap($intPC$).parents** $\cup \{$**PMap($i$)**$\}$
    - $\blacksquare$ **BMap($intPC$).succs** $\cup \{hPC\}$
    - $\blacksquare$ **BMap($intPC$).insn** $= \{$**moveexception**$\}$
    - $\blacksquare$ **BMap($hPC$).parents** $\cup \{intPC\}$

  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 anyway during translation

- $P[i]$ is method invocation instruction. The treatment here is similar to that of object manipulation, where the next instruction is primary successor, and the exception handler for this instruction are 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 it's primary successor, and the next instruction ($i+1$) will have $l$ added to its list of parents, i.e.
  - $\circ$ $l = $ **getAvailableLabel**;
  - $\circ$ **BMap(PMap($i$)).succs** $\cup \{l\}$;
  - $\circ$ **BMap(PMap($i$)).pSucc** $= l$;
  - $\circ$ **BMap(PMap($i$)).parents** $= \{i\}$;
  - $\circ$ **BMap** $\oplus \{l \mapsto NewBlock\}$;
  - $\circ$ **BMap($l$).succs** $= \{i + 1\}$;
  - $\circ$ **BMap($l$).pSucc** $= (i + 1)$;
  - $\circ$ **BMap($l$).insn** $= \{$**moveresult**$\}$
  - $\circ$ **BMap($i + 1$).parents** $\cup \{l\}$; and
  - $\circ$ for each exception handler $j$ associated with $i$, let $intPC = maxLabel + j.$**handlerPC** and $hPC = j.$**handlerPC**:
    - $\blacksquare$ **BMap(PMap($i$)).succs** $\cup \{intPC\}$;
    - $\blacksquare$ **BMap(PMap($i$)).handlers** $\cup \{j\}$;
    - $\blacksquare$ **BMap($intPC$).parents** $\cup \{$**PMap($i$)**$\}$
    - $\blacksquare$ **BMap($intPC$).succs** $\cup \{hPC\}$
    - $\blacksquare$ **BMap($intPC$).insn** $= \{$**moveexception**$\}$
    - $\blacksquare$ **BMap($hPC$).parents** $\cup \{intPC\}$

- $P[i]$ is throw instruction. This instruction only add 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**:
  - $\circ$ **BMap(PMap($i$)).succs** $\cup \{intPC\}$;
  - $\circ$ **BMap(PMap($i$)).handlers** $\cup \{j\}$;
  - $\circ$ **BMap($intPC$).parents** $\cup \{$**PMap($i$)**$\}$
  - $\circ$ **BMap($intPC$).succs** $\cup \{hPC\}$
  - $\circ$ **BMap($intPC$).insn** $= \{$**moveexception**$\}$
  - $\circ$ **BMap($hPC$).parents** $\cup \{intPC\}$

- $P[i]$ is any other instruction : depending whether the next instruction is a start of a block or not.
  - $\circ$ 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.
    - $\blacksquare$ **BMap(PMap($i$)).succs** $\cup \{i + 1\}$; and
    - $\blacksquare$ **BMap($i + 1$).parents** $\cup \{$**PMap($i$)**$\}$
  - $\circ$ If the next instruction is not start of a block, then just point the next instruction to have the same pointer as the current block, i.e. **PMap($i + 1$)** $= $ **PMap($i$)**

## C. Reading Java Bytecodes (**Translate**)

Table I list the resulting DEX translation for each of the JVM bytecode instruction listed in section III. The full translation scheme with their typing rules can be seen in

| | | Translation | Side effect |
|---|---|---|---|
| $[\![\text{push}]\!]$ | = | $\mathbf{const}(r(\mathbf{TS}_i), n)$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) + 1$ |
| $[\![\text{pop}]\!]$ | = | $\varnothing$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) - 1$ |
| $[\![\text{load } x]\!]$ | = | $\mathbf{move}(r(\mathbf{TS}_i), r_x)$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) + 1$ |
| $[\![\text{store } x]\!]$ | = | $\mathbf{move}(r_x, r(\mathbf{TS}_i - 1))$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) - 1$ |
| $[\![\text{binop } op]\!]$ | = | $\mathbf{binop}(op, r(\mathbf{TS}_i - 2), r(\mathbf{TS}_i - 2), r(\mathbf{TS}_i - 1))$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) - 1$ |
| $[\![\text{swap}]\!]$ | = | $\mathbf{move}(r(\mathbf{TS}_i), r(\mathbf{TS}_i - 2))$ $\mathbf{move}(r(\mathbf{TS}_i + 1), r(\mathbf{TS}_i - 2))$ $\mathbf{move}(r(\mathbf{TS}_i - 1), r(\mathbf{TS}_i + 1))$ $\mathbf{move}(r(\mathbf{TS}_i - 2), r(\mathbf{TS}_i))$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i)$ |
| $[\![\text{goto } t]\!]$ | = | $\varnothing$ | $\mathbf{TS}(t) = \mathbf{TS}(i)$ |
| $[\![\text{ifeq } t]\!]$ | = | $\mathbf{ifeq}(r(\mathbf{TS}_i - 1), t)$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) - 1$ $\mathbf{TS}(t) = \mathbf{TS}(i) - 1$ |
| $[\![\text{return}]\!]$ | = | $\mathbf{move}(r_0, r(\mathbf{TS}_i - 1))$ $\mathbf{return}(r_0)$ or $\mathbf{goto}(ret)$ | |
| $[\![\text{new } C]\!]$ | = | $\mathbf{new}(r(\mathbf{TS}_i - 1), C)$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) + 1$ |
| $[\![\text{getfield } f]\!]$ | = | $\mathbf{iget}(r(\mathbf{TS}_i - 1), r(\mathbf{TS}_i - 1), f)$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) + 1$ |
| $[\![\text{putfield } f]\!]$ | = | $\mathbf{iput}(r(\mathbf{TS}_i - 1), r(\mathbf{TS}_i - 2), f)$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) - 2$ |
| $[\![\text{newarray } t]\!]$ | = | $\mathbf{newarray}(r(\mathbf{TS}_i - 1), r(\mathbf{TS}_i - 1), t)$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i)$ |
| $[\![\text{arraylength}]\!]$ | = | $\mathbf{arraylength}(r(\mathbf{TS}_i - 1), r(\mathbf{TS}_i - 1))$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i)$ |
| $[\![\text{arrayload}]\!]$ | = | $\mathbf{aget}(r(\mathbf{TS}_i - 2), r(\mathbf{TS}_i - 2), r(\mathbf{TS}_i - 1))$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) - 1$ |
| $[\![\text{arraystore}]\!]$ | = | $\mathbf{aput}(r(\mathbf{TS}_i - 1), r(\mathbf{TS}_i - 3), r(\mathbf{TS}_i - 2))$ | $\mathbf{TS}(i+1) = \mathbf{TS}(i) - 3$ |
| $[\![\text{invoke } m]\!]$ | = | $\mathbf{invoke}(n, m, \vec{p})$ $\mathbf{moveresult}(r(\mathbf{TS}_i - n))$ at block $l$ | $l = \mathbf{getAvailableLabel}$ $\mathbf{TS}(i+1) = \mathbf{TS}(i) - n$ |
| $[\![\text{throw}]\!]$ | = | $\mathbf{throw}(r(\mathbf{TS}_i - 1))$ | |

TABLE I: Instruction Translation Table

table II in the appendix. 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 in verified JVM bytecodes, we can safely assume that these top of the stacks will be consistent (even though an instruction may have a lot of 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 $r_x$.

### D. Ordering Blocks (**PickOrder**)

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. Starting from the initial block, we pick the first unordered successor and then keep on tracing until there is no more successor.

After we reached one end, we pick an unordered block and do the "trace analysis" again. But 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 1 describes how we implement this "trace analysis".

---

**Algorithm 1** PickOrder($blocks$)

---

$order := 0;$
**while** there is still block $x \in blocks$ without order; **do**
  $var := PickStartingPoint(x, \{x\});$
  $order = TraceSuccessors(source, order);$
**return** order;

---

- **Pick Starting Point**
  This function is a recursive function with an auxiliary data structure to prevent ancestor loop from 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 2** PickStartingPoint($x, loop$)

---

**for all** $p \in BMap(x).parents$ **do**
  **if** $p \in loop$ **then return** $x$;
  bp = BMap(p);
  **if** $bp.pSucc = x$ **and** $bp.order = -1$ **then**
    $loop = loop \cup \{p\};$
    **return** PickStartingPoint(p, loop)
**return** order;

---

- **Trace Successors**
  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 3** TraceSuccessors($x, order$)

---

$BMap(x).order = order;$
**if** $BMap(x).psucc \neq -1$ **then**
  pSucc = BMap(x).pSucc;
  **if** $BMap(pSucc).order = -1$ **then return** $TraceSuccessors(pSucc, order + 1);$
  **for all** $s \in BMap(x).succs$ **do**
    **if** $BMap(pSucc).order = -1$ **then return** $TraceSuccessors(s, order + 1);$
**return** order;

---

Since the translation phase already translated the JVM instruction and ordered the block, this phase basically just output the instructions in order of the block. Nevertheless, there are 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 of the 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 B.1.

- 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 store 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 4** makeHandlerEntry($cH, cS, cE$)

---

$\quad$ **for all** handler $h \in cH$ **do**
$\quad\quad hPC = h.handlerPC;$
$\quad\quad t = h.catchType;$
$\quad\quad DEXHandler \quad\quad = \quad\quad DEXHandler \quad +$
$\quad\quad newHandler(cS, cE, hPC, t);$

---

The only information that are needed to produce the information about exception handlers in DEX is the basic blocks contained in **BMap**. The procedure **translateExceptionHandlers** (Algorithm 5) 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.

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 details of the steps in this phase is contained in Algorithm 6.

---

**Algorithm 5** translateExceptionHandlers($blocks$)

---

$\quad cH = \varnothing;$ // current handler
$\quad cS$ // current start PC
$\quad cE$ // current start PC
$\quad$ **for all** block $x$ in order **do**
$\quad\quad$ **if** $x.handlers$ is not empty **then**
$\quad\quad\quad$ **if** $cH = x.handlers;$ **then**
$\quad\quad\quad\quad cE = x.endPC;$
$\quad\quad\quad$ **else if** $cH \neq x.handlers$ **then**
$\quad\quad\quad\quad makeHandlerEntry(cH, cS, cE);$
$\quad\quad\quad\quad cS = x.startPC;$
$\quad\quad\quad\quad cE = x.endPC;$
$\quad\quad\quad\quad cH = x.handlers;$
$\quad makeHandlerEntry(cH, cS, cE);$

---

**Algorithm 6** output

---

$\quad blocks =$ ordered blocks $\in BMap;$
$\quad lbl = \varnothing;$ // label mapping
$\quad out = \varnothing;$ // list of DEX output
$\quad pc = 0;$ // DEX program counter
$\quad$ **for all** block $x$ in order **do**
$\quad\quad next =$ next block in order;
$\quad\quad lbl[x] = pc;$
$\quad\quad pc = pc + x.insn.length;$
$\quad\quad out = out + x.insn;$
$\quad\quad$ **if** $p.pSucc \neq next$ **then**
$\quad\quad\quad$ **if** $x.insn.last$ is $ifeq$ **then**
$\quad\quad\quad\quad t = x.insn.last.target;$
$\quad\quad\quad\quad$ **if** $t = next$ **then**
$\quad\quad\quad\quad\quad out.last = oppositeCondition(x.insn.last);$
$\quad\quad\quad\quad$ **else**
$\quad\quad\quad\quad\quad out = out + goto(next);$
$\quad\quad\quad$ **else**
$\quad\quad\quad\quad out = out + goto(next);$
$\quad$ **for all** index $i$ in $out$ **do**
$\quad\quad$ **if** $out[i]$ is a jump instruction **then**
$\quad\quad\quad out[i].target = lbl[out[i].target];$
$\quad translateExceptionHandlers(blocks);$

---

The full translation scheme from JVM to DEX can be seen in table II.

