Formal Certification of Android Bytecode

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

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

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. As a side note, during the writing of this work Android is moving towards a new runtime framework called ART. Nevertheless our work is still applicable as essentially both Dalvik VM and ART still use the same DEX instructions, so our analysis is not affected by this move.

An Android application is typically written by developers in Java and compiled to Java classes (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 provided, and produce a DEX file to be bundled together with other resource files in the APK. These DEX files contain the actual bytecode that gets executed by the Dalvik VM. Given that a lot of work has been done in developing security analysis techniques for Java (both source and bytecode), it is natural to ask whether security guarantees achieved through these techniques for Java source codes or bytecode can be carried over to DEX bytecode. A crucial step here is to study the properties of the translation from Java bytecode to DEX bytecode, as implemented in the dx tool. We provide what we believe is the first such study, for a subset of information flow related properties, formalized via a notion of non-interference. We follow a language-based approach to non-interference, i.e., we design a type system for DEX bytecode such that typability implies non-interference.

With that goal in mind, our work is divided into several parts. Firstly, we provide a formal operational semantics of DEX instructions, and propose the design of a non-interference type system for DEX instructions which closely resembles the one designed for the JVM by Barthe et. al. [3]. We show that our type system for DEX is sound w.r.t. the non-interference property, that is if the DEX bytecode is typable, then it is non-interferent. Then we show that the idealized (non-optimizing) compilation done by the dx tool preserves typability: if the source Java bytecode are typeable in Barthe et. al's type system, then the resulting translation in DEX bytecode are also typeable. By the type soundness result for DEX, this also means that the translation preserves the class of non-inteference properties captured by Barthe et. al's type system. When combined with existing work on type-preserving compilation from Java source codes to JVM, such as [4], we could in principle produce a formally certified Android bytecode from its Java source.

The development of the operational semantics and the type systems for DEX bytecode follows closely the framework set up in [3]. 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 [3], 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. However, one of the registers in DEX, i.e., the first register r_0, can serve different purposes during a run of a program, e.g., it is used to store the return value within a function in addition to other uses, so a naive adaptation of operand stack types to register types will result in most non-interferent programs being rejected by the type systems, e.g., in situations where register such as r_0 is used to hold high value and the return type of the function is specified as low value. We thus have to introduce additional type annotations to distinguish between different uses of the same register (see Section IV). This additional type annotations complicates slightly the type-preserving proof of the bytecode translation. 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. [5], [3] on enforcing non-interference via type systems. We discuss other related work in the following.

Bian et. al. [6] 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. [5] 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 [4]. 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 [7], [8], [9], [10], [11], [12], [13], [14], [15], [16] although some of them are geared towards the privilege escalation problem. These works base their context of Android security studied in [16]. 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 [7], with the underlying type system and static analysis tool for security specification in Android [17]. 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 [8]. 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. TrustDroid [10] and ScanDal [11]. TrustDroid is another static analysis tool on Android bytecode, trying to prevent information leaking. TrustDroid is more interested in doing taint analysis on the program, although different from TaintDroid [9] 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 targetting the DEX instructions directly, aggregating the instructions in a language they call Dalvik Core. They enumerate all possible states and note when any value from any predefined information source is flowing through a predefined information sink. Their work assumed that predefined sources and sinks are given, whereas we are more interested in a flexible policy to define them.

Since the property that we are interested in is non-interference, it is also worth mentioning Sorbet, a run time enforcement of the property by modifying the Android operating system [12], [13]. 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 [3] for a more detailed

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_{ID}	:	Invoke method indicated by m_{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

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 $\rightsquigarrow_{m,\tau} \subseteq \mathrm{State} \times (\mathrm{State} + (\mathcal{V}, \mathbf{heap}))$ where m indicates the method under which the relation is considered and τ indicates whether the instruction is executing normally (indicated by Norm) or throwing an exception. (sometimes we omit m whenever it is clear which m we are referring to, we may also remove τ 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 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.

$\underline{\mathrm{op}}$ 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. $\{\mathrm{np}\}$ 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 $\rightsquigarrow_m^+$ from method invocations of method m and its associated result, to be more precise $\rightsquigarrow_m^+$ is a transitive closure of $\rightsquigarrow_m$. Then, for instructions that may not throw an exception, we remove the subscript $\{m, \mathrm{Norm}\}$ from $\rightsquigarrow$ 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 9 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 $i \mapsto^{\mathrm{np}} t$.
- $P_m[i]$ is an instruction throwing a null pointer exception, and there is no handler for it ($\mathbf{Handler}(i, \mathrm{np}) \uparrow$). In this case it is a return point denoted by $i \mapsto^{\mathrm{np}}$.
- $P_m[i] = \mathbf{throw}$, throwing an exception $C \in \mathbf{classAnalysis}(m, i)$, and $\mathbf{Handler}(i, C) = t$. The successor relation is $i \mapsto^C t$.
- $P_m[i] = \mathbf{throw}$, throwing an exception $C \in \mathbf{classAnalysis}(m, i)$, and $\mathbf{Handler}(i, C) = t$. It is a return point and the successor relation is $i \mapsto^C$.

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

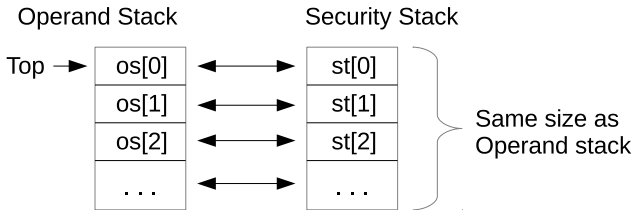
$$\frac{P_m[i] = \textbf{store } x \quad x \in \textbf{dom}(\rho)}{\langle i, \rho, v :: os\rangle \rightsquigarrow \langle i+1, \rho \oplus \{x \mapsto v\}, os\rangle} \qquad \frac{P_m[i] = \textbf{load } x}{\langle i, \rho, os\rangle \rightsquigarrow \langle i+1, \rho, \rho(x) :: os\rangle} \qquad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

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

Fig. 2: JVM Operational Semantic (Selected)

- $P_m[i]$ = **invoke** m_{ID}, throwing an exception $C \in \textbf{excAnalysis}(m_{\text{ID}})$, and $\textbf{Handler}(i, C) = t$. The successor relation is $i \mapsto^C t$.
- $P_m[i]$ = **invoke** m_{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 $k \in \mathcal{S}$, $\textbf{lift}_{\textbf{k}}$ is a point-wise extension to stack types of $\lambda l.k \sqcup l$. The policy of a method is also defined relative to a security level k_{obs} which denotes the capability of an observer to observe values from local variables, fields, and return values whose security level are below k_{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 $L, H \in \mathcal{S}, L \leq k_{\text{obs}}, H \not\leq k_{\text{obs}}$ 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_{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_{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 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 Γ is used to define the table of method signatures which will associate a method identifier m_{ID} and a security level $k \in \mathcal{S}$ (of the object invoked) to a security signature $\Gamma_{m_{\text{ID}}}[k]$. The collection of security signatures of a method m is defined as $\textbf{Policies}_{\boldsymbol{\Gamma}}(m_{\text{ID}}) = \{\Gamma_{m_{\text{ID}}}[k] \mid k \in \mathcal{S}\}$.

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).

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

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

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

Fig. 3: JVM Transfer Rule (Selected)

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* τ *such that* $i \mapsto j$ *and* $i \mapsto^{\tau} k$ *and* $j \neq k$ *(*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* τ*, 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* τ*, 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* τ_1, τ_2 *if* $\textbf{jun}(i, \tau_1)$ *and* $\textbf{jun}(i, \tau_2)$ *are defined and* $\textbf{jun}(i, \tau_1) \neq \textbf{jun}(i, \tau_2)$ *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* τ*, if* $j \in \textbf{region}(i, \tau)$ *and* j *is a return point then for all tags* τ' *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* τ_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 :

Γ:	a table of method signatures, needed to define the transfer rules for method invocation;
ft:	a map from fields to their global policy level;
CDR:	a structure consists of (**region**, **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 10 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* Γ*, a global field policy* **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 β mapping. β 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* $\rho \sim_{k_{\text{obs}}, \vec{k_a}, \beta} \rho'$ *if* ρ *and* ρ' *have the same domain and* $\rho(x) \sim_\beta \rho'(x)$ *for all* $x \in \textbf{dom}(\rho)$ *such that* $\vec{k_a}(x) \leq k_{\text{obs}}$.

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 $o_1 \sim_{k_{\text{obs}},\beta} o_2$) if and only if o_1 and o_2 are objects of the same class and $o_1.f \sim_\beta o_2.f$ for all fields $f \in \mathbf{dom}(o_1)$ s.t. $\mathbf{ft}(f) \leq k_{\text{obs}}$.*

Definition III.6 (Array indistinguishability)**.** *Two arrays $a_1, a_2 \in \mathcal{A}$ are indistinguishable w.r.t. an attacker level k_{obs} and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ (noted by $a_1 \sim_{k_{\text{obs}},\beta} o_2$) if and only if $a_1.\mathbf{length} = a_2.\mathbf{length}$ and, moreover, if $\mathbf{at}(a_1) \leq k_{\text{obs}}$, 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 $h_1 \sim_{k_{\text{obs}},\beta} h_2$, with respect to an attacker level $k_{\text{obs}}$and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ iff:*

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

Definition III.8 (Output indistinguishability)**.** *Given an attacker level k_{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 :*

$$\frac{h_1 \sim_{k_{\text{obs}},\beta} h_2 \qquad \vec{k_r}[n] \leq k_{\text{obs}} \rightarrow v_1 \sim_\beta v_2}{(v_1, h_1) \sim_{k_{\text{obs}},\beta,\vec{k_r}} (v_2, h_2)}$$

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

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

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

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

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

Definition III.9 (Non-interferent JVM method)**.** *A method m is* non-interferent *w.r.t. a policy $\vec{k_a} \rightarrow \vec{k_r}$, if for every attacker level k_{obs}, every partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 \in \mathcal{X} \rightharpoonup \mathcal{V}, h_1, h_2, h_1', h_2' \in \mathit{Heap}, r_1, r_2 \in \mathcal{V} + \mathcal{L}$ s.t.*

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

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

Fig. 4: DEX Instruction List

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

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

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

Definition 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 $k \not\leq \mathbf{ft}(f)$.*

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 $v \in \mathcal{V}, \langle 1, \rho, \epsilon, h \rangle \rightsquigarrow_m^+ v, h'$ implies $h \preceq_{k_h} h'$.*

Definition III.12 (Safe JVM method)**.** *A method m is* safe *w.r.t. a policy $\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}$ if m is side-effect safe w.r.t. k_h and m is non-interferent w.r.t. $\vec{k_a} \rightarrow \vec{k_r}$.*

Definition III.13 (Safe JVM program)**.** *A program is* safe *w.r.t. a table Γ 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 Γ of method signatures. Then P is safe w.r.t. Γ.*

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 ($locR$),
- registers used to store parameters,
- and registers used to simulate the stack.

The registers then can be distinguished between those in $locR$ and those not in $locR$. There is no special name for registers used to store parameters as they are used only once on the entry point of a method. The translation of a JVM method then 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 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 ρ 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 11 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 $rt : (\mathcal{R} \rightharpoonup \mathcal{S})$ instead of stack typing. In the non-optimized translation, register r_0 is used for self reference (**this** in Java) and the return value. With the dual usage of the same register in the local variable side, we find it necessary to extend the typing with a flag to indicate whether r_0 is used as a return value. It is because there will be a possibility to reject a perfectly fine program otherwise (see Example 2). The intuition behind this flag is that register r_0 is only ever updated when the function is about to return. So in this case, the flag itself is only used for sanity checking of a DEX program.