| JVM | DEX | Original Transfer Rule | Related DEX Transfer Rule |
|---|---|---|---|
| Push | Const | $$\frac{P[i] = \mathbf{Push}\,v}{se, i \vdash^{\mathrm{Norm}} st \Rightarrow se(i) :: st}$$ | $$\frac{P[i] = \mathbf{Const}(r, n)}{se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(i)\}}$$ |
| Pop | None | $$\frac{P[i] = \mathbf{Pop}}{i \vdash st \Rightarrow st}$$ | None |
| Load | Move | $$\frac{P[i] = \mathbf{Load}\,x}{se, i \vdash^{\mathrm{Norm}} st \Rightarrow (se(i) \sqcup \vec{k_a}(x)) :: st}$$ | $$\frac{P[i] = \mathbf{Move}(r, r_s)}{se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto (se(i) \sqcup rt(r_s))\}}$$ |
| Store | Move | $$\frac{P[i] = \mathbf{Store}\,x \quad k \sqcup se(i) \le \vec{k_a}(x)}{\vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow st}$$ | $$\frac{P[i] = \mathbf{Move}(r, r_s)}{se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(i) \sqcup rt(r_s)\}}$$ |
| Binop | Binop | $$\frac{P[i] = \mathbf{Binop}}{se, i \vdash^{\mathrm{Norm}} a :: b :: st \Rightarrow (se(i) \sqcup a \sqcup b) :: st}$$ | $$\frac{P[i] = \mathbf{Binop}(r, r_a, r_b)}{se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto (se(i) \sqcup rt(r_a) \sqcup rt(r_b))\}}$$ |
| Swap | Move | $$\frac{P[i] = \mathbf{Swap}}{i \vdash^{\mathrm{Norm}} k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st}$$ | $$\frac{P[i] = \mathbf{Move}(r, r_s)}{se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto (se(i) \sqcup rt(r_s))\}}$$ |
| Goto | Goto | $$\frac{P[i] = \mathrm{Goto}\ t}{i \vdash st \Rightarrow st}$$ | $$\frac{P[i] = \mathrm{Goto}\ t}{i \vdash rt \Rightarrow rt}$$ <br><br> *) Not directly translated |
| Ifeq | Ifeq / Ifneq | $$\frac{P[i] = \mathbf{ifeq}\,t \quad \forall j' \in \mathbf{region}(i), k \le se(j')}{\mathbf{reigon}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow \mathrm{lift}_k(st)}$$ <br><br> Ifeq may be translated into Ifneq on certain condition | $$\frac{P[i] = \mathbf{ifeq}(r, t) \quad \forall j' \in \mathrm{region}(i), se(i) \sqcup rt(r) \le se(j')}{\mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt}$$ <br><br> $$\frac{P[i] = \mathbf{ifneq}(r, t) \quad \forall j' \in \mathrm{region}(i), se(i) \sqcup rt(r) \le se(j')}{\mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt}$$ |
| New | New | $$\frac{P[i] = \mathbf{new}\,C}{se, i \vdash^{\mathrm{Norm}} st \Rightarrow se(i) :: st}$$ | $$\frac{P[i] = \mathbf{new}(r, c)}{se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto se(i)\}}$$ |
| Getfield | Iget | $$\frac{P[i] = \mathbf{getfield}\,f \quad k \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \le se(j)}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow \mathrm{lift}_{\mathbf{k}}((k \sqcup \mathbf{ft}(f) \sqcup se(i)) :: st)}$$ <br><br> $$\frac{P[i] = \mathbf{getfield}\,f \quad k \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$$ <br><br> $$\frac{P[i] = \mathbf{getfield}\,f \quad k \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad k \le \vec{k_r}[\mathrm{np}]}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k :: st \Rightarrow}$$ | $$\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(r_o) \le se(j)}{\mathbf{ft}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto rt(r_o) \sqcup \mathbf{ft}(f) \sqcup se(i)\}}$$ <br><br> $$\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{ft}, se, i \vdash^{\mathrm{np}} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto rt(r_o) \sqcup se(i)\}}$$ <br><br> $$\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad rt(r_o) \in \mathcal{S} \quad se(i) \sqcup rt(r_o) \le \vec{k_r}[\mathrm{np}] \quad \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash^{\mathrm{np}} rt \Rightarrow}$$ |
| Putfield | Iput | $$\frac{\begin{array}{c} P[i] = \mathbf{putfield}\,f \quad k_h \le \mathbf{ft}(f) \quad k_1 \sqcup se(i) \sqcup k_2 \le \mathbf{ft}(f) \\ k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k_2 \le se(j) \end{array}}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} k_1 :: k_2 :: st \Rightarrow \mathrm{lift}_{\mathbf{k_2}}(st)}$$ <br><br> $$\frac{\begin{array}{c} P[i] = \mathbf{putfield}\,f \quad k_1 \sqcup se(i) \sqcup k_2 \le \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) = t \\ k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k_2 \le se(j) \end{array}}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$$ <br><br> $$\frac{\begin{array}{c} P[i] = \mathbf{putfield}\,f \quad k_1 \sqcup se(i) \sqcup k_2 \le \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \\ k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k_2 \le se(j) \quad k_2 \le \vec{k_r}[\mathrm{np}] \end{array}}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: st \Rightarrow}$$ | $$\frac{\begin{array}{c} P[i] = \mathbf{iput}(r_s, r_o, f) \quad k_h \le \mathbf{ft}(f) \quad rt(r_o) \in \mathcal{S} \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \\ rt(r_o) \sqcup se(i) \sqcup rt(r_s) \le \mathbf{ft}(f) \\ \forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(r_o) \le se(j) \end{array}}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt}$$ <br><br> $$\frac{\begin{array}{c} P[i] = \mathbf{iput}(r_s, r_o, f) \quad k_h \le \mathbf{ft}(f) \quad rt(r_o) \in \mathcal{S} \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \\ rt(r_o) \sqcup se(i) \sqcup rt(r_s) \le \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) = t \\ \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \le se(j) \end{array}}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash^{\mathrm{np}} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto rt(r_o) \sqcup se(i)\}}$$ <br><br> $$\frac{\begin{array}{c} P[i] = \mathbf{iput}(r_s, r_o, f) \quad k_h \le \mathbf{ft}(f) \quad rt(r_o) \in \mathcal{S} \quad rt(r_s) \in \mathcal{S}^{\mathrm{ext}} \\ rt(r_o) \sqcup se(i) \sqcup rt(r_s) \le \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \\ \forall j \in \mathbf{region}(i, \mathrm{np}), rt(r_o) \le se(j) \quad se(i) \sqcup rt(r_o) \le \vec{k_r}[\mathrm{np}] \end{array}}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash^{\mathrm{np}} rt \Rightarrow}$$ |

| JVM | DEX | Original Typing Rule | Related DEX Typing Rule |
|---|---|---|---|
| Newarray | Newarray | $$\frac{P[i] = \textbf{newarray}\,t \quad k \in \mathcal{S}}{i \vdash^{\text{Norm}} k :: st \Rightarrow k[\textbf{at}(i)] :: st}$$ | $$\frac{P[i] = \textbf{newarray}(r, r_l, t) \quad rt(r_l) \in \mathcal{S}}{i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto rt(r_l)[\textbf{at}(i)]\}}$$ |
| Arraylength | Arraylength | $$\frac{\begin{array}{c}P[i] = \textbf{arraylength} \quad \forall j \in \textbf{region}(i, \text{Norm}), k \le se(j) \\ k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}}\end{array}}{\textbf{region}, se, i \vdash^{\text{Norm}} k[k_c] :: st \Rightarrow \textbf{lift}_{\mathbf{k}}(k :: st)}$$ $$\frac{\begin{array}{c}P[i] = \textbf{arraylength} \quad \forall j \in \textbf{region}(i, \text{np}), k \le se(j) \\ k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \textbf{Handler}(i, \text{np}) = t\end{array}}{\textbf{region}, se, i \vdash^{\text{np}} k[k_c] :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$$ $$\frac{\begin{array}{c}P[i] = \textbf{arraylength} \quad \forall j \in \textbf{region}(i, \text{np}), k \le se(j) \\ k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \textbf{Handler}(i, \text{np}) \uparrow \quad k \le \vec{k_r}[\text{np}]\end{array}}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k[k_c] :: st \Rightarrow}$$ | $$\frac{\begin{array}{c}P[i] = \textbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \\ k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{Norm}), k \le se(j)\end{array}}{\textbf{region}, se, i \vdash^{norm} rt \Rightarrow rt \oplus \{r \mapsto k\}}$$ $$\frac{\begin{array}{c}P[i] = \textbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \\ k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k \le se(j) \\ \textbf{Handler}(i, \text{np}) = t\end{array}}{\textbf{region}, se, i \vdash^{np} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}}$$ $$\frac{\begin{array}{c}P[i] = \textbf{arraylength}(r, r_a) \quad k[k_c] = rt(r_a) \quad k \in \mathcal{S} \\ k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \textbf{region}(i, \text{np}), k \le se(j) \\ \textbf{Handler}(i, \text{np}) \uparrow \quad se(i) \sqcup k \le \vec{k_a}[\text{np}]\end{array}}{\vec{k_a} \to \vec{k_r}, \textbf{region}, se, i \vdash^{np} rt \Rightarrow}$$ |
| Arrayload | Aget | $$\frac{\begin{array}{c}P[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \\ \forall j \in \textbf{region}(i, \text{Norm}) k_2 \le se(j)\end{array}}{\begin{array}{c}\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{Norm}} k_1 :: k_2[k_c] :: st \Rightarrow \\ \textbf{lift}_{\mathbf{k_2}}(((k_1 \sqcup k_2) \sqcup^{\text{ext}} k_c) :: st)\end{array}}$$ $$\frac{\begin{array}{c}P[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \\ \forall j \in \textbf{region}(i, \text{np}) k_2 \le se(j) \quad \textbf{Handler}(i, \text{np}) = t\end{array}}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2[k_c] :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$$ $$\frac{\begin{array}{c}P[i] = \textbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad k_2 \le \vec{k_r}[\text{np}] \\ \forall j \in \textbf{region}(i, \text{np}) k_2 \le se(j) \quad \textbf{Handler}(i, \text{np}) \uparrow\end{array}}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2[k_c] :: st \Rightarrow}$$ | $$\frac{\begin{array}{c}P[i] = \textbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k_c \in \mathcal{S}^{\text{ext}} \\ k, rt(r_i) \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \text{Norm}), k \le se(j)\end{array}}{\begin{array}{c}\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{norm} rt \Rightarrow \\ rt \oplus \{r \mapsto ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} k_c)\}\end{array}}$$ $$\frac{\begin{array}{c}P[i] = \textbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k_c \in \mathcal{S}^{\text{ext}} \\ k, rt(r_i) \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \text{np}), k \le se(j) \\ \textbf{Handler}(i, \text{np}) = t\end{array}}{\textbf{region}, se, i \vdash^{np} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}}$$ $$\frac{\begin{array}{c}P[i] = \textbf{aget}(r, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k_c \in \mathcal{S}^{\text{ext}} \\ k, rt(r_i) \in \mathcal{S} \quad \forall j \in \textbf{region}(i, \text{np}), k \le se(j) \\ \textbf{Handler}(i, \text{np}) \uparrow \quad se(i) \sqcup k \le \vec{k_r}[\text{np}]\end{array}}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{np} rt \Rightarrow}$$ |
| Arraystore | Aput | $$\frac{\begin{array}{c}P[i] = \textbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \quad k_2, k_3 \in \mathcal{S} \\ ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \le^{\text{ext}} k_c \quad \forall j \in \textbf{region}(i, \text{Norm}), k_2 \le se(j)\end{array}}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{Norm}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow \textbf{lift}_{\mathbf{k_2}}(st)}$$ $$\frac{\begin{array}{c}P[i] = \textbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \\ ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \le^{\text{ext}} k_c \quad k_2, k_3 \in \mathcal{S} \\ \textbf{Handler}(i, \text{np}) = t \quad \forall j \in \textbf{region}(i, \text{np}), k_2 \le se(j)\end{array}}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$$ $$\frac{\begin{array}{c}P[i] = \textbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\text{ext}} \quad k_2, k_3 \in \mathcal{S} \\ ((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \le^{\text{ext}} k_c \quad \forall j \in \textbf{region}(i, \text{np}), k_2 \le se(j) \\ \textbf{Handler}(i, \text{np}) \uparrow \quad k_2 \le \vec{k_r}[\text{np}]\end{array}}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{np}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow}$$ | $$\frac{\begin{array}{c}P[i] = \textbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \\ ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \le^{\text{ext}} k_c \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \\ \forall j \in \textbf{region}(i, \text{Norm}), k \le se(i)\end{array}}{\vec{k_a} \to k_r, \textbf{region}, se, i \vdash^{\text{Norm}} rt \Rightarrow rt}$$ $$\frac{\begin{array}{c}P[i] = \textbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \\ ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \le^{\text{ext}} k_c \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \\ \forall j \in \textbf{region}(i, \text{np}), k \le se(i) \quad \textbf{Handler}(i, \text{np}) = t\end{array}}{\textbf{region}, se, i \vdash^{np} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}}$$ $$\frac{\begin{array}{c}P[i] = \textbf{aput}(r_s, r_a, r_i) \quad k[k_c] = rt(r_a) \quad k, rt(r_i) \in \mathcal{S} \\ ((k \sqcup rt(r_i)) \sqcup^{\text{ext}} rt(r_s)) \le^{\text{ext}} k_c \quad k_c, rt(r_s) \in \mathcal{S}^{\text{ext}} \\ \forall j \in \textbf{region}(i, \text{np}), k \le se(i) \quad \textbf{Handler}(i, \text{np}) = t \\ se(i) \sqcup k \le \vec{k_r}[\text{np}]\end{array}}{\vec{k_a} \to \vec{k_r} \textbf{region}, se, i \vdash^{\text{np}} rt \Rightarrow}$$ |
| Return | Move **and** Goto **or** Return | $$\frac{P[i] = \textbf{return} \quad se(i) \sqcup k \le k_r}{\vec{k_a} \xrightarrow{k_h} \vec{k_r}, se, i \vdash k :: st \Rightarrow}$$ | $$\frac{P[i] = \textbf{Move}(r_0, r_s)}{se, i \vdash rt \Rightarrow rt \oplus \{r \mapsto (se(i) \sqcup rt(r_s))\}}$$ $$\frac{P[i] = \textbf{goto}(t)}{i \vdash rt \Rightarrow rt}$$ $$\frac{P[i] = \textbf{return}(r_s) \quad se(i) \sqcup rt(r_s) \le k_r}{\vec{k_a} \xrightarrow{k_h} \vec{k_r}, se, i \vdash rt \Rightarrow}$$ |

| JVM | DEX | Original Typing Rule | Related DEX Typing Rule |
|---|---|---|---|
| Invoke | Invoke | $$\frac{\begin{array}{c} P_m[i] = \textbf{invoke}\,m_{\text{ID}} \quad \textbf{length}(st_1) = \text{nbArguments}(m_I D) \\ k \leq \vec{k_a'}[0] \quad \forall i \in [0, \textbf{length}(st_1) - 1]\, st_1[i] \leq \vec{k_a'}[i+1] \\ k \sqcup k_h \sqcup se(i) \leq k_h' \quad k_e = \bigsqcup\{\vec{k_r'}[e] \mid e \in \textbf{excAnalysis}(m_{\text{ID}})\} \\ \Gamma_{m_{ID}}[k] = \vec{k_a} \overset{k_h'}{\to} k_r' \quad \forall j \in \textbf{region}(i, \text{Norm}), k \sqcup k_e \leq se(j) \end{array}}{\begin{array}{c} \Gamma, \textbf{region}, se, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, i \vdash^{\text{Norm}} st_1 :: k :: st_2 \Rightarrow \\ \textbf{lift}_{\mathbf{k} \sqcup \mathbf{k_e}}((k_r' \sqcup k \sqcup se(i)) :: st_2) \end{array}}$$ | $$\frac{\begin{array}{c} P_m[i] = \textbf{invoke}\,(n, m, \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k_a} \overset{k_h'}{\to} k_r' \\ rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k_r' \quad \forall_{0 \leq j < n}\, rt(\vec{p}[j]) \leq \vec{k_a}[j] \\ k_e = \bigsqcup\{\vec{k_r'}[e] \mid e \in \textbf{excAnalysis}(m')\} \\ \forall j \in \textbf{region}(i, \text{Norm}), rt(\vec{p}[0]) \sqcup k_e \leq se(j) \end{array}}{\begin{array}{c} \Gamma, \textbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash^{\text{Norm}} rt \Rightarrow \\ rt \oplus \{ret \mapsto k_r'[n] \sqcup rt(\vec{p}[0]) \sqcup se(i)\} \end{array}}$$ |
| | | $$\frac{\begin{array}{c} P_m[i] = \textbf{invoke}\,m_{\text{ID}} \quad \textbf{length}(st_1) = \text{nbArguments}(m_I D) \\ k \leq \vec{k_a'}[0] \quad \forall i \in [0, \textbf{length}(st_1) - 1]\, st_1[i] \leq \vec{k_a'}[i+1] \\ k \sqcup k_h \sqcup se(i) \leq k_h' \quad e \in \textbf{excAnalysis}(m_{\text{ID}}) \cup \{\text{np}\} \\ \Gamma_{m_{ID}}[k] = \vec{k_a} \overset{k_h'}{\to} k_r' \quad \forall j \in \textbf{region}(i, e), k \sqcup k_e \leq se(j) \\ \textbf{Handler}(i, e) = t \end{array}}{\Gamma, \textbf{region}, se, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, i \vdash^e st_1 :: k :: st_2 \Rightarrow (k \sqcup \vec{k_r'}[e]) :: \epsilon}$$ | $$\frac{\begin{array}{c} P_m[i] = \textbf{invoke}\,(n, m, \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k_a} \overset{k_h'}{\to} k_r' \\ rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k_h' \quad \forall_{0 \leq j < n}\, rt(\vec{p}[j]) \leq \vec{k_a}[j] \\ e \in \textbf{excAnalysis}(m') \cup \{\text{np}\}\} \quad \textbf{Handler}(i, e) = t \\ \forall j \in \textbf{region}(i, e), rt(\vec{p}[0]) \sqcup \vec{k_r'}[e] \leq se(j) \end{array}}{\begin{array}{c} \Gamma, \textbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash^e rt \Rightarrow \\ \vec{k_a} \oplus \{ex \mapsto k_r'[e] \sqcup sec(\vec{p}[0])\} \end{array}}$$ |
| | | $$\frac{\begin{array}{c} P_m[i] = \textbf{invoke}\,m_{\text{ID}} \quad \textbf{length}(st_1) = \text{nbArguments}(m_I D) \\ k \leq \vec{k_a'}[0] \quad \forall i \in [0, \textbf{length}(st_1) - 1]\, st_1[i] \leq \vec{k_a'}[i+1] \\ k \sqcup k_h \sqcup se(i) \leq k_h' \quad e \in \textbf{excAnalysis}(m_{\text{ID}}) \cup \{\text{np}\} \\ \Gamma_{m_{ID}}[k] = \vec{k_a} \overset{k_h'}{\to} k_r' \quad \forall j \in \textbf{region}(i, e), k \sqcup k_e \leq se(j) \\ \textbf{Handler}(i, e) \uparrow \quad k \sqcup se(i) \sqcup \vec{k_r'}[e] \leq \vec{k_r}[e] \end{array}}{\Gamma, \textbf{region}, se, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, i \vdash^e st_1 :: k :: st_2 \Rightarrow}$$ | $$\frac{\begin{array}{c} P_m[i] = \textbf{invoke}\,(n, m, \vec{p}) \quad \Gamma_{m'}[sec(\vec{p}[0])] = \vec{k_a} \overset{k_h'}{\to} k_r' \\ rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k_h' \quad \forall_{0 \leq j < n}\, rt(\vec{p}[j]) \leq \vec{k_a}[j] \\ e \in \textbf{excAnalysis}(m') \cup \{\text{np}\}\} \quad \textbf{Handler}(i, e) \uparrow \\ \forall j \in \textbf{region}(i, e), rt(\vec{p}[0]) \sqcup k_r'[e] \leq se(j) \\ rt(\vec{p}[0]) \sqcup se(i) \sqcup \vec{k_r'}[e] \leq \vec{k_r}[e] \end{array}}{\Gamma, \textbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash^e rt \Rightarrow}$$ |
| | Moveresult | | $$\frac{P_m[i] = \textbf{moveresult}(r)}{i \vdash^{\text{Norm}} rt \Rightarrow rt \oplus \{r \mapsto rt(ret)\}}$$ |

TABLE II: Translation Table

## APPENDIX B
## PROOF OF LEMMAS

### A. Proofs that Translation Preserves SOAP Satisfiability

We first start this section with proofs of lemmas that are omitted in the paper due to space requirement.

**Lemma B.1.** *Let $P$ be a JVM program and $P[a] = Ins_a$ and $P[b] = Ins_b$ are 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*

$$\llbracket \lfloor a \rfloor[n-1] \rrbracket \mapsto^{\text{Norm}} \llbracket \lfloor b \rfloor[0] \rrbracket$$
$$or$$
$$\left( \begin{array}{c} \llbracket \lfloor a \rfloor[n-1] \rrbracket \mapsto^{\text{Norm}} (\llbracket \lfloor a \rfloor[n-1] \rrbracket + 1) \\ and \\ (\llbracket \lfloor a \rfloor[n-1] \rrbracket + 1) \mapsto^{\text{Norm}} \llbracket \lfloor b \rfloor[0] \rrbracket \end{array} \right)$$

*in* $\llbracket P \rrbracket$.

*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 whether what instruction is $Ins_a$ and where $a$ and $b$ are located. Either:

- $\lfloor b \rfloor$ are placed directly after $\lfloor a \rfloor$ and $\lfloor a \rfloor[n-1]$ is sequential instruction;

In this case, appealing to the definition of successor relation this trivially holds as the first case.

- $\lfloor Ins_a \rfloor$ ends in a sequential instruction and will have a **goto** instruction appended that points to $\llbracket \lfloor b \rfloor[0] \rrbracket$; Again appealing to the definition of successor relation this trivially holds as the second case, where $P_{\text{DEX}}[(\llbracket \lfloor a \rfloor[n-1] \rrbracket + 1)] = \textbf{goto}(\llbracket \lfloor b \rfloor[0] \rrbracket)$.

- $\lfloor Ins_a \rfloor[n-1]$ is a branching instruction and $b$ is one of its child, and $\lfloor b \rfloor$ is placed directly after $\lfloor a \rfloor$ 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.

- $\lfloor Ins_a \rfloor[n-1]$ is a branching instruction and $b$ is one of its child, nevertheless $\lfloor b \rfloor$ is not placed directly after $\lfloor a \rfloor$ nor is pointed to by the branching instruction; In this case, according to the **Output** phase, a **goto** instruction will be appended in $(\llbracket \lfloor a \rfloor[n-1] \rrbracket + 1)$ and thus we are in the second case. Use the definition of successor relation to conclude the proof.

∎

**Lemma B.2.** *Let $P$ be a JVM program and $P[a] = Ins_a$ and $P[b] = Ins_b$ are 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 $\lfloor a \rfloor$ that throws exception. If $a \mapsto^{\tau} b$, then $\lceil \lfloor a \rfloor [e] \rceil \mapsto^{\tau} \lceil \lfloor h \rfloor \rceil$ and $\lceil \lfloor h \rfloor \rceil \mapsto^{\mathrm{Norm}} \lceil \lfloor b \rfloor [0] \rceil$*

*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 B.3.** *Let $P$ be a JVM program and $P[a] =$ **invoke** and $P[b] = Ins_b$ are two of its instructions at program points $a$ and $b$. If $a \mapsto^{\mathrm{Norm}} b$, then $\lceil \lfloor a \rfloor [0] \rceil \mapsto^{\mathrm{Norm}} \lceil \lfloor a \rfloor [1] \rceil$ and $\lceil \lfloor a \rfloor [1] \rceil \mapsto^{\mathrm{Norm}} \lceil \lfloor b \rfloor [0] \rceil$*

*Proof:* This is trivial based on the unfolding definition of the compiler since the primary successor of $\lceil \lfloor a \rfloor [0] \rceil$ is $\lceil \lfloor a \rfloor [1] \rceil$, where $\lceil P \rceil [\lceil \lfloor a \rfloor [1] \rceil] =$ **moveresult**, and the primary successor of $\lceil \lfloor a \rfloor [1] \rceil$ is $\lceil \lfloor b \rfloor [0] \rceil$. The proof is then concluded by the definition of successor relation in DEX. ∎

**Lemma B.4.** *Let $P$ be a JVM program and $P[a] = Ins_a$ and $P[b] = Ins_b$ are 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 instruction that cause branching will never be translated into empty sequence). If $a \mapsto b$, then either*

$$\lceil \lfloor a \rfloor [n-1] \rceil \mapsto^{\mathrm{Norm}} \lceil \lfloor s \rfloor [0] \rceil$$
$$or$$
$$\left( \begin{array}{c} \lceil \lfloor a \rfloor [n-1] \rceil \mapsto^{\mathrm{Norm}} (\lceil \lfloor a \rfloor [n-1] \rceil + 1) \\ and \\ (\lceil \lfloor a \rfloor [n-1] \rceil + 1) \mapsto^{\mathrm{Norm}} \lceil \lfloor s \rfloor [0] \rceil \end{array} \right)$$

*Proof:* We use induction on the length of successor's chain. In the base case where the length is 0, we can use Lemma B.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. ∎

**Lemma** (VII.1). *SOAP properties are preserved in the translation from JVM to DEX.*

*Proof:* We do prove 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 instruction. Denote $i_b$ as the program point in the sequence and is a branching point. Let $\lfloor P[k] \rfloor$ be the translation of instruction $P[k]$ and $k_1$ is the address of

its first instruction ($\lfloor P[k] \rfloor [0]$). Using the first case in the Lemma B.1, Lemma B.2, Lemma B.3 and Lemma B.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 Definition VII.2. In the case that $k = \mathbf{jun}(i_b, \tau)$, we then will have $k_1 = \mathbf{jun}(i_b, \tau)$ using Definition VII.6.
Special cases for the second case of Lemma B.1, Lemma B.2 and Lemma B.3 in that they contain additional instruction in the lemma. We argue that the property still holds using Definition VII.3 Definition VII.4, and Definition VII.5. Suppose $k'$ is the program point that points to the extra instruction, then we have $k' \in \mathbf{region}(i_b, \tau)$ from the 3 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 $\lfloor P[j] \rfloor$. Denote $i_b$ as the program point in the sequence $\lfloor i \rfloor$ and is a branching point. Let $\lfloor P[k] \rfloor$ be the translation of instruction $P[k]$ and $k_1$ is the address of its first instruction ($\lfloor P[k] \rfloor [0]$). Using Definition VII.2, we obtain $j_n \in \mathbf{region}(i_b, \tau)$. Using the first case of Lemma B.1, Lemma B.2, Lemma B.3 and Lemma B.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 $k \in \mathbf{region}(i, \tau)$, that means using Definition VII.2 we will have $k_1 \in \mathbf{region}(i_n, \tau)$. Assume $k = \mathbf{jun}(i, \tau)$, we use Definition VII.6 and obtain that $k_1 = \mathbf{jun}(i_n, \tau)$. Either way, SOAP property is preserved for SOAP2. Similar argument as SOAP1 to establish the second case of Lemma B.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 $\lfloor i \rfloor$ that branch and $k_{11}$ and $k_{21}$ be the first instruction in $\lfloor P[k_1] \rfloor$ and $\lfloor P[k_2] \rfloor$ respectively. We proceed by using Definition VII.2 and the knowledge that the JVM bytecode satisfy 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 Definition VII.6 we have $k_1 = \mathbf{jun}(i_n, \tau')$. Using Definition VII.2, we know that $k_1 \in \mathbf{region}(i_n, \tau)$. If we then set $k_1$ to be such 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 proving SOAP5, with the addition of simple property where the size of a code and its translation is covariant in a sense that if an program $a$ has more codes than $b$, then $\lfloor a \rfloor$ also has more codes than $\lfloor b \rfloor$.

∎

To prove the typability preservation of the compilation processes, we define an intermediate type system closely resembles that of DEX, except that the addressing is using block addressing. The purpose of this intermediate addressing is to know the existence of registers typing to satisfy typability and the constraint satisfaction for each instructions. We omit the details to avoid more clutters.

Following monotony lemma is useful in proving the relation of $\sqsubseteq$ between registers typing obtained from compiling stack types.

**Lemma B.5** (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. ∎

**Lemma** (VII.3). *For any JVM program $P$ with instruction $Ins$ at address $i$ and tag Norm, let the length of $\|Ins\|$ denoted by $n$. Let $RT_{\|i\|[0]} = \|S_i\|$. If according to the transfer rule for $P[i] = ins$ exists $st$ s.t. $i \vdash^{\mathrm{Norm}} S_i \Rightarrow st$ then*

$$\left( \begin{array}{c} \forall 0 \leq j < (n-1).\exists rt'.\|i\|[j] \vdash^{\mathrm{Norm}} RT_{\|i\|[j]} \Rightarrow rt', \\ rt' \sqsubseteq RT_{\|i\|[j+1]} \\ \textit{and} \\ \exists rt.\|i\|[n-1] \vdash^{\mathrm{Norm}} RT_{\|i\|[n-1]} \Rightarrow rt, rt \sqsubseteq \|st\| \end{array} \right)$$

*according to the transfer rule(s) of $\|Ins\|$*

*Proof:* It is case by case instruction, although for most of the instructions they are straigthforward as they only translate into one instruction. For the rest of the proof, using definition VII.7 to say that the translated $se(\|i\|)$ have the same security level as $se(i)$.

- **Push**
  We appeal directly to both of the transfer rule of **Push** and **Const**. In **Push** case, it only appends top of the stack with $se(i)$. Let such $rt$ be

  $$rt = RT_{\|i\|[0]} \oplus \{r(TS_i) \mapsto se(\|i\|[0])\}$$

  referring to **Const** transfer rule. Since **Push** is translated into $\mathbf{Const}(r(TS_i))$, where $TS_i$ corresponds to the top of the stack, we know that $\|st\| = rt$ because $RT_{\|i\|[0]} = \|S_i\|$ and the $rt$ we have is the same as $\|se(i) :: S_i\|$ thus $rt \sqsubseteq \|st\|$

- **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 $se(\|i\|[0]) \sqcup \vec{k_a}(x)$. And although there are several transfer rules for **move**, there is only one applicable because the source register comes from local variable register, and the target register is one of the stack space. Using this transfer rule, we can trivially show that $rt = \|st\|$ where $st = (se(i) \sqcup \vec{k_a}(x)) :: S_i$ and $rt = RT_{\|i\|[0]} \oplus \{r(TS_i) \mapsto se(\|i\|[0]) \sqcup \vec{k_a}(x)\}$, thus $rt \sqsubseteq \|st\|$.