DEX itself does not really distinguish between registers used for local variables and stacks. Therefore, we think it will be convenient to define a function $sec : \mathcal{R} \to \mathcal{S}$ which is defined as the following :

$$sec(r) = \begin{cases} \vec{k_a}(r) & \text{if} \quad r \in locR \\ rt(r) & \text{if} \quad r \notin locR \end{cases}$$

To be more precise, for each transfer rule that stores something in the target register r, there will be two types of instances, one is when $r \in locR$ and one is when $r \notin locR$. The main difference between the two is that when $r \in locR$ then we have to take into account the method signature that governs the security level of values that can be put into the local variable. In this case, there is an additional constraint that the security level added to the mapping r has to be lower than that of $\vec{k_a}(r)$. We also define the lift function to affect only those registers used to simulate the stack ($\forall r \notin locR$).

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 $i \vdash RT_i \Rightarrow rt, rt \sqsubseteq RT_j$, and also the additional flag to indicate that r_0 has been modified. 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 Γ, a global field policy* **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 $\mathbf{RT} : \mathcal{PP} \to (\mathcal{R} \rightharpoonup \mathcal{S})$ and $\mathbf{Fl} : \mathcal{PP} \to \{0, 1\}$ s.t. $RT_1 = \vec{k_a}, Fl_1 = 0$ and for all $i, j \in \mathcal{PP}, e \in \{\text{Norm} + \mathcal{C}\}$:*

- *$i \mapsto^e j$ implies there exists $rt \in (\mathcal{R} \rightharpoonup \mathcal{S})$ and $f \in \{0, 1\}$ such that Γ,* **ft***,* **region***, se, sgn, $i \vdash^e \langle Fl_i, RT_i\rangle \Rightarrow \langle f, rt\rangle$ and $rt \sqsubseteq RT_j, f = Fl_j$;*
- *$i \mapsto^e$ implies $\Gamma, \mathbf{ft}, \mathbf{region}, se, sgn, i \vdash^e \langle Fl_i, S_i\rangle \Rightarrow$*

where $rt \sqsubseteq rt'$ is taken to mean that for all $x \in \mathbf{dom}(rt)$, whenever $x \notin locR$, we have $rt(x) \leq rt'(x)$ or $x \notin \mathbf{dom}(rt')$.

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

Definition IV.2 (Local variable indistinguishability)**.** *For $\rho, \rho' : (\mathcal{R} \rightharpoonup \mathcal{V})$, we have $\rho \sim_{k_{\text{obs}}, \vec{k_a}, \beta} \rho'$ if ρ and ρ' have the same domain in $locR$ and $\rho(x) \sim_\beta \rho'(x)$ for all $x \in locR$ such that $\vec{k_a}(x) \leq k_{\text{obs}}$.*

Definition IV.3 (Stack registers indistinguishability)**.** *For $\rho, \rho' : (\mathcal{R} \rightharpoonup \mathcal{V})$ and $rt, rt' : (\mathcal{R} \rightharpoonup \mathcal{S})$, we have $\rho \sim_{k_{\text{obs}}, rt, rt', \beta}$*

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

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

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

Fig. 5: DEX Operational Semantic (Selected)

$$\frac{P_m[i] = \mathbf{const}(r, v) \quad r \notin locR}{i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i)\}\rangle} \qquad \frac{P[i] = \mathbf{move}(r, r_s) \quad r \notin locR \quad r_s \neq r_0}{i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(sec(r_s) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{move}(r, r_s) \quad r \in locR \backslash \{r_0\} \quad r_s \neq r_0 \quad sec(r_s) \sqcup se(i) \leq \vec{k_a}(r)}{i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(sec(r_s) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{return}(r_s) \quad se(i) \sqcup rt(r_s) \leq \vec{k_r}[n]}{i \vdash \langle f, rt\rangle \Rightarrow} \qquad \frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b) \quad r \notin locR \quad r_a \neq r_0 \ \mathbf{and}\ r_b \neq r_0}{i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(sec(r_a) \sqcup sec(r_b) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{goto}(j)}{i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt\rangle} \qquad \frac{P_m[i] = \mathbf{ifeq}(r, t) \quad r \neq r_0 \quad \forall_{j'} \in \mathbf{region}(i, \mathrm{Norm}), se(i) \sqcup sec(r) \leq se(j')}{i \vdash \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec(r)}}(rt)\rangle}$$

Fig. 6: DEX Transfer Rule (Selected)

ρ' inductively defined by the following clauses:

$$\frac{\mathbf{high}(\rho, rt) \qquad \mathbf{high}(\rho', rt')}{\rho \sim_{k_{\mathrm{obs}}, rt, rt', \beta} \rho'}$$

$$\frac{\rho \sim_{k_{\mathrm{obs}}, rt, rt', \beta} \rho' \qquad v \sim v' \qquad k \leq k_{\mathrm{obs}}}{\rho \oplus \{x \mapsto v\} \sim_{k_{\mathrm{obs}}, rt \oplus \{x \mapsto k\}, rt' \oplus \{x \mapsto k\}, \beta} \rho' \oplus \{x \mapsto v\}}$$

$$\frac{\rho \sim_{k_{\mathrm{obs}}, rt, rt', \beta} \rho' \qquad k \not\leq k_{\mathrm{obs}} \qquad k'(x) \not\leq k_{\mathrm{obs}}}{\rho \oplus \{x \mapsto v\} \sim_{k_{\mathrm{obs}}, rt \oplus \{x \mapsto k\}, rt' \oplus \{x \mapsto k'\}, \beta} \rho' \oplus \{x \mapsto v'\}}$$

where $x \notin locR$, $v \in \mathcal{V}$, and $k, k' \in \mathcal{S}$.