- **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 register. The $rt$ in this case will be

  $$rt = RT_{\|i\|[0]} \oplus \{ r_x \mapsto se(\|i\|[0]) \sqcup \\ RT_{\|i\|[0]}(r(TS_i - 1))\}$$

  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 things happened 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

  $$rt = RT_{\|i\|[0]} \oplus \{r(TS_i - 2) \mapsto se(\|i\|[0]) \sqcup \\ RT_{\|i\|[0]}(r(TS_i - 1)) \sqcup (RT_{\|i\|[0]}(r(TS_i - 2)))\}$$

  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 $S_i = k_a :: k_b :: st'$ by Lemma VII.2

- **Swap**
  In dx tool, this instruction is translated into 4 **move** instructions. In this case, such $rt$ is

  $$rt = RT_{\|i\|[0]} \oplus \{ \\ r(TS_i) \mapsto se(\|i\|[0]) \sqcup RT_{\|i\|[0]}(r(TS_i - 2)), \\ r(TS_i + 1) \mapsto se(\|i\|[1]) \sqcup RT_{\|i\|[1]}(r(TS_i - 1)), \\ r(TS_i - 2) \mapsto se(\|i\|[2]) \sqcup RT_{\|i\|[2]}(r(TS_i - 1)), \\ r(TS_i - 1) \mapsto se(\|i\|[3]) \sqcup RT_{\|i\|[3]}(r(TS_i - 2))\}$$

  justified by applying transfer rule for **move** 4 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 $rt$ whereas there is no such occurrence in $st$. But on a closer look, 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 are 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 are 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 these registers typing

$$RT_{\|i\|[1]} = RT_{\|i\|[0]} \oplus \{r(TS_i) \mapsto se(\|i\|[0]) \sqcup RT_{\|i\|[0]}(r(TS_i - 2))\}$$
$$RT_{\|i\|[2]} = RT_{\|i\|[1]} \oplus \{r(TS_i + 1) \mapsto se(\|i\|[1]) \sqcup RT_{\|i\|[1]}(r(TS_i - 1))\}$$
$$RT_{\|i\|[3]} = RT_{\|i\|[2]} \oplus \{r(TS_i - 2) \mapsto se(\|i\|[2]) \sqcup RT_{\|i\|[3]}(r(TS_i - 1))\}$$

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 $\|S_i\| \oplus \{r(TS_i) \mapsto se(\|i\|[0])\}$.

- **Getfield**
  In this case, the transfer rule for the translated instruction coincides with the transfer rule for **Getfield**. Let

  $$rt = RT_{\|i\|[0]} \oplus \{r(TS_i - 1) \mapsto se(\|i\|[0]) \sqcup \mathbf{ft}(f)\}$$

  Then we have $rt = \|st\|$ which can be trivially shown with $st = se(i) \sqcup \mathbf{ft}(f) :: S_i$ thus giving us $rt \sqsubseteq \|st\|$.

- **Putfield**
  Since the JVM transfer rule for the operation itself only removes the top 2 stack, and the transfer rule for DEX keep the registers typing, when we have $rt = \|S_i\|$, then by the definition of $\sqsubseteq$ we'll have $rt \sqsubseteq \|st\|$ since $S_i = k_o :: k_v :: st$. As before, the registers that is not contained in the stack will by definition satisfy the $\sqsubseteq$ by Lemma VII.2.

- **Newarray**
  Similar to the argument of **load**, we have

  $$rt = RT_{\|i\|[0]} \quad \oplus \{r(TS_i - 1) \mapsto RT_{\|i\|[0]}(r(TS_i - 1))[\mathbf{at}(\|i\|[0])]\}$$

  , $rt = \|st\|$, where $st = k[at(i)] :: st', S_i = k :: st'$, which will give us $rt \sqsubseteq \|st\|$.

- **Arraylength**
  Let $k[k_c] = RT_{\|i\|[0]}(r(TS_i - 1)) = S_i[0]$. 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 $k[k_c] = RT_{\|i\|[0]}(r(TS_i - 2)) = S_i[1]$. In this case

  $$rt = RT_{\|i\|[0]} \oplus \{r(TS_i - 2) \mapsto (se(i) \sqcup k \sqcup RT_{\|i\|[0]}(r(TS_i - 1))) \sqcup^{\text{ext}} k_c\}$$

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 with **putfield** where the JVM instruction remove top of the stack and DEX instruction preserves the registers typing for $rt$. Thus appealing to the definition of $\sqsubseteq$ we have that $rt \sqsubseteq \|st\|$.

- **Invoke**
  This instruction itself yield 1 or 2 instructions depending 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 $\vec{k_a'} \overset{k_h'}{\to} \vec{k_r'}$ be the policy for method invoked. Type system wise, there will be 3 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 2 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 $t = locN + (\mathbf{length}(S_i) - \mathbf{length}(st_1) - 1)$, where $locN$ is the number of local variables. Let $k$ be the security level of object referenced and $k_e = \bigsqcup \{\vec{k_r'}[e] \mid e \in \mathbf{excAnalysis}(m_{\text{ID}})$. Since the method can also throw an exception, we have to also include the lub of security level for possible exceptions, denoted by $k_e$. In this case, such $rt$ can be

  $$RT_{\|i\|[0]} \oplus \{ \quad ret \mapsto (\vec{k_r'}[n] \sqcup se(\|i\|[0])), \\ r_t \mapsto (\vec{k_r'}[n] \sqcup se(\|i\|[1]))\}$$

  and by definition of $\sqsubseteq$ we will have that $rt \sqsubseteq \|st\|$, where $st = \mathbf{lift}_{k \sqcup k_e}((\vec{k_r'}[n] \sqcup se(i)) :: st_2)$ and $S_i = st_1 :: k :: st2$. With that form of $rt$ in mind, then the registers typing for $\|i\|[1]$ can be

  $$RT_{\|i\|[0]} \oplus \{ret \mapsto (\vec{k_r'}[n] \sqcup se(\|i\|[0]))$$

  coming from the 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** (VII.4). *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 $\|Ins\|$ until the instruction that throw exception $\tau$ denoted by $n$. Let $(be, 0) = \|i_e\|$ 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*

$$\left( \begin{array}{c} \forall 0 \le j < (n-1).\exists rt'.\lfloor\!\lfloor i \rfloor\!\rfloor[j] \vdash^{\text{Norm}} RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[j]} \Rightarrow rt', \\ rt' \sqsubseteq RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[j+1]} \end{array} \right)$$
$$and$$
$$\exists rt.\lfloor\!\lfloor i \rfloor\!\rfloor[n-1] \vdash^\tau RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[n-1]} \Rightarrow rt, rt \sqsubseteq RT_{(be,0)}$$
$$and$$
$$\exists rt.(be,0) \vdash^{\text{Norm}} RT_{(be,0)} \Rightarrow rt, rt \sqsubseteq \lfloor\!\lfloor st \rfloor\!\rfloor$$

*according to the transfer rule(s) of first $n$ instruction in $\lfloor\!\lfloor Ins \rfloor\!\rfloor$ and* **moveexception**.

*Proof:* Case by case possibly throwing instructions:

- **Invoke**
  We only need to take care of the case where the exception is caught, as uncaught exception is a return point therefore 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 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 $rt \sqsubseteq \lfloor\!\lfloor st \rfloor\!\rfloor$, where $st = (k \sqcup \vec{k'_r}[e]) :: \epsilon$. Such $rt$ 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 registers 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 uncaught exception, there is nothing to prove here as there is no resulting $st$. For caught exception, let $k$ be the security level of the exception and $locN$ be the number of local variable. Such $rt$ can be

  $$rt = \{\vec{k_a}, ex \mapsto \big(k \sqcup se(\lfloor\!\lfloor i \rfloor\!\rfloor[0])\big), \\ r(locN) \mapsto \big(k \sqcup se(\lfloor\!\lfloor i \rfloor\!\rfloor[0])\big)\}$$

  and it will make the relation $rt \sqsubseteq \lfloor\!\lfloor st \rfloor\!\rfloor$ 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

  $$RT_{(be,0)}\{\vec{k_a}, ex \mapsto (k \sqcup se(\lfloor\!\lfloor i \rfloor\!\rfloor[0]))\}$$

  which will give us the final $rt$ that 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** (VII.5)**.** *Let $ins$ be instruction at address $i$, $i \mapsto j$, $st$, $S_i$ and $S_j$ be stack types such that $i \vdash S_i \Rightarrow st, st \sqsubseteq S_j$. Let $n$ be the length of $\lfloor\!\lfloor ins \rfloor\!\rfloor$. Let $RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]} = \lfloor\!\lfloor S_i \rfloor\!\rfloor$, $RT_{\lfloor\!\lfloor j \rfloor\!\rfloor[0]} = \lfloor\!\lfloor S_j \rfloor\!\rfloor$ and $rt$ is obtained from the transfer rules involved in $\lfloor\!\lfloor ins \rfloor\!\rfloor$. Then $rt \sqsubseteq RT_{\lfloor\!\lfloor j \rfloor\!\rfloor[0]}$.*

*Proof:* Using Lemma VII.3 and Lemma VII.4 to establish that we have $rt \sqsubseteq \lfloor\!\lfloor st \rfloor\!\rfloor$. Then we conclude by using Lemma B.5 to establish that $rt \sqsubseteq RT_{\lfloor\!\lfloor j \rfloor\!\rfloor[0]}$ because $st \sqsubseteq S_j$. ∎

**Lemma** (VII.6)**.** *Let $Ins$ be instruction at program point $i$, $S_i$ its corresponding stack types, and let $RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]} = \lfloor\!\lfloor S_i \rfloor\!\rfloor$. If $P[i]$ satisfy the typing constraint for $Ins$ with the stack type $S_i$, then $\forall (bj, j) \in \lfloor\!\lfloor i \rfloor\!\rfloor.P_{DEX}[bj, j]$ will also satisfy the typing constraints for all instructions in $\lfloor\!\lfloor Ins \rfloor\!\rfloor$ with the initial registers typing $RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]}$.*

*Proof:* We do this by case by case instruction:

- **Push**
  This instruction is translated into **Const** which does not have any constraints.

- **Pop**: does not get translated.

- **Load** $x$
  This instruction is translated into **Move** which does not have any constraints.

- **Store** $x$
  This instruction is translated into **Move** which does not have any constraints.

- **Goto**: does not get translated

- **Ifeq** $t$
  This instruction will get translated to **ifeq** instruction where the condition is based on top of the stack $(TS_i - 1)$. There is only one constraint of the form $\forall j' \in \textbf{region}(i, \text{Norm}), rt\big(r(TS_i - 1)\big) \le se(j')$, and we know that in the JVM bytecode the constraint $\forall j' \in \textbf{region}(i, \text{Norm}), k \le se(j')$ is fulfilled. Based on the definition of $\lfloor\!\lfloor . \rfloor\!\rfloor$, we will have $k = rt\big(r(TS_i - 1)\big)$. Thus we only need to prove that the difference in region will still preserve the constraint satisfaction. We do this by proof by contradiction. Suppose there exists such instruction at address $(bj, j) \in \textbf{region}(\lfloor\!\lfloor i \rfloor\!\rfloor[n])$ such that $k \not\le se(bj, j)$. But according to definition VII.2, such instruction will come from an instruction at address $i'$ s.t. $i' \in \textbf{region}(i)$ thus it will satisfy $k \le se(i')$. By definition VII.7, $se(bj, j) = se(i')$, thus we will have $k \le se(bj, j)$. A plain contradiction.

- **Binop**
  This instruction is translated into **Binop** or **BinopConst** both of which does not have any constraints.

- **Swap**
  Trivially holds as well because all the 4 **move** in-

structions translated from **swap** do not have any constraints.

- **New**
  Trivially holds as the **New** does not have any constraints.

- **Getfield**
  There are different sets of constraints depending on whether the instruction executes normally, throw a caught exception, or throw an uncaught exception.
  In the case of **Getfield** executing normally, there are only two constraints that we need to take care, one is that $rt(r_o) \in \mathcal{S}$ and $\forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(r_o) \leq se(j)$. 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 similar argument to the satisfaction of region constraint in **Ifeq**.
  In the case of **Getfield** is 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 $rt(r_o) \leq \vec{k_r}[\mathrm{np}]$. This constraint is also trivially holds as the policy is translated directly, i.e. $\vec{k_r}[\mathrm{np}]$ is the same both in JVM type system and DEX type system, and that $rt(r_o) = k$. Since JVM typing satisfy $k \leq \vec{k_r}[\mathrm{np}]$, then so does 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 $\mathcal{S}^{\mathrm{ext}}$ and the security level of $S_i[1]$ is in the set of $\mathcal{S}$. Thus we know then know that the constraints $rt(r_o) \in \mathcal{S}$ and $rt(r_s) \in \mathcal{S}^{\mathrm{ext}}$ 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, $(rt(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} rt(r_s) \leq \mathbf{ft}(f)$ 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 $(rt(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} rt(r_s) \leq \mathbf{ft}(f)$ is also satisfied because $(k_2 \sqcup se(i)) \sqcup^{\mathrm{ext}} k_1 \leq \mathbf{ft}(f)$ 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 region, handler's existence / non-existence, and constraint against $\vec{k_r}$ on uncaught exception).

- **Newarray**
  Trivially holds as the instruction **Newarray** does not have any constraints.

- **Arraylength**
  We first deal with the constraints $k \in \mathcal{S}$ and $k_c \in \mathcal{S}^{\mathrm{ext}}$. 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 constraints. For the rest of the constraints refer to the proof in **Getfield** as they are essentially the same (the constraint for 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 $k_c \in \mathcal{S}^{\mathrm{ext}}$. 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 $k_c \in \mathcal{S}^{\mathrm{ext}}$ since in JVM typing $k_c \in \mathcal{S}^{\mathrm{ext}}$. For the rest of the constraints refer to the proof in **Getfield** as they are essentially the same (the constraint for 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 $k_1, k_c \in \mathcal{S}^{\mathrm{ext}}$ gives us $k'_c, rt(r_s) \in \mathcal{S}^{\mathrm{ext}}$. In this setting as well, it is easy to show that DEX typing satisfies $((k' \sqcup rt(r_i)) \sqcup^{\mathrm{ext}} rt(r_s)) \leq^{\mathrm{ext}} k'_c$ because JVM typing satisfies $((k_2 \sqcup k_3) \sqcup^{\mathrm{ext}} k_1) \leq^{\mathrm{ext}} k_c$. For the rest of the constraints refer to the proof in **Getfield** as they are essentially the same (the constraint for region, handler's existence / non-existence, and constraint against $\vec{k_r}$ on uncaught exception).

- **Invokevirtual**
  There will be 3 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 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 satisfy $k \sqcup k_h \sqcup se(i) \leq k'_h$. We also know that $rt(\vec{p}[0]) \leq \vec{k'_a}[0]$ since $k \leq \vec{k'_a}[0]$. 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 $\forall j \in \mathbf{region}(i, \mathrm{Norm}).rt(\vec{p}[0]) \sqcup k_e \leq se(j)$ is satisfied.

The second case is when method invocation thrown a caught exception. Basically the same arguments as that of normal execution, except that the region condition is based upon particular exception $\forall j \in$ **region**$(i, e)$. $rt(\vec{p}[0]) \sqcup k'_r[e] \leq se(j)$. Since the policy stays the same, JVM instruction satisfy 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 $e \in$ **excAnalysis**$(m_{\text{ID}}) \sqcup \{np\}$ in JVM side implies the satisfaction of $e \in$ **excAnalysis**$(m') \sqcup \{np\}$ in DEX side.

The last case is when method invocation thrown an uncaught exception. Same argument as the caught exception with the addition that escaping exception are 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 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**

  Similar arguments to that of **Invokevirtual** addressing the similar form of the constraints. In the case of caught exception case, the constraint $e \in$ **classAnalysis**$(i) \cup \{np\}$ is satisfied because, as before, **classAnalysis** and classes $(e)$ are the same. So, if JVM program satisfy the constraint the translated DEX program will also satisfy it. The same with $\forall j \in$ **region**$(i, e) rt(r) \leq se(j)$ since $rt(r) = k$.

  The case where exception is uncaught is the same as the caught case with addition that the security level of 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.

  ■

This lemma states that a typable JVM program (block wise and within blocks) will translate into typable DEX program.

**Lemma** (VII.7). *Let $P$ be a JVM program such that*

$$\forall i, j.i \mapsto j. \exists st.i \vdash S_i \Rightarrow st \quad and \quad st \sqsubseteq S_j$$

*Then $\|P\|$ will be*

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*

$$
\begin{array}{llll}
RTs_{bi} = \|S_i\| & with & \|i\| = (bi, 0) \\
RTs_{bj} = \|S_j\| & with & \|j\| = (bj, 0), \\
RT_{(bi,i)} = \|S_{i'}\| & when & \|i'\| = (bi, i) \\
RT_{(bi,j)} = \|S_{j'}\| & when & \|j'\| = (bj, j)
\end{array}
$$

*Proof:* For the first property, they are mainly proved using Lemma VII.5 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 VII.5, 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 VII.3, and for throwing instructions we use the first case of Lemma VII.4.

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 $j$ comes from the same JVM instruction or not. If $i$ and $j$ comes from the same JVM instruction, then we use the first case of Lemma VII.3. Otherwise, we use Lemma VII.5.

■

Before we proceed to the proof of Lemma VII.8, we define a property which is satisfied after the ordering and output phase.

**Property B.1.** *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. 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** (VII.8). *Let $\|P\|$ be a typable DEX blocks resulted from translation of JVM instruction still in the block form, i.e.*

$$\|P\| = \textbf{Translate}(\textbf{TraceParentChild}(\textbf{StartBlock}(P)))$$

*Given the ordering scheme to output the block contained in **PickOrder**, if the starting block starts with flag 0 ($F_{(0,0)} = 0$) 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's left is in ensuring that this successor relation 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 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 program output.

- 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 registers typing for the next instruction (in another word $RT_{(bj,0)}$). Therefore, the typability property trivially holds.

  The second case is that the successor block is not the next block in order. According to step performed in the **Output** phase, the property B.1 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 registers 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 complementary, 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** and 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 registers 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 **moveresult** is not yet ordered, we know that it will be the next 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**.

∎

## Appendix C
### Full JVM Operational Semantics and Transfer Rules

The following figure 7 is the full operational semantics for JVM in section III. The function **fresh** : Heap $\to \mathcal{L}$ is an allocator function that given a heap returns the location for that object. The function **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 numeric type, and $null$ if the field is of object type. Similarly **defaultArray** : $\mathbb{N} \times \mathcal{T}_J \to (\mathbb{N} \rightharpoonup \mathcal{V})$. The $\leadsto$ relation which defines transition between state is $\leadsto \subseteq \mathbf{State} \times (\mathbf{State} + \mathcal{V} \times \mathbf{Heap})$.

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.

For method invocation, program comes equipped with a set $\mathcal{M}$ of method names, and for each method $m$ there are associated list of instructions $P_m$. Each method is identified by method identifier $m_{\mathrm{ID}}$ which can refer to several methods in the case of overriding. Therefore we also need to know which class this method is invoked from, which can be identified by auxilliary function **lookup$_\mathbf{P}$** which returns the precise method to be executed based on the method identifier and class.

To handle exception, program will also comes 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.

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

The next figure 8 is the full version of figure 3 in section III. The full typing judgement takes the form of $\Gamma, \mathbf{ft}, \mathbf{region}, sgn, se, i \vdash^\tau st \Rightarrow st'$ where $\Gamma$ is the table of method policies, $\mathbf{ft}$ is the global policy for fields, **region** is the CDR information for the current method, $sgn$ is the policy for the current method taking the form of $\vec{k_a} \overset{k_h}{\to} \vec{k_r}$, $se$ is the security environment, $i$ is the current program point, $st$ is the stack typing for the current instruction, and $st'$ is the stack typing after the instruction is executed.

As in the main paper, 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 $\leadsto$, e.g. we may write $\leadsto$ instead of $\leadsto_{m,\mathrm{Norm}}$ to mean the same thing. In the table of transfer rules, we may drop the superscript of tag from $\vdash^\tau$ and write $\vdash$ instead. The same case applies to the typing judgement, we may write $i \vdash^\tau st \Rightarrow st'$ instead of $\Gamma, \mathbf{ft}, \mathbf{region}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash^\tau st \Rightarrow st'$.

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

$$\frac{P_m[i] = \textbf{ifeq } j \quad n \neq 0}{\langle i, \rho, n :: os \rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, os \rangle} \qquad \frac{P_m[i] = \textbf{ifeq } j \quad n = 0}{\langle i, \rho, n :: os \rangle \rightsquigarrow_{m,\text{Norm}} \langle j, \rho, os \rangle} \qquad \frac{P_m[i] = \textbf{store } x \quad x \in \textbf{dom}(\rho)}{\langle i, \rho, v :: os \rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho \oplus \{x \mapsto v\}, os \rangle}$$

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

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

$$\frac{P_m[i] = \textbf{getfield } f \quad l' = \textbf{fresh}(h)}{\langle i, \rho, null :: os, h \rangle \rightsquigarrow_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)} \qquad \frac{P_m[i] = \textbf{push } n}{\langle i, \rho, os \rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, n :: os \rangle}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 7: Full JVM Operational Semantic

$$\frac{P_m[i] = \textbf{load } x}{se, i \vdash st \Rightarrow (\vec{k_a}(x) \sqcup se(i)) :: st} \qquad \frac{P_m[i] = \textbf{store } x \qquad se(i) \sqcup k \le \vec{k_a}(x)}{se, i \vdash k :: st \Rightarrow st} \qquad \frac{P_m[i] = \textbf{swap}}{i \vdash k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st}$$

$$\frac{P_m[i] = \textbf{ifeq } j \qquad \forall j' \in \textbf{region}(i, \text{Norm}), k \le se(j')}{\textbf{region}, se, i \vdash k :: st \Rightarrow \textbf{lift}_{\mathbf{k}}(st)} \qquad \frac{P_m[i] = \textbf{goto } j}{i \vdash st \Rightarrow st} \qquad \frac{P_m[i] = \textbf{return} \qquad se(i) \sqcup k \le k_r[n]}{\vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash k :: st \Rightarrow}$$

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

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

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

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

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

$$\frac{P_m[i] = \textbf{putfield } f \qquad (se(i) \sqcup k_2) \sqcup^{\text{ext}} k_1 \le \textbf{ft}(f) \qquad k_1 \in \mathcal{S}^{\text{ext}} \qquad k_2 \in \mathcal{S} \qquad k_h \le \textbf{ft}(f) \\ \forall j \in \textbf{region}(i, \text{Norm}), k_2 \le se(j)}{\Gamma, \textbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \textbf{region}, se, i \vdash^{\text{Norm}} k_1 :: k_2 :: st \Rightarrow \textbf{lift}_{\mathbf{k_2}}(st)}$$

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c} P_m[i] = \mathbf{invoke}\ m_{\mathrm{ID}} \quad \mathbf{length}(st_1) = \mathbf{nbArguments}(m_{\mathrm{ID}}) \quad \Gamma_{m_{\mathrm{ID}}}[k] = \vec{k_a'} \overset{k_h'}{\to} \vec{k_r'} \\ \forall i \in [0, \mathbf{length}(st_1) - 1].st_1[i] \le \vec{k_a'}[i+1] \quad k \le \vec{k_a'}[0] \quad k \sqcup k_h \sqcup se(i) \le k_h' \\ e \in \mathbf{excAnalysis}(m_{\mathrm{ID}}) \cup \{\mathrm{np}\} \quad \mathbf{Handler}(i,e) = t \quad \forall j \in \mathbf{region}(i,e), k \sqcup \vec{k_r'}[e] \le se(j) \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^e st_1 :: k :: st_2 \Rightarrow (k \sqcup \vec{k_r'}[e]) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathbf{invoke}\ m_{\mathrm{ID}} \quad \mathbf{length}(st_1) = \mathbf{nbArguments}(m_{\mathrm{ID}}) \quad \Gamma_{m_{\mathrm{ID}}}[k] = \vec{k_a'} \overset{k_h'}{\to} \vec{k_r'} \\ \forall i \in [0, \mathbf{length}(st_1) - 1].st_1[i] \le \vec{k_a'}[i+1] \quad k \le \vec{k_a'}[0] \quad k \sqcup k_h \sqcup se(i) \le k_h' \quad k \sqcup se(i) \sqcup \vec{k_r'}[e] \le \vec{k_r}[e] \\ e \in \mathbf{excAnalysis}(m_{\mathrm{ID}}) \cup \{\mathrm{np}\} \quad \mathbf{Handler}(i,e) \uparrow \quad \forall j \in \mathbf{region}(i,e), k \sqcup \vec{k_r'}[e] \le se(j) \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^e st_1 :: k :: st_2 \Rightarrow}$$

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

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

Fig. 8: JVM Transfer Rule

The following figure 9 is the full operational semantics for DEX in section IV. It is similar to that of JVM, with several differences, e.g. the state in DEX does not have operand stack but its functionality is covered by the registers (local variables) $\rho$. The function $\mathbf{fresh} : \mathrm{Heap} \to \mathcal{L}$ 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 numeric type, and $null$ if the field is of object type. Similarly $\mathbf{defaultArray} : \mathbb{N} \times \mathcal{T}_D \to (\mathbb{N} \rightharpoonup \mathcal{V})$. The $\leadsto$ relation which defines transition between state is $\leadsto \subseteq \mathbf{State} \times (\mathbf{State} + \mathcal{V} \times \mathbf{Heap})$.

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.

To handle exception, program will also comes equipped with two parameters $\mathbf{classAnalysis}$ and $\mathbf{excAnalysis}$. $\mathbf{classAnalysis}$ contains information on possible classes of exception of a program point, and $\mathbf{excAnalysis}$ contains possible escaping exception of a method.

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

The next figure 10 is the full version of figure 6 in section IV. The full typing judgement takes the form of $\Gamma, \mathbf{ft}, \mathbf{region}, sgn, se, i \vdash^\tau rt \Rightarrow rt'$ where $\Gamma$ is the table of method policies, $\mathbf{ft}$ is the global policy for fields, $\mathbf{region}$ is the CDR information for the current method, $sgn$ is the policy for the current method taking the form of $\vec{k_a} \overset{k_h}{\to} \vec{k_r}$, $se$ is the security environment, $i$ is the current program point, $rt$ is the register typing for the current instruction, $rt'$ is the register typing after the instruction is executed.

As in the main paper, 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 $m, \mathrm{Norm}$ from $\leadsto$, e.g. we may write $\leadsto$ instead of $\leadsto_{m, \mathrm{Norm}}$ to mean the same thing. In the table of transfer rules, we may drop the superscript of tag from $\vdash^\tau$ and write $\vdash$ instead. The same case applies to the typing judgement, we may write $i \vdash^\tau rt \Rightarrow rt'$ instead of $\Gamma, \mathbf{ft}, \mathbf{region}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash^\tau rt \Rightarrow rt'$.

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

$$\frac{P[i]_m = \textbf{ifeq}(r,j) \quad \rho(r) = 0}{\langle i, \rho, h\rangle \leadsto_{m,\text{Norm}} \langle t, \rho, h\rangle} \qquad \frac{P_m[i] = \textbf{ifeq}(r,t) \quad \rho(r) \neq 0}{\langle i, \rho, h\rangle \leadsto_{m,\text{Norm}} \langle i+1, \rho, h\rangle} \qquad \frac{P[i]_m = \textbf{return}(r_s) \quad r_s \in \textbf{dom}(\rho)}{\langle i, \rho, h\rangle \leadsto_{m,\text{Norm}} \rho(r_s), h}$$

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

$$\frac{P_m[i] = \textbf{iget}(r, r_o, f) \quad \rho(r_o) \in \textbf{dom}(h) \quad f \in \textbf{dom}(h(\rho(r_o)))}{\langle i, \rho, h\rangle \leadsto_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto h(\rho(r_o)).f\}, h\rangle} \qquad \frac{P_m[i] = \textbf{new}(r,c) \quad l = \textbf{fresh}(h)}{\langle i, \rho, h\rangle \leadsto \langle i+1, \rho \oplus \{r \mapsto l\}, h \oplus \{l \mapsto \textbf{default}(c)\}\rangle}$$

$$\frac{P_m[i] = \textbf{iget}(r, r_o, f) \quad \rho(r_o) = null \quad l' = \textbf{fresh}(h)}{\langle i, \rho, h\rangle \leadsto_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)} \qquad \frac{P_m[i] = \textbf{iput}(r_s, r_o, f) \quad \rho(r_o) = null \quad l' = \textbf{fresh}(h)}{\langle i, \rho, h\rangle \leadsto_{n,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)}$$

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

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

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

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

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

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

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

$$\frac{P_m[i] = \textbf{aput}(r_s, r_a, r_i) \quad \rho(r_a) = null \quad l' = \textbf{fresh}(h)}{\langle i, \rho, h\rangle \leadsto_{m,\text{np}} \textbf{RuntimeExcHandling}(h, l', \text{np}, i, \rho)} \qquad \frac{P_m[i] = \textbf{moveresult}(r) \quad r \in \textbf{dom}(\rho)}{\langle i, \rho, h\rangle \leadsto_{m,\text{Norm}} \langle i+1, \rho \oplus \{r \mapsto \rho(ret)\}, h\rangle}$$