Definition IV.4 (Registers indistinguishability)**.** *For $\rho, \rho' : (\mathcal{R} \rightharpoonup \mathcal{V})$ and $rt, rt' : (\mathcal{R} \rightharpoonup \mathcal{S})$, we have $\rho \sim_{k_{\mathrm{obs}}, \vec{k_a}, rt, rt', \beta} \rho'$ iff $\rho \sim_{k_{\mathrm{obs}}, \vec{k_a}, \beta} \rho'$ and $\rho \sim_{k_{\mathrm{obs}}, rt, rt', \beta} \rho'$.*

Definition IV.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 IV.6 (Array indistinguishability)**.** *Two arrays $a_1, a_2 \in \mathcal{A}$ are indistinguishable w.r.t. an attacker level k_{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 IV.7 (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 :*

- *β 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.8 (Output indistinguishability)**.** *Given an attacker level k_{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 :*

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

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

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

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

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

where $\rightarrow$ indicates logical implication.

Definition IV.9 (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_{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.*

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

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

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

Definition IV.10 (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 $k \not\leq \mathbf{ft}(f)$.*

Definition IV.11 (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 $v \in \mathcal{V}, \langle 1, \rho, h \rangle \rightsquigarrow_m^+ v, h'$ implies $h \preceq_{k_h} h'$.*

Definition IV.12 (Safe DEX method)**.** *A method m is* safe *w.r.t. a policy $\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}$ if m is side-effect safe w.r.t. k_h and m is non-interferent w.r.t. $\vec{k_a} \rightarrow \vec{k_r}$.*

Definition IV.13 (Safe DEX program)**.** *A program is* safe *w.r.t. a table Γ 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 Γ of method signatures. Then P is safe w.r.t. Γ.*

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 first **store** 2 constraint. The reasoning is that $se(i)$ for **push** 0 will be H, thus the constraint will be $H \sqcup H \leq L$ 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 $H \sqcup H \leq L$ which also can not be satisfied, thus this program will be rejected by the DEX type system.

The next example shows the need to include a flag in the DEX type system.

Example V.2. Assume that $\vec{k_a}(r_0) = L, \vec{k_a}(r_1) = L$ (which means local variable 1 is low) $\vec{k_r}[n] = L$, and the se at the current program point is L. r_2 is the start of the stack space.

...	...
load 1	**move**$(r_2, r1)$
return	**move**(r_0, r_2)
	return(r_0)

The above JVM bytecode and its translation will be accepted in the type system for the JVM bytecode as the constraint $L \sqcup L \leq L$ can be satisfied. But in the DEX type system, the last **move**(r_0, r_2) instruction will have a constraint because the target of the instruction is r_0 which is part of the local variables, thus will be rejected by the type system because it can not reconcile the constraint $L \sqcup H \leq L$. Therefore in the DEX type system we have a flag to allow this move to r_0 once to store the return value.

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.3. 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 $\mathbf{putfield}\ f$ at the last line there is a constraint with the security level of the object. In this case, the $\mathbf{load}\ 1$ instruction will push a reference of the object with high security level, therefore, the constraint that $L \sqcup H \sqcup L \leq L$ can not be satisfied. The same goes for the DEX type system, it will also reject the translated program. The reasoning is that the $\mathbf{move}(r_4, r_1)$ 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 $\mathbf{iput}(r_5, r_4, f)$ we won't be able to satisfy $L \sqcup H \sqcup L \leq L$.

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

Example V.4. Assume that $\vec{k_a} = \{r_1 \mapsto H, r_2 \mapsto L, r_3 \mapsto H\}$. $\mathbf{Handler}(l_2, \mathrm{np}) = l_h$, and for any $e, \mathbf{Handler}(l_2, e) \uparrow$. The following JVM bytecode and its translation will be rejected by the typing system for the JVM bytecode.

	...		...
	$\mathbf{load}\ 1$		$\mathbf{move}(r_4, r_1)$
$l_1:$	$\mathbf{ifeq}\ l_2$	$l_1:$	$\mathbf{ifeq}(r_4, l_2)$
	$\mathbf{new}\ C$		$\mathbf{new}(r_4, C)$
	$\mathbf{store}\ 3$		$\mathbf{move}(r_3, r_4)$
	$\mathbf{load}\ 3$		$\mathbf{move}(r_4, r_3)$
$l_2:$	$\mathbf{invokevirtual}\ m$	$l_2:$	$\mathbf{invoke}(1, m, r_4)$
	$\mathbf{push}\ 0$		$\mathbf{const}(r_4, 0)$
	$\mathbf{store}\ 2$		$\mathbf{move}(r_2, r_4)$
$l_h:$	$\mathbf{push}\ 1$	$l_h:$	$\mathbf{const}(r_4, r_1)$
	$\mathbf{store}\ 2$		$\mathbf{move}(r_2, l_4)$
	...		...

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 $\mathbf{store}\ 2$ (line 8 and 10) the constraint is violated since we have $\vec{k_a}(r_2) = L$, 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 $H \leq L$ 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 [18] 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 $[\![P]\!]$, where*

$$[\![P]\!] = \mathbf{Translate}(\mathbf{TraceParentChild}(\mathbf{StartBlock}(P))).$$

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

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

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

$$[\![P]\!] = \lceil\!\lceil \lfloor\!\lfloor P \rfloor\!\rfloor \rceil\!\rceil.$$

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 $\mathbf{region}(i,\tau)$ and $P[i]$ is a branching instruction, let i_b be the program point in $\lfloor\!\lfloor i \rfloor\!\rfloor$ such that $P_{\mathrm{DEX}}[i_b]$ is a branching instruction, then*

$$\lfloor\!\lfloor \mathbf{region}(i,\tau) \rfloor\!\rfloor = \mathbf{region}(i_b, \lfloor\!\lfloor \tau \rfloor\!\rfloor) = \bigcup_{j \in \mathbf{region}(i,\tau)} \lfloor\!\lfloor j \rfloor\!\rfloor$$
and
$$[\![\mathbf{region}(i,\tau)]\!] = \mathbf{region}(i_b, [\![\tau]\!]) = \bigcup_{j \in \mathbf{region}(i,\tau)} [\![j]\!]$$

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

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

$$\mathbf{handler}(i_e,\tau) \in \mathbf{region}(i_e,\tau) \text{ in } \lfloor\!\lfloor P \rfloor\!\rfloor$$
and
$$\mathbf{handler}(\lceil\!\lceil i_e \rceil\!\rceil,\tau) \in \mathbf{region}(\lceil\!\lceil i_e \rceil\!\rceil,\tau) \text{ in } [\![P]\!]$$

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

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

$$\begin{aligned}\forall b \in \lfloor\!\lfloor P \rfloor\!\rfloor. \quad & P_{DEX}[\lceil\!\lceil b.\mathbf{lastAddress} \rceil\!\rceil + 1] = \mathbf{goto} \\ & \rightarrow \big(\forall .i \in \mathcal{PP}_{DEX}.b.\mathbf{lastAddress} \in \mathbf{region}(i,\tau) \\ & \rightarrow (\lceil\!\lceil b.\mathbf{lastAddress} \rceil\!\rceil + 1) \in \mathbf{region}(i,\tau)\big)\end{aligned}$$

where $\rightarrow$ indicates logical implication.

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

$$\lfloor\!\lfloor j \rfloor\!\rfloor[0] = \mathbf{jun}(\lfloor\!\lfloor i \rfloor\!\rfloor[i_b],\tau) \text{ in } \lfloor\!\lfloor P \rfloor\!\rfloor$$
and
$$[\![\lfloor\!\lfloor j \rfloor\!\rfloor[0]]\!] = \mathbf{jun}([\![\lfloor\!\lfloor i \rfloor\!\rfloor[i_b]]\!],\tau) \text{ in } [\![P]\!].$$

Definition VII.7 (Security Environment Translation and Compilation)**.** $\forall i \in \mathcal{PP}, j \in \lfloor\!\lfloor i \rfloor\!\rfloor.se(j) = se(i)$ *in* $\lfloor\!\lfloor P \rfloor\!\rfloor$ *and* $\forall i \in \mathcal{PP}, j \in \lfloor\!\lfloor i \rfloor\!\rfloor.se(\lceil\!\lceil j \rceil\!\rceil) = se(i)$ *in* $[\![P]\!]$.

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 $\lfloor\!\lfloor . \rfloor\!\rfloor$ and $[\![.]\!]$ 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). More formally, if there are n local variables denoted by $v_1, \ldots, v_n$ and stack type with the height of m (0 denotes the top of the stack), then $[\![st]\!] = \{r_0 \mapsto \vec{k_a}(v_1), \ldots, r_{n-1} \mapsto \vec{k_a}(v_n), r_n \mapsto st[m-1], \ldots, r_{n+m-1} \mapsto st[0]\}$. Lastly, the function $\lceil\!\lceil . \rceil\!\rceil$ is also overloaded for addressing (bi, i) to denote abstract address in the DEX side which will actually be instantiated when producing the output DEX program from the blocks.

Definition VII.8 ($\lfloor\!\lfloor \mathbf{excAnalysis} \rfloor\!\rfloor$ and $[\![\mathbf{excAnalysis}]\!]$)**.**

$$\forall m \in \mathcal{M}.\lfloor\!\lfloor \mathbf{excAnalysis}(m) \rfloor\!\rfloor = \mathbf{excAnalysis}(m) \text{ in } \lfloor\!\lfloor P \rfloor\!\rfloor$$
and
$$\forall m \in \mathcal{M}.[\![\mathbf{excAnalysis}(m)]\!] = \mathbf{excAnalysis}(m) \text{ in } [\![P]\!].$$

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

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

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

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

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.2, Lemma VII.3 and Lemma VII.4 to prove the typability of register typing. Then, using Lemma VII.6 and Lemma VII.7 we can establish the typability of all sequential instructions concerning the flag because we know that the instructions will always preserve the flag which will be 0.

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

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

according to the typing rule(s) of $[\![Ins]\!]$.

Lemma VII.3. *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 throws exception τ be denoted by n. Let $(be, 0) = [\![i_e]\!]$ 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*

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

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

Lemma VII.4. *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 $[\![Ins]\!]$. Let $RT_{[\![i]\!][0]} = [\![S_i]\!]$, let $RT_{[\![j]\!][0]} = [\![S_j]\!]$ and rt be registers typing obtained from the transfer rules involved in $[\![Ins]\!]$. Then $rt \sqsubseteq RT_{[\![j]\!][0]}$.*

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

Lemma VII.6 (Sequential Instructions Maintain Flag)**.** *Instructions within a block (sequential instructions) do not modify the flag, i.e.*

$$\forall bi, \{i, j\} \in bi \textit{ s.t. } (bi, i) \mapsto (bi, j), \exists rt, rt'.(bi, i) \vdash \langle F_{(bi,i)}, rt\rangle \Rightarrow \langle F_{(bi,i)}, rt'\rangle.$$

Lemma VII.7 (Block Starting Flag)**.** *All blocks (except the return block) will have flag 0 at its starting instruction, i.e.$\forall bi \backslash \{Ret\}.F_{(bi,0)} = 0$ where Ret indicates return block.*