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

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

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

$$P_m[i] = \mathbf{invoke}(n, m', \vec{p}) \quad \vec{p} \in \mathbf{dom}(\rho) \quad \langle 1, \{\vec{x} \mapsto \vec{p}\}, h \rangle \leadsto_{m'}^{+} \langle l' \rangle, h'$$
$$e = \mathbf{class}(h'(l')) \quad \mathbf{Handler_m}(i, e) \uparrow \quad e \in \mathbf{excAnalysis}(m')$$
$$\overline{\langle i, \rho, h \rangle \leadsto_{m,e} \langle l' \rangle, h'}$$

$$\frac{P_m[i] = \mathbf{throw}(r) \quad \rho(r) \in \mathbf{dom}(h) \quad e = \mathbf{class}(h(\rho(r))) \quad \mathbf{Handler_m}(i, e) = t \quad e \in \mathbf{classAnalysis}(m, i)}{\langle i, \rho, h \rangle \leadsto_{m,e} \langle t, \rho \oplus \{ex \mapsto \rho(r)\}, h \rangle}$$

$$\frac{P_m[i] = \mathbf{throw}(r) \quad \rho(r) \in \mathbf{dom}(h) \quad e = \mathbf{class}(h(\rho(r))) \quad \mathbf{Handler_m}(i, e) \uparrow \quad e \in \mathbf{classAnalysis}(m, i)}{\langle i, \rho, h \rangle \leadsto_{m,e} \langle \rho(r) \rangle, h}$$

$$\frac{P_m[i] = \mathbf{throw}(r) \quad l' = \mathbf{fresh}(h) \quad \rho(r) = null}{\langle i, \rho, h \rangle \leadsto_{m,\mathrm{np}} \mathbf{RuntimeExcHandling}(h, l', \mathrm{np}, i, \rho)} \qquad \frac{P_m[i] = \mathbf{moveexception}(r) \quad r \in \mathbf{dom}(\rho)}{\langle i, \rho, h \rangle \leadsto_{m,\mathrm{Norm}} \langle i+1, \rho \oplus \{r \mapsto \rho(ex)\}, h \rangle}$$

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

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

Fig. 9: DEX Operational Semantic

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

$$\frac{P_m[i] = \mathbf{return}(r_s) \quad se(i) \sqcup rt(r_s) \le \vec{k_r}[n]}{\vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash rt \Rightarrow} \qquad \frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b)}{se, i \vdash rt \Rightarrow rt \oplus \{r \mapsto (rt(r_a) \sqcup rt(r_b) \sqcup se(i))\}}$$

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

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

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

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

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

$$\frac{P_m[i] = \mathbf{iput}(r, r_o, f) \quad rt(r) \in \mathcal{S}^{\mathrm{ext}} \quad rt(r_o) \in \mathcal{S} \quad (rt(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} rt(r_0) \le \mathbf{ft}(f) \quad k_h \le \mathbf{ft}(f)}{\forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(r_o) \le se(j)}$$
$$\overline{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c} P_m[i] = \mathbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \le k_h' \quad \forall 0 \le i < n.rt(\vec{p}[i]) \le \vec{k_a'}[i] \\ k_e = \bigsqcup \{\vec{k_r'}[e] \mid e \in \mathbf{excAnalysis}(m')\} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), rt(\vec{p}[0]) \sqcup k_e \le se(j) \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow \left(rt \oplus \{ret \mapsto \vec{k_r'}[n] \sqcup se(i)\}\right)}$$

$$\frac{\begin{array}{c} P_m[i] = \mathbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \le k_h' \quad \forall 0 \le i < n.rt(\vec{p}[i]) \le \vec{k_a'}[i] \\ \mathbf{Handler}(i, e) = t \quad e \in \mathbf{excAnalysis}(m') \cup \{\mathrm{np}\} \quad \forall j \in \mathbf{region}(i, e), rt(\vec{p}[0]) \sqcup \vec{k_r'}[e] \le se(j) \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \mathbf{region}, se, i \vdash^{e} rt \Rightarrow \vec{k_a} \oplus \{ex \mapsto (rt(\vec{p}[0]) \sqcup \vec{k_r'}[e])\}}$$

$$\frac{\begin{array}{c} P_m[i] = \mathbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[rt(\vec{p}[0])] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \quad rt(\vec{p}[0]) \sqcup k_h \sqcup se(i) \le k_h' \quad \forall 0 \le i < n.rt(\vec{p}[i]) \le \vec{k_a'}[i] \\ rt(\vec{p}[0]) \sqcup se(i) \sqcup \vec{k_r'}[e] \le \vec{k_r}[e] \quad e \in \mathbf{excAnalysis}(m') \cup \{\mathrm{np}\} \\ \mathbf{Handler}(i, e) \uparrow \quad \forall j \in \mathbf{region}(i, e), rt(\vec{p}[0]) \sqcup k_r'[e] \le se(j) \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \mathbf{region}, se, i \vdash^{e} rt \Rightarrow}$$

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

$$\frac{\begin{array}{c} P_m[i] = \mathbf{throw}(r) \quad e \in \mathbf{classAnalysis}(i) \cup \{\mathrm{np}\} \quad \forall j \in \mathbf{region}(i, e), rt(r) \le se(j) \\ \mathbf{Handler}(i, e) \uparrow \quad se(i) \sqcup rt(r) \le \vec{k_r}[e] \end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, \mathbf{region}, se, i \vdash^{e} rt \Rightarrow}$$

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

Fig. 10: DEX Transfer Rule

## APPENDIX E
## AUXILLIARY LEMMAS

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

**Lemma E.1.** *Let $k \in S$ a security level, for all heap $h \in \mathrm{Heap}$ and object / array $o \in \mathcal{O}$ (or $o \in \mathcal{A}$), $h \preceq_k h \oplus \{\mathbf{fresh}(h) \mapsto o\}$*

**Lemma E.2.** *For all heap $h, h_0 \in \mathrm{Heap}$, object $o \in \mathcal{O}$ and $l = \mathbf{fresh}(h)$, $h \sim_\beta h_0$ implies $h \oplus \{l \mapsto o\} \sim_\beta h_0$*

**Lemma E.3.** *For all heap $h, h_0 \in \mathrm{Heap}$ and $\mathbf{ft}(f) \not\le k_{\mathrm{obs}}$, $h \sim_\beta h_0$ implies $h \oplus \{l \mapsto h(l) \oplus \{f \mapsto v\}\} \sim_\beta h_0$*

**Lemma E.4.** *For all heap $h, h_0 \in \mathrm{Heap}$, $r \in R$, $\rho \in R \rightharpoonup V$, $rt \in (\mathcal{R} \to \mathcal{S})$, $\rho(r) \in \mathbf{dom}(h)$, $\rho(r)$ is an array, integer $0 \le i < h(\rho(r)).\mathbf{length}$, $rt(\rho(r)) = k[k_c]$ and $k_c \not\le^{\mathrm{ext}} k_{\mathrm{obs}}$, $h \sim_\beta h_0$ implies $h \oplus \{\rho(r) \mapsto h(\rho(r)) \oplus \{i \mapsto v\}\} \sim_\beta h_0$*

**Lemma E.5.** *For all heap $h, h', h_0 \in \mathrm{Heap}$, $k \not\le k_{\mathrm{obs}}$, and $h \preceq_k h'$, $h \sim_\beta h_0$ implies $h' \sim_\beta h_0$*

**Lemma E.6.** *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*

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

**Lemma E.7.** *$\rho_1 \sim_{rt_1, rt_2, \beta} \rho_2$ implies for any register $r \in \rho_1$ :*

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

## APPENDIX F
## PROOF THAT TYPABLE $\mathrm{DEX}_{\mathcal{I}}$ IMPLIES NON-INTERFERENCE

In this appendix, we present the soundness of our type system for DEX program i.e. typable DEX program implies that the program is safe. We also base our proof construction

on the work Barthe et. al., including the structuring of the submachine. In the paper, we present the type system for the aggregate of the submachines. In the proof construction, we will have 4 submachines: standard instruction without modifying the heap ($DEX_\mathcal{I}$), object and array instructions ($DEX_\mathcal{O}$), method invocation ($DEX_\mathcal{C}$), and exception mechanism ($DEX_\mathcal{G}$).

There are actually more definitions on indistinguishability that would be required to establish that typability implies non-interference. Before we go to the definition of operand stack indistinguishability, there is a definition of high registers : let $\rho \in (\mathcal{R} \rightharpoonup \mathcal{V})$ be register mapping and $rt \in (\mathcal{R} \to \mathcal{S})$ be a registers typing.

Several notes here in this submachine, since the execution is always expected to return normally, 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, therefore we will drop them from the proofs.

**Definition F.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 $\langle i, \rho \rangle \sim_{\vec{k}_a, rt, rt'} \langle i', \rho' \rangle$, iff $\rho \sim_{\vec{k}_a, rt, rt'} \rho'$*

**Lemma F.1** (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 two registers types $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ such that $s_1 \sim_{\vec{k}_a, rt_1, rt_2} s_2$.*

- *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 $s_1' \sim_{\vec{k}_a, rt_1', rt_2'} s_2'$.*

- *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 $k_r \le k_{\text{obs}}$ implies $v_1 \sim v_2$.*

*Proof:* By contradiction. Assume that all the precedent are true, but the conclusion is false. That means, $s_1'$ is distinguishable from $s_2'$, which means that $\rho_{s_1'} \not\sim \rho_{s_2'}$, where $\rho_{s'}$ is part of $s_1'$ and $\rho_{s_2'}$ are parts of $s_2'$. This can be the case only if the instruction at $i$ is modifying some low values in $\rho_1$ and $\rho_2$ to have different values. We will do this by case for possible instructions :

- **move**$(r, r_s)$. This case is trivial, as the distinguishability for $\rho_{s_1'}$ and $\rho_{s_2'}$ will depend only on the source register. If the source register is low, then since we have that $\rho_1 \sim \rho_2$, they have to have the same value ($\rho_1(r_s) = \rho_2(r_s)$), therefore the value put in $r$ will be the same as well. If the source register is high, then the target register will have high security level as well (the security of both values will be $rt(r_s) \sqcup se(i)$, where $rt(r_s) \not\le k_{\text{obs}}$), thus preserving the indistinguishability.

- **binop**$(r, r_a, r_b)$. Following the argument from **move**, the distinguishability for $\rho_{s_1'}$ and $\rho_{s_2'}$ will depend only on the source registers. If source registers are low, then since we have that $\rho_1 \sim \rho_2$, they have to have the same values ($\rho_1(r_a) = \rho_2(r_a)$ and $\rho_1(r_b) = \rho_2(r_b)$), therefore the result of binary operation will be the same (no change in indistinguishability). If any of the source register is high, then the target register will have high security level as well (the security level of the resulting value will be $rt(r_a) \sqcup rt(r_b) \sqcup se(i)$,

where $rt(r_a) \not\le k_{\text{obs}}$ and/or $rt(r_b) \not\le k_{\text{obs}}$), thus preserving the indistinguishability.

- **const**$(r, v)$. Nothing to prove here, the instruction will always give the same value anyway, regardless whether the security level of the register to store the value is high or low.

- **goto**$(j)$. Nothing to prove here, as the instruction only modify the program counter.

- **return**$(r_s)$. This is a slightly different case here than before, where we are comparing the results instead of the state ($v_1 \sim v_2$). Again, the reasoning is that to have different result and they are distinguishable, we need the register from which the value is returned to be high ($rt(r_s) \not\le k_{\text{obs}}$), but the security level of the return value of the method is low ($k_r \le k_{\text{obs}}$). But this is already taken care of by the transfer rule which state $rt(r_s) \le k_r$. Therefore, a contradiction.

- **ifeq**$(r, t)$. A special case where there might be a branching thus the states compared are at two different program counters. If the register used in comparison is low ($rt(r) \le k_{\text{obs}}$), we know that the program counter will be the same and there will be nothing left to prove (**ifeq** is just modifying program counter). If the register is high ($rt(r) \not\le k_{\text{obs}}$), the operational semantics tells us that there is no modification to the registers. Therefore, register wise these two states are indistinguishable.

■

**Lemma F.2** (High Branching). *Let $s_1, s_2 \in \text{State}_I$ be two $DEX_\mathcal{I}$ states at the same program point $i$ and let two registers types $rt_1, rt_2 \in (\mathcal{R} \to \mathcal{S})$ such that $s_1 \sim_{\vec{k}_a, rt_1, rt_2} s_2$. If two states $\langle i_1, \rho_1' \rangle, \langle i_2, \rho_2' \rangle \in \text{State}_I$ and two registers type $rt_1', rt_2' \in (\mathcal{R} \to \mathcal{S})$ s.t. $i_1 \ne i_2$, $s_1 \rightsquigarrow \langle i_1, \rho_1' \rangle$, $s_2 \rightsquigarrow \langle i_2, \rho_2' \rangle$, $i \vdash rt_1 \Rightarrow rt_1'$, $i \vdash rt_2 \Rightarrow rt_2'$ then $\forall j \in \textbf{region}(i), se(j) \not\le k_{\text{obs}}$.*

*Proof:* This is already by definition of the branching instruction (**ifeq** and **ifneq**). $se(i)$ will be high because $r$ will by definition be high. This level can not 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 for scope of the region, we have $\forall j \in \textbf{region}(i), se(j) \not\le k_{\text{obs}}$.

■

**Lemma F.3** (indistinguishablility double monotony). *if $s \sim_{\vec{k}_a, S, T} t, S \sqsubseteq U$, and $T \sqsubseteq U$ then $s \sim_{\vec{k}_a, U, U} t$*

**Lemma F.4** (indistinguishablility single monotony). *if $s \sim_{k_{\text{obs}}, S, T} t, S \sqsubseteq S'$ and $S$ is high then $s \sim_{k_{\text{obs}}, S', T} t$*

APPENDIX G
PROOF THAT TYPABLE $DEX_\mathcal{O}$ IMPLIES NON-INTERFERENCE

Indistinguishability between states can be defined with the additional definition of heap indistinguishability, so we do not need additional indistinguishability definition. In the $DEX_\mathcal{O}$ part, we only need to appropriate the lemmas used to establish the proof.

**Definition G.1** (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 registers typing $rt, rt' \in (\mathcal{R} \rightarrow \mathcal{S})$, denoted $\langle i, \rho, h \rangle \sim_{\vec{k_a}, rt, rt', \beta} \langle i', \rho', h' \rangle$, iff $\rho \sim_{\vec{k_a}, rt, rt'} \rho'$ and $h \sim_\beta h'$ hold.