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

Lemma VII.8. *Let P be a Java program such that*

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

Then $[\![P]\!]$ will satisfy

1) *for all blocks bi, bj s.t. $bi \mapsto bj$, $\exists rt_b$. s.t. $RTs_{bi} \Rightarrow^* rt_b, rt_b \sqsubseteq RTs_{bj}$; and*
2) *$\forall bi, i, j \in bi$. s.t. $(bi, i) \mapsto (bi, j).\exists rt$. s.t. $(bi, i) \vdash RT_{(bi,i)} \Rightarrow rt, rt \sqsubseteq RT_{(bi,j)}$*

where

$$\begin{array}{llll} RTs_{bi} = [\![S_i]\!] & \textit{with} & [\![i]\!] = (bi, 0) \\ RTs_{bj} = [\![S_j]\!] & \textit{with} & [\![j]\!] = (bj, 0), \\ RT_{(bi,i)} = [\![S_{i'}]\!] & \textit{with} & [\![i']\!] = (bi, i) \\ RT_{(bi,j)} = [\![S_{j'}]\!] & \textit{with} & [\![j']\!] = (bj, j). \end{array}$$

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.9. *Let $[\![P]\!]$ be typable basic blocks resulting from translation of JVM instructions still in the block form, i.e.*

$$[\![P]\!] = \mathbf{Translate}(\mathbf{TraceParentChild}(\mathbf{StartBlock}(P))).$$

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.8 and Lemma VII.9. 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 (*$\mathbf{region}, \mathbf{jun}$*), PA-Analysis (*$\mathbf{classAnalysis}$ *and* $\mathbf{excAnalysis}$*), and method policies Γ, then $[\![P]\!]$ according to the translation scheme has the property that*

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

according to a safe CDR ($[\![\mathbf{region}]\!], [\![\mathbf{jun}]\!]$), $[\![PA - Analysis]\!]$, and $[\![\Gamma]\!]$.

VIII. Implementation

A. Overall Architecture

Figure 7 shows the overall architecture of our work. There are a lot of components already fixed in place when we are talking about the compilation from Java source to Android program. Nevertheless we did not include Java source code itself as one of the component as we are more concerned about the Java classes compiled from Java source code (JVM Bytecode). In this setting, our contributions are highlighted in grey.

As we have mentioned before, the whole Android compilation setting starts from a developer writing the application in Java. The components of the applications will then get compiled to Java classes. DX tool which is bundled with the Android SDK will then aggregate these classes into one (possibly many) DEX file, usually named "classes.dex". The rest of the toolchain will also bundle the manifest file, other resources, and assets into an APK. This APK is what we call

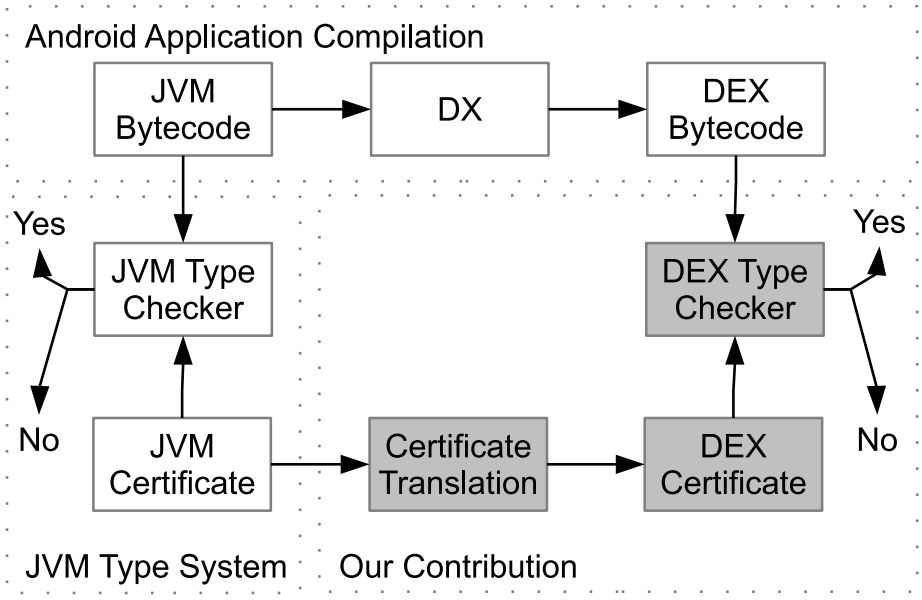


Fig. 7: Overall architecture

an Android application, which can be installed into a mobile phone running Android OS.

The work by Barthe et. al. comes into picture in providing a type system to guarantee that a JVM bytecode is non-interferent. Crucial in the work is the component that check whether a given JVM bytecode is typable, and produce a certificate if it is (type inference). There is then another component which type check whether a JVM bytecode match its certificate. Since our aim is the translation of certificate, we do not include the type inference component in the figure. Nevertheless, this component is crucial in providing us with the certificate for JVM bytecode which is necessary for the whole system to even start. We will detail this component in the next section.

Our contributions are mainly contained in two components. The first component, certificate translation, basically takes the certificates for the source Java classes and translate it into a certificate for the corresponding DEX file, independently of the non-optimizing compilation done by the DX tool. The other component parse the DEX file, take the certificate, and check whether they match. We will detail this component in the next section as well.

B. Component Details

In this section, we will go into more depth on each of the components that provides us with the setting of this experimentation.

1) Certificate Structure: Here we will describe how we structure our certificate, both for JVM and DEX. They mainly differ in that JVM uses stack type while DEX uses registers type. Figure 8 shows high level structure of the certificate.