**Lemma G.1** (Locally Respects). *Let $\beta$ a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \text{State}_O$ be two $DEX_O$ states at the same program point $i$ and let two registers types $rt_1, rt_2 \in (\mathcal{R} \rightarrow \mathcal{S})$ such that $s_1 \sim_{\vec{k_a}, rt_1, rt_2, \beta} s_2$.*

- *Let $s'_1, s'_2 \in \text{State}_O$ and $rt'_1, rt'_2 \in (\mathcal{R} \rightarrow \mathcal{S})$ such that $s_1 \leadsto s'_1, s_2 \leadsto s'_2, i \vdash rt_1 \Rightarrow rt'_1$, and $i \vdash rt'_2 \Rightarrow rt'_2$, then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $s'_2 \sim_{\vec{k_a}, rt'_1, rt'_2, \beta'} s'_2$ and $\beta \subseteq \beta'$.*

- *Let $v_1, v_2 \in \mathcal{V}$ such that $s_1 \leadsto v_1$, $s_2 \leadsto v_2$, $i \vdash rt_1 \Rightarrow$, and $i \vdash rt'_2 \Rightarrow$, then $k_r \leq k_{\text{obs}}$ implies $v_1 \sim_\beta v_2$.*

*Proof:* By contradiction. Assume that all the precedent are true, but the conclusion is false. That means, $s'_1$ is distinguishable from $s'_2$, which means either $\rho'_1 \not\sim \rho'_2$ or $h'_1 \not\sim h'_2$, where $\rho'_1, h'_1$ are parts of $s'_1$ and $\rho'_2, h'_2$ are parts of $s'_2$.

- assume $h'_1 \not\sim h'_2$. This can be the case only if the instruction at $i$ are **iput**, **newarray**, and **aput**.
  - **iput**$(r_s, r_o, f)$ can only cause the difference by putting different values $(\rho_1(r_s) \neq \rho_2(r_s))$ with $rt_1(r_s) \not\leq k_{\text{obs}}$ and $rt_2(r_s) \not\leq k_{\text{obs}}$ on a field where $\mathbf{ft}(f) \leq k_{\text{obs}}$. But the transfer rule for **iput** states that the security level of the field has to be at least as high as $r_s$, i.e. $sec \leq \mathbf{ft}(f)$ where $sec = rt_1(r_s) = rt_2(r_s)$. A plain contradiction.
  - **aput**$(r_s, r_a, r_i)$ can only cause the difference by putting different values $(\rho_1(r_s) \neq \rho_2(r_s))$ with $rt_1(r_s) \not\leq k_{\text{obs}}$ and $rt_2(r_s) \not\leq k_{\text{obs}}$ on an array whose content is low $(k_c \leq k_{\text{obs}}, k[k_c]$ is the security level of the array). But the typing rule for **aput** states that the security level of the array content has to be at least as high as $r_s$, i.e. $sec \leq k_c$ where $sec = rt_1(r_s) = rt_2(r_s)$. A plain contradiction.
  - **newarray**$(r_a, r_l, t)$ can only cause the difference by creating array of different lengths $(\rho_1(r_l) \neq \rho_2(r_l))$ with $rt_1(r_s) \not\leq k_{\text{obs}}$ and $rt_2(r_s) \not\leq k_{\text{obs}}$. But if that's the case, then that means this new array does not have to be included in the mapping $\beta'$ and therefore the heap will stay indistinguishable. A contradiction.

- assume $\rho'_1 \not\sim \rho'_2$. This can be the case only if the instruction at $i$ is modifying some low values in $\rho'_1$ and $\rho'_2$ to have different values. There are only three possible instructions in this extended submachine which can cause $\rho'_1 \not\sim \rho'_2$: **iget**, **aget**, and **arraylength**. We already have by the assumption that the original state is indistinguishable, which means that the heaps are indistinguishable as well $(h_1 \sim h_2)$. Based on the transfer rule we have that the security level of the value put in the target register will at least be as high as the source. If the security is low, we know from the assumption of indistinguishability that the value is the same, thus it will maintain registers indistinguishability. If the security is high, then the value put

in the target register will also have high security level, maintaining registers indistinguishability.

∎

**Lemma G.2** (High Branching). *Let $\beta$ a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \text{State}_O$ be two $DEX_O$ states at the same program point $i$ and let two registers types $rt_1, rt_2 \in (\mathcal{R} \rightarrow \mathcal{S})$ such that $s_1 \sim_{\vec{k_a}, rt_1, rt_2, \beta} s_2$. 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 registers type $rt'_1, rt'_2 \in (\mathcal{R} \rightarrow \mathcal{S})$ s.t. $i_1 \neq i_2, s_1 \leadsto \langle i_1, \rho'_1, h'_1 \rangle$, $s_2 \leadsto \langle i_2, \rho'_2, h'_2 \rangle$. If $i \vdash rt_1 \Rightarrow rt'_1, i \vdash rt_2 \Rightarrow rt'_2$ then $\forall j \in \mathbf{region}(i), se(j) \not\leq k_{\text{obs}}$*

## APPENDIX H
## PROOF THAT TYPABLE $\text{DEX}_\mathcal{C}$ IMPLIES SECURITY

Since now the notion of secure program also defined with *side-effect-safety* due to method invocation, we also need to establish that typable program implies that it is *side-effect-safe*. We show this by showing the property that all instruction step transforms a heap $h$ into a heap $h'$ s.t. $h \preceq_{k_h} h'$.

**Lemma H.1.** *Let $\langle i, \rho, h \rangle, \langle i', \rho', h' \rangle \in \text{State}_C$ be two states s.t. $\langle i, \rho, h \rangle \rangle \leadsto_m \langle i', \rho', h' \rangle$. Let two registers types $rt, rt' \in (\mathcal{R} \rightarrow \mathcal{S})$ s.t. $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} k_r, i \vdash^{\text{Norm}} rt \Rightarrow rt'$ 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 E.1 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 H.2.** *Let $\langle i, \rho, h \rangle \in \text{State}_C$ be a state, $h' \in \mathbf{Heap}$, and $v \in V$ s.t. $\langle i, \rho, h \rangle \leadsto v, h'$. Let $rt \in (\mathcal{R} \rightarrow \mathcal{S})$ s.t. $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} k_r, i \vdash^{\text{Norm}} rt \Rightarrow$, then $h \preceq_{k_h} h'$*

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

**Lemma H.3.** *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 with respect to $\mathbf{region_m}$ and to 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. $\langle i, \rho, h \rangle \leadsto_m \langle i', \rho', h' \rangle$. Let two registers types $rt, rt' \in (\mathcal{R} \rightarrow \mathcal{S})$ s.t. $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} k_r, i \vdash^{\text{Norm}} rt \Rightarrow rt'$ 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 throws an exception. Therefore, for any method $m_0$ called by **invoke**, there can be one or more chain of call

$$m_0 \leadsto m_1 \leadsto ... \leadsto m_n$$

where $m \leadsto m'$ signify that an instruction in method $m$ calls $m'$. Since the existence of such 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 H.1 and Lemma H.2 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 $m_0 \leadsto m_1$ is dropped from the call chain, we know that **invoke** on $m_1$ will fulfill *side-effect-safety*. Since all possible instructions are maintaining *side-effect-safety*, we know that this lemma holds. ∎

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

**Lemma H.4.** *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 with respect to* $\mathbf{region_m}$ *and to all signatures in* $\mathbf{Policies_\Gamma}(m)$. *Then all method $m$ is* 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 for exception throwing one in the subsequent machine).

**Lemma H.5** (Locally Respect Lemma). *Let $P$ a program and a table of signature $\Gamma$ s.t. all of its method $m'$ 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 $\text{DEX}_C$ states at the same program point $i$ and two registers types $rt_1, rt_2 \in (\mathcal{R} \rightarrow \mathcal{S})$ 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 $rt'_1, rt'_2 \in (\mathcal{R} \rightarrow \mathcal{S})$ s.t.*

$$s_1 \leadsto_m s'_1 \quad and \quad \Gamma, \text{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash rt_1 \Rightarrow rt'_1$$

*and*

$$s_2 \leadsto_{m'} s'_2 \quad and \quad \Gamma, \text{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash rt_2 \Rightarrow rt'_2$$

*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:* By contradiction. Assume that all the precedent are true, but the conclusion is false. That means, $s'_1$ is distinguishable from $s'_2$, which means either $\rho'_1 \nsim \rho'_2$ or $h'_1 \nsim h'_2$, where $\rho'_1, h'_1$ are parts of $s'_1$ and $\rho'_2, h'_2$ are parts of $s'_2$.

- Assume $h'_1 \nsim h'_2$. **invoke**$(m, n, \vec{p})$ can only cause the state to be distinguishable if the arguments passed to the function have some difference. And since the registers are indistinguishable in the initial state, this means that those registers with different values are

registers with security level higher than $k_{\text{obs}}$ (let's say this register $x$). By the transfer rules of **invoke**, this will imply that the $\vec{k'_a}[x] \nleq k_{\text{obs}}$ (where $0 \leq x \leq n$). Now assume that there is an instruction in $m$ using the value in $x$ to modify the heap, which can not be the case because in $\text{DEX}_\mathcal{O}$ we already proved that the transfer rules prohibit any object / array manipulation instruction to update the low field / content with high value.

- Assume $\rho_{s'} \nsim \rho_{t'}$. **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 (which will make $ret$ also indistinguishable). As for **moveresult**, we can follow the arguments in **move** (in $\text{DEX}_\mathcal{I}$), except that the source is now the pseudo-register $ret$.

∎

# APPENDIX I
## PROOF THAT TYPABLE $\text{DEX}_\mathcal{G}$ IMPLIES SECURITY

Like the one in $\text{DEX}_\mathcal{C}$, 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 $\text{DEX}_\mathcal{C}$. The only difference is that there is a possibility for 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 do not concern whether the returned value $r$ is in $L$ or in $V$. The one case for **throws** use the same lemma E.1 as it differs only in the allocation of exception in the heap. The complete definition :

**Lemma I.1.** *Let $\langle i, \rho, h \rangle, \langle i', \rho', h' \rangle \in \text{State}_G$ be two states s.t. $\langle i, \rho, h \rangle \leadsto_m \langle i', \rho', h' \rangle$. Let two registers types $rt, rt' \in (\mathcal{R} \rightarrow \mathcal{S})$ s.t. $\mathbf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash rt \Rightarrow rt'$ and $P[i] \neq$ **invoke**, then $h \preceq_{k_h} h'$*

*Proof:* The only instruction that can cause this difference are array / object manipulation instructions that throws a null pointer exception. For creating new objects or arrays and allocating the space for exception, Lemma E.1 shows that they still preserve the *side-effect-safety*. **throw** instruction itself does not allocate space for exception, so no modification to the heap. ∎

**Lemma I.2.** *Let $\langle i, \rho, h \rangle \in \text{State}_G$ be a state, $h' \in \mathbf{Heap}$, and $v \in V$ s.t. $\langle i, \rho, h \rangle \leadsto v, h'$. Let $rt \in (\mathcal{R} \rightarrow \mathcal{S})$ s.t. $\mathbf{region}, se, \vec{k_a} \xrightarrow{k_h} k_r, i \vdash rt \Rightarrow$, then $h \preceq_{k_h} h'$*

*Proof:* This can only be one of two cases, either it is **return** instruction or 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 E.1 to conclude that it will still make $h \preceq_{k_h} h'$ holds ∎

**Lemma I.3.** *Let for all method $m \in P$, ($\mathbf{region}_m$, $\mathbf{jun}_m$) a safe CDR for $m$. Suppose all methods $m \in P$ are typable w.r.t. $\mathbf{region}_m$ and to all signatures in $\mathbf{Policies_\Gamma}(m)$. Let $\langle i, \rho, h \rangle, \langle i', \rho', h' \rangle \in \mathrm{State}_G$ be two states s.t. $\langle i, \rho, h \rangle \leadsto_m \langle i', \rho', h' \rangle$. 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 $P[i] = \mathbf{invoke}$, then $h \preceq_{k_h} h'$*

*Proof:* In the case of **invoke** executing normally, we can refer to the proof in Lemma H.3. In the case of caught exception, if it is caught then we can follow the same reasoning in Lemma I.1). In the case of uncaught exception it will fall to Lemma I.2. ∎

**Lemma I.4.** *Let for all method $m \in P$, ($\mathbf{region}_m$, $\mathbf{jun}_m$) a safe CDR for $m$. Suppose all methods $m \in P$ are typable w.r.t. $\mathbf{region}_m$ and to all signatures in $\mathbf{Policies_\Gamma}(m)$. Then all method $m$ are side-effect-safe w.r.t. the heap effect level of all the policies in $\mathbf{Policies_\Gamma}(m)$.*

*Proof:* We use the definition of typable method and Lemma I.1, Lemma I.2, and Lemma I.3. Given typable method, for a derivation

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

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.

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

Using the lemmas, then we will get

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

which 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 I.1** (High Result). *Given $(r, h) \in (V + L) \times \mathbf{Heap}$ and an output level $\vec{k_r}$, the predicate $\mathbf{highResult_{k_r}}(r, h)$ is defined as :*

$$\frac{\vec{k_r}[n] \not\preceq k_{\mathrm{obs}} \quad v \in V}{\mathbf{highResult_{k_r}}(v, h)} \quad \frac{\vec{k_r}[\mathbf{class}(h(l))] \not\preceq k_{\mathrm{obs}} \quad l \in \mathbf{dom}(h)}{\mathbf{highResult_{k_r}}(\langle l \rangle, h)}$$

**Definition I.2** (Typable Execution).

- *An execution step $\langle i, \rho, h \rangle \leadsto_{m, \tau} \langle i', \rho', h' \rangle$ 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 $\langle i, \rho, h \rangle \leadsto_{m, \tau} (r, h')$ is typable w.r.t. $RT \in PP \to (\mathcal{R} \to \mathcal{S})$ if $i \vdash^\tau RT_i \Rightarrow$*

- *An execution sequence $s_0 \leadsto_{m, \tau_0} s_1 \leadsto_{m, \tau_1} \ldots s_k \leadsto_{m, \tau_k} (r, h')$ is typable w.r.t. $RT \in PP \to (\mathcal{R} \to \mathcal{S})$ if :*
  - *$\forall i, 0 \leq i < k, s_i \leadsto_{m, \tau_i} s_{i+1}$ is typable w.r.t. $RT$;*
  - *$s_n \leadsto_{m, \tau_n} (r, h')$ is typable w.r.t. $RT$.*

**Lemma I.5** (High Security Environment High Result). *Let $\langle i, \rho, h \rangle, \langle i', \rho', h' \rangle \in \mathrm{State}_G$, $se(i)$ is high, $\langle i, \rho, h \rangle \sim_\beta \langle i', \rho', h' \rangle$ $i \mapsto$ and $\langle i, \rho, h \rangle \leadsto (r, h_r)$, then $\mathbf{highResult}(r, h_r)$ and $h_r \sim_\beta h'$.*

*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 $\vec{k_r}[n]$ will be at least as high as $se$ which is high. So by definition the lemma holds. Since return does not modify heaps, we know that $h_r \sim_\beta h'$

- 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 the lemma holds. Heap wise, we know that the exception is newly generated so we can use Lemma E.6 to say that $h_r \sim_\beta h'$.