Below, we describe how we represent the certificate as a file. Since we will infer the type for JVM, we do not provide the instructions typing for JVM certificate, just instructions typing for DEX. We used the notation un to denote n bytes of data for each type. We first describe how we represent a string and extended security level, then we describe the certificate and its details.

```
string {
        u1      string_size;
        u1      char[string_size];
}
```

[**string_size**:] the value of this item is equal to the number of characters in the string. The index to char is valid if it is greater than zero and less than string_size. The string does not have to be ended by '\0'.

[**char**:] this item contains a list of character in the order of its appearance in the string. Each character is represented by its ASCII which is in the range of 0-255.

```
security_level {
        u1      level_depth;
        u1      security_level_ID[level_depth];
}
```

[**level_depth**:] the value of this item is equal to the depth of the extended security level. The depth is valid if it is greater than zero, because zero is reserved for a security level of $\perp$ which should never exists in a valid certificate except for pseudo-register ret.

[**security_level_ID**:] this item constitutes a representation of an extended level, in the form of bytes in the order they appear in the security level. A security level with only 1 depth denotes a simple security level (is an element of $\mathcal{S}$) whereas depth more than 1 represent extended security level ($\mathcal{S}^{\text{ext}}$). A depth of more than one will have the following form : depth of 2

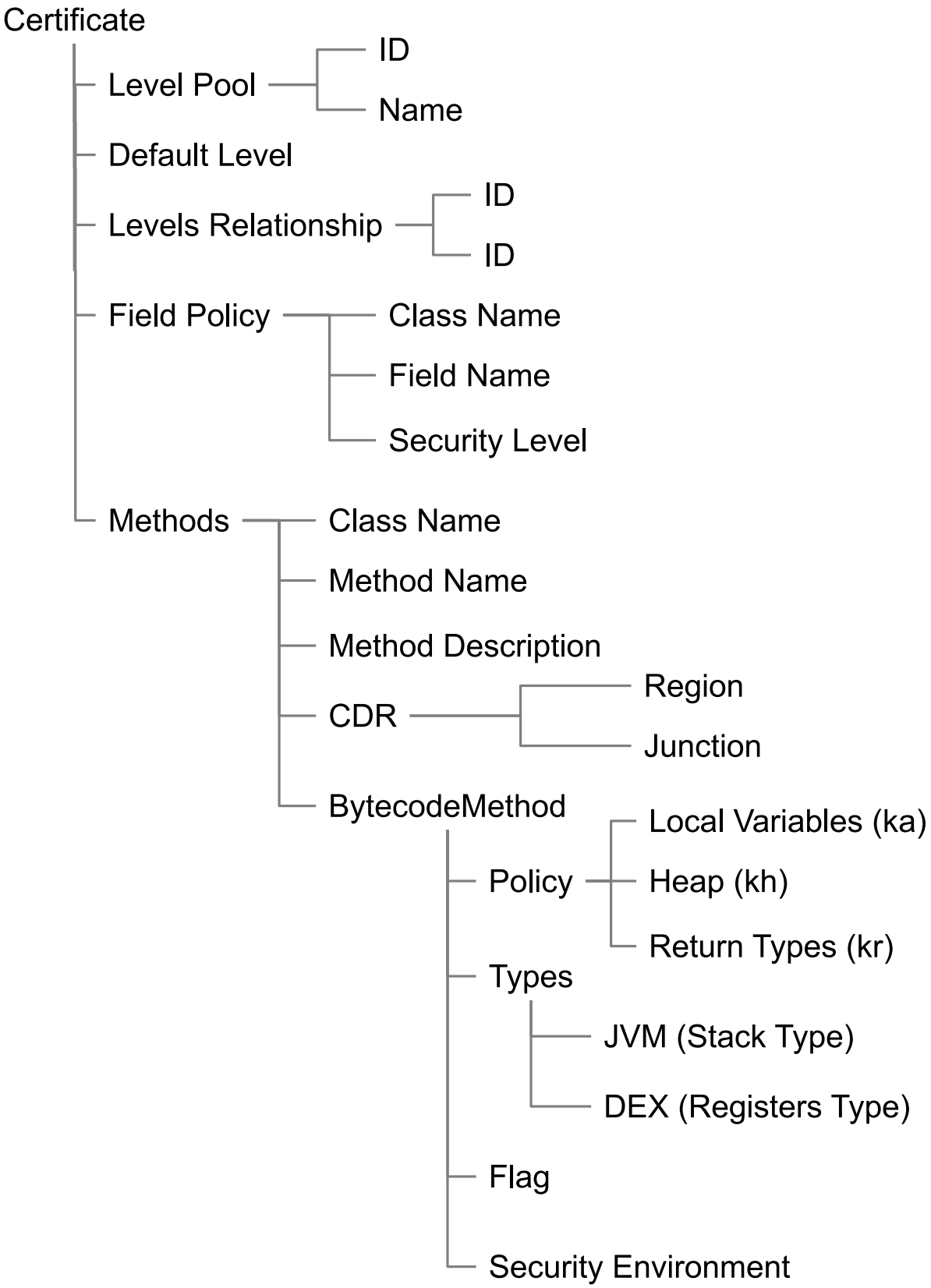


Fig. 8: Certificate Structure

will be $b_1[b_2]$, depth of 3 will be $b_1[b_2[b_3]]$, and so on, where b_n indicates the n-th byte.

```
Certificate {
    u1              level_pool_size;
    string          level_pool[level_pool_size];
    u1              default_level;
    u1              lv_rel_size; //levels_relationship_size
    u1*u1           levels_relationship[lv_rel_size];
    u2              field_size;
    field_info      fields[field_size];
    u2              methods_size;
    method_info     methods[method_size];
}
```

[**level_pool_size**:] the value of this item is equal to the number of entries in the level_pool. The index to level_pool is valid if it is greater or equal than zero and less than level_pool_size

[**level_pool**:] this item contains a list of the name of the security level. The security level itself will be represented as a pair of ID and its name. The list of name here is expected to follow the order of the ID of security level, i.e. the first item in the list will have ID 0, the second item will have ID 1, and so on.