- Iget : the transfer rule where the instruction is throwing an uncaught exception np has constraint saying that $\vec{k_r}[\mathrm{np}]$ will be at least as high as $se$ and the security level of the object. We know $se$ is high, so by definition the lemma holds. Heap wise, we know that the exception is newly generated so we can use Lemma E.6 to say that $h_r \sim_\beta h'$.

- Iput : Same as Iget.

- Aget : the transfer rule where the instruction is throwing an uncaught exception np has constraint saying that $\vec{k_r}[\mathrm{np}]$ will be at least as high as $se$ and the security level of the array. We know $se$ is high, so by definition the lemma holds. Heap wise, we know that the exception is newly generated so we can use Lemma E.6 to say that $h_r \sim_\beta h'$.

- 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 the lemma holds. Throw does not modify heap as well, so we have $h_r \sim_\beta h'$.

∎

**Lemma I.6** (High Region High Result). *Let $se$ be high in $\mathbf{region}(s, \tau)$, $\mathbf{jun}(s, \tau)$ is never defined, $\langle i_0, \rho_0, h_0 \rangle, \langle i', \rho', h' \rangle \in \mathrm{State}_G$, $\langle i_0, \rho_0, h_0 \rangle \sim_\beta \langle i', \rho', h' \rangle$ and there is an execution trace*

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

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

*Proof:* We do induction on the length of the execution. For the base case where $k$ is 0 we can use Lemma I.5. 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 $h_r \sim_\beta h'$ by Lemma E.3 and Lemma E.4. ∎

**Lemma I.7** (High Register Stays). *Let $\langle i_0, \rho_0, h_0 \rangle \in State_G$ and there is an execution step such that*

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

*and $se$ is high in $i_0, \ldots, i_k$, 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 do possible instructions. If the length of execution is 0 we have $\langle i_0, \rho_0, h_0 \rangle \leadsto_{m,\tau_0} \langle i_k, \rho_k, h_k \rangle$. The instruction is not a return point, since it contradicts the assumption. If the instruction is an instruction that modify $r$, 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 $\langle i_1, \rho_1, h_1 \rangle \leadsto_{m,\tau_1} \ldots \langle i_k, \rho_k, h_k \rangle$ ∎

**Lemma I.8** (Changed Register High). *Let $\langle i_0, \rho_0, h_0 \rangle \in State_G$ and there is an execution step such that*

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

*and $se$ is high in $i_0, \ldots, i_k$, 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 I.7 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 I.9** (Unchanged Register Stays). *Let $\langle i_0, \rho_0, h_0 \rangle \in State_G$ and there is an execution step such that*

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

*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 do possible instructions. If the length of execution is 0 we have $\langle i_0, \rho_0, h_0 \rangle \leadsto_{m,\tau_0} \langle i_k, \rho_k, h_k \rangle$. The instruction is not a return point, since it contradicts the assumption. If the instruction is an instruction that modify $r$, 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 $\langle i_1, \rho_1, h_1 \rangle \leadsto_{m,\tau_1} \ldots \langle i_k, \rho_k, h_k \rangle$. ∎

**Lemma I.10** (Junction Point Indistinguishable). *Let $\beta$ a partial function $\beta \in L \rightharpoonup L$ and $\langle i_0, \rho_0, h_0 \rangle, \langle i'_0, \rho'_0, h'_0 \rangle \in \text{State}_G$ two $DEX_G$ states such that*

$$\langle i_0, \rho_0, h_0 \rangle \sim_{RT_{i_0}, RT_{i'_0}, \beta} \langle i'_0, \rho'_0, h'_0 \rangle$$

*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*

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

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

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

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

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

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

2) $(r, h) \sim_{\vec{k_r}, \beta} (r', h')$

*Proof:* We do a case analysis on whether a junction point is defined for both of the execution traces. There are 3 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 I.8 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 I.9 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 $\langle i_j, \rho_j, h_j \rangle \sim_{RT_{i_j}, RT_{i'_{j'}}, \beta} \langle i'_{j'}, \rho'_{j'}, h'_{j'} \rangle$ 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 I.6 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 I.6 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 I.6 on both executions to obtain $\mathbf{highResult}(r, h)$ and $\mathbf{highResult}(r', h')$, hence we are in Case 2.

∎

**Lemma I.11** (High Branching). *Let all method $m'$ in $P$ are 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 types $rt_1, rt_2 \in (\mathcal{R} \rightarrow \mathcal{S})$ s.t.*

$$s_1 \sim_{rt_1, rt_2, \beta} s_2$$

1) *If there exists two states $\langle i'_1, \rho'_1, h'_1\rangle, \langle i'_2, \rho'_2, h'_2\rangle \in$ State$_G$ and two registers types $rt'_1, rt'_2 \in (\mathcal{R} \to \mathcal{S})$ s.t. $i'_1 \neq i'_2$*

$$s_1 \leadsto_{m,\tau_1} \langle i'_1, \rho'_1, h'_1\rangle \qquad s_2 \leadsto_{m,\tau_2} \langle i'_2, \rho'_2, h'_2\rangle$$
$$i \vdash^{\tau_1} rt_1 \Rightarrow rt'_1 \qquad\qquad i \vdash^{\tau_2} rt_2 \Rightarrow rt'_2$$

*then*
$$se \text{ is high in } \mathbf{region}(i, \tau_1)$$
$$se \text{ is high in } \mathbf{region}(i, \tau_2)$$

2) *If there exists a state $\langle i'_1, \rho'_1, h'_1\rangle \in$ State$_G$, a final result $(v_2, h'_2) \in (V + L) \times$ Heap and a registers types $rt'_1 \in (\mathcal{R} \to \mathcal{S})$ s.t.*

$$s_1 \leadsto_{m,\tau_1} \langle i'_1, \rho'_1, h'_1\rangle \qquad s_2 \leadsto_{m,\tau_2} (r_2, h'_2)$$
$$i \vdash^{\tau_1} rt_1 \Rightarrow rt'_1 \qquad\qquad i \vdash^{\tau_2} rt_2 \Rightarrow$$

*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 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 are non-interferent, and we have that $\rho_1 \sim_{rt_1, rt_2, \beta} \rho_2$, 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 $\vec{k'_a} \overset{k'_h}{\to} \vec{k'_r}$. 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 $\vec{k'_a} \overset{k'_h}{\to} \vec{k'_r}$. 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 exception both are uncaught, then this lemma does not apply. Assume that the $e_1$ 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 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 I.12** (Locally Respect (Specialized))**.** *Let all method $m'$ in $P$ are non-interferent w.r.t. all the policies in $\mathbf{Policies_\Gamma}(m')$. Let $m$ a method in $P$, $\beta \in L \rightharpoonup L$ a partial function, $s_1, s_2 \in$ 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. $s_1 \sim_{rt_1, rt_2, \beta} s_2$.*

1) *If there exists two states $s'_1, s'_2 \in$ State$_G$ and the program point of $s'_1$ is the same as $s'_2$ and two registers types $rt'_1, rt'_2 \in (\mathcal{R} \to \mathcal{S})$ s.t.*

$$s_1 \leadsto_{m,\tau_1} s'_1 \qquad s_2 \leadsto_{m,\tau_2} s'_2$$
$$i \vdash^{\tau_1} rt_1 \Rightarrow rt'_1 \quad i \vdash^{\tau_2} rt_2 \Rightarrow rt'_2$$

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

$$s'_1 \sim_{rt'_1, rt'_2, \beta'} s'_2 \qquad \beta \subseteq \beta'$$

2) *If there exists a state $\langle i'_1, \rho'_1, h'_1\rangle \in$ State$_G$, a final result $(r_2, h'_2) \in (V + L) \times$ Heap and a registers types $rt'_1 \in (\mathcal{R} \to \mathcal{S})$ s.t.*

$$s_1 \leadsto_{m,\tau_1} \langle i'_1, \rho'_1, h'_1\rangle \qquad s_2 \leadsto_{m,\tau_2} (r_2, h'_2)$$
$$i \vdash^{\tau_1} rt_1 \Rightarrow rt'_1 \qquad\qquad i \vdash^{\tau_2} rt_2 \Rightarrow$$

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

$$h'_1 \sim_{\beta'} h'_2, \qquad \mathbf{highResult_{k_r}}(r_2, h'_2) \qquad \beta \subseteq \beta'$$

3) *If there exists two final results $(r_1, h'_1), (r_2, h'_2) \in (V + L) \times$ Heap s.t.*

$$s_1 \leadsto_{m,\tau_1} (r_1, h'_1) \qquad s_2 \leadsto_{m,\tau_2} (r_2, h'_2)$$
$$i \vdash^{\tau_1} rt_1 \Rightarrow \qquad\qquad i \vdash^{\tau_2} rt_2 \Rightarrow$$

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

$$(r_1, h'_1) \sim_{\beta'} (r_2, h'_2) \qquad \beta \subseteq \beta'$$

*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 for instructions without exception to be 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 to only affect those that have the same successor's program point.

- **invoke**. There are 6 possible successors here, but we only consider the 4 exception related one (since one of them can be subsumed by the other) :
  **1)** One normal and one has 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 uncaught exception (the case where one throw caught exception and one throw uncaught exception is proved using similar arguments). In this case, we have one successor state while the other will return value or location (case 2). So in this case we only need to prove that $\mathbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$ (the part about heap indistinguishability is already proved). We can easily again appeal to the 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 $\vec{k_r}[e] \not\leq k_{\mathrm{obs}}$ which in turns implies that $\mathbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$, thus we are in Case 2.
  **3)** Both has caught exception If they have the same exception, then we that the content of $ex$ register will be the same thus the register will be indistinguishable. The case where the exceptions are different is not covered in this lemma since they will lead to different handlers, thus different program point.
  **4)** Both has uncaught exception (and different exception 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 $\vec{k_r'}[e_1] \not\leq k_{\mathrm{obs}}$ and $\vec{k_r'}[e_2] \not\leq k_{\mathrm{obs}}$. Then, using the transfer rules for uncaught exceptions whichs 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 normal execution and one has caught null exception. In this case we know that the lemma is not applicable since the successors have different program points.
  **2)** One is normal execution and one has 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 ($sec(r_o) \not\leq k_{\mathrm{obs}}$), otherwise $s_1 \not\sim_{rt_1, rt_2 \beta} s_2$. The transfer rule for **iget** with uncaught exception states that $sec(r_o) \leq \vec{k_r}[\mathrm{np}]$, which will give us $\vec{k_r}[\mathrm{np}] \not\leq k_{\mathrm{obs}}$, which will implies that $\mathbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$, thus we are in Case 2.
  **3)** Both has caught null exception. In this case, there are two things that needs consideration: the new objects in the heap, and the pseudo-register $ex$ containing the new null exception. Since we have $h_1 \sim_\beta h_2$

and the exception is created fresh ($l_1 = \mathbf{fresh}(h_1)$, $l_1 \mapsto \mathbf{default}_{\mathrm{np}}$, $l_2 = \mathbf{fresh}(h_2)$, $l_2 \mapsto \mathbf{default}_{\mathrm{np}}$), by lemma E.6 we have that $h_1' \sim_\beta h_2'$ 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 $l_1 \sim_{\beta'} l_2$ and this will give us $\rho_1' \sim_{\beta'} \rho_2'$ since $\rho_1' = \{ex \mapsto l_1\}$ and $\rho_2' = \{ex \mapsto l_2\}$, thus we are in Case 1.
  **4)** Both has uncaught null exception. Following the previous arguments, we have $h_1' \sim_{\beta'} h_2'$, and $l_1 \sim_{\beta'} l_2$, which will give us $(\langle l_1 \rangle, h_1') \sim_{\beta'} (\langle l_2 \rangle, h_2')$, thus we are in Case 3.

- **iput**, **aget**, and **aput**. The arguments closely follows that of **iget**

- **throw**. There are four cases to consider here :
  **1)** Two same exception. 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 $\rho_1' \sim_{\beta'} \rho_2'$ 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 states 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 to have different value, therefore $\vec{k_r}[e_1]$ and $\vec{k_r}[e_2]$ must be high as well. With this, we will have that $(r_1, h_1') \sim_{\beta'} (r_2, h_2')$ 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 $\mathbf{highResult}_{\mathbf{k_r}}(r_2, h_2')$, $h_1' \sim_{\beta'} h_2'$ (throw instruction does not modify the heap), and $\beta \subseteq \beta'$ (Case 2).

∎

**Lemma I.13** (Typable DEX Program is Non-Interferent).
*Suppose we have $\beta$ a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and $\langle i_0, \rho_0, h_0 \rangle, \langle i_0', \rho_0', h_0' \rangle \in \mathrm{State}_G$ two DEX$_G$ states s.t. $i_0 = i_0'$ and $\langle i_0, \rho_0, h_0 \rangle \sim_{RT_{i_0}, RT_{i_0'}, \beta} \langle i_0', \rho_0', h_0' \rangle$. 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.*

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

*Proof:* Following the proof in 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 I.12. There are several possible cases for the induction step:

1) $k > 0$ and $k' = 0$: then we can use case 2 of Lemma I.12 to get existence of $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ s.t.

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

Using case 2 of Lemma I.11 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 later case is rendered impossible due to SOAP3. Applying Lemma I.6 we get

$$\mathbf{highResult_{k_r}}(r, h'_1) \quad \text{and} \quad h_1 \sim_\beta h'_1 \quad [2]$$

Combining [1] and [2] we get

$$h_1 \sim_{\beta'} h'_1, \quad h_1 \sim_{\beta'} h', (r, h) \sim (r', h')$$

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 the case 1 of Lemma I.12 and induction hypothesis. Otherwise we will have registers 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 I.11 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 I.12 there exists $\beta', \beta \subseteq \beta'$ s.t. (with the help of Lemma F.4

$$\langle i_1, \rho_1, h_1 \rangle \sim_{RT_{i_1}, RT_{i'_1}, \beta'} \langle i'_1, \rho'_1, h'_1 \rangle$$

Invoking Lemma I.10 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 $\langle i_j, \rho_j, h_j \rangle \sim_{RT_{i_j}, RT_{i'_j}, \beta} \langle i'_j, \rho'_j, h'_j \rangle$. We can then use induction hypothesis on the rest of executions to conclude.
- $(r, h) \sim_{\vec{k_a}, \beta'} (r', h')$ and we can directly conclude.

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

∎

Proof of Theorem IV.1 is direct application of Lemma I.13 and Lemma I.4.