[**default_level**:] the value of this item will denote the lowest level in the level_pool.

[**levels_relationship_size**:] the value of this item is equal to the number of entries in the levels_relationship.

[**levels_relationship**:] this item contains a list of a pair of ID x, y to denote that $x \leq y$. The value in this item do not have to be sorted, because they are represented as a relation map.

[**field_size**:] the value of this item is equal to the number of entries in the fields.

[**fields**:] this item contains the global security policy for the field (ft). Each field is represented as a string of class name, a string of field name, and a security_level_info of security level. The list of this fields does not have to be sorted according to some order, because we represent the policy as a map from a pair of string to its security level.

[**methods_size**:] the value of this item is equal to the number of entries in the methods.

[**methods**:] this item contains methods (the structure of each method is described in method_info), where each item is a string of class name, a string of method name, a string of method description, and the content of the method. The entries in this item do not have to be sorted as they will be represented as a map from a triple of string to a method_info structure

```
field_info {
    string          class_name;
    string          field_name;
    security_level  field_policy;
}
```

This entry basically says that the global policy (ft) for this particular field identified with class_name and field_name has the security level of field_policy. The structure of the field is the following:

[**class_name**:] the value of this item identifies the field with the class name. There are no rules regarding what should be put here. what we need to be concerned with is that if the class name does not match with the one in the original program, then the the program and its certificate will not type check.

[**field_name**:] the value of this item identifies the field with its particular name. There are no rules regarding what should be put here. what we need to be concerned with is that if the field name does not match with the one in the original program, then the the program and its certificate will not type check.

[**field_policy**:] this item will have the structure of security_level and will indicate the extended security level assigned to this particular field.

```
method_info {
    string          class_name;
    string          method_name;
    string          method_description;
    u2              instruction_size;
    CDR_info        CDR[instruction_size];
    bm_info         bytecodeMethod[level_pool_size];
}
```

This entry basically says for this particular method identified with class_name, method_name, and method_description has the CDR structure of CDR, and policy and types of bytecodeMethod for each level in the level_pool. The structure of the method is the following:

[**class_name**:] the value of this item identifies the field with the class name. There are no rules regarding what should be put here. what we need to be concerned with is that if the class name does not match with the one in the original program, then the the program and its certificate will not type check.

[**method_name**:] the value of this item identifies the method with its particular name. There are no rules regarding what should be put here. what we need to be concerned with is that if the method name does not match with the one in the original program, then the the program and its certificate will not type check.

[**method_description**:] the value of this item identifies the method with its particular name. There are no rules regarding what should be put here. what we need to be concerned with is that if the method description does not match with the one in the original program, then the the program and its certificate will not type check.

[**instruction_size**:] the value of this item identifies how many instructions are there in the method, mainly to identify how many CDRs we are expecting. A note need to be made that this instruction_size can be different from the one in the bytecodeMethod to cater for default methods that do not need a detailed instructions typing.

[**CDR**:] Each element this item will have the structure of CDR_info, which will indicate the labels, regions and junctions

for this method. The elements of this CDR does not have to be ordered according to the labels, because we represent the CDR information as a mapping from program points to regions or junction.

[**bytecodeMethod**:] this item will contain a number of policies and typings according to the number of security levels there are in the level pool. The main purpose of this different bytecodeMethod is to cater for the Γ policy which can be different according to the security level of the object that the method is invoked from. In order for the program to type check, all the methods (and all security level of the object) need to be typechecked.

```
CDR_info {
    u2      program_label
    u2      region_size;
    u2      region[region_size];
    u2      junction;
}
```

[**program_label**:] the value of this item indicates the label of the program for which has the region information and junction.

[**region_size**:] the value of this item is equal the number of entries in region. A valid value for this region is greater or equal to zero and less than instruction_size.

[**region**:] the value of this item shows which program labels are under the region of program_label, it may be empty.

[**junction**:] the value of this item shows which program point serves as the junction of program_label. This item has the value of zero if there is no such junction for program_label.

```
bm_info { // bytecodeMethod_info
    u1              object_level;
    u2              bm_instruction_size;
    bm_insn_info    instructions[bm_instruction_size];
    u1              ka_size;
    security_level  ka[ka_size];
    u1              kh;
    u1              kr_size;
    security_level  kr[kr_size];
}
```

[**object_level**:] the value of this item indicates the security of the object level that invokes this particular method.

[**bm_instruction_size**:] the value of this item indicates the number of instructions in the method. This value will be either 0 or the same as instruction_size in the method structure.

[**instructions**:] this item describes the registers type and security environment for each instruction in the method.

[**ka_size**:] the value of this item denotes how many locals variable are there. Even though there is no such thing as local variable in the DEX itself, this value is carried over from the JVM certificate where there is a concept of local variable.

[**ka**:] this item will denote the security level for each of the local variables. The order in this item matters, i.e. the first security_level denotes the security level of $\vec{k_a}[0]$, the second security_level denotes the security level of $\vec{k_a}[1]$, and so on.

[**kh**:] the value of this item refers to the level_pool, which will denote the method policy for heap.

[**kr_size**:] the value of this item denotes how many return types are there. On a normal method without exception returning a value, the value will be 1. The value of kr_size is the 1 + the number of exceptions possibly thrown by the method. At the current implementation, we have not implemented exception handling yet, so the value will either be 0 or 1.

[**kr**:] this item will denote the security level for each of the return type. We reserve kr(0) to be the security level of the normal return value.

```
bm_insn_info { // bm_instruction_info
    u2                  program_label;
    u2                  register_type_size;
    u2*security_level   register_level[register_type_size];
    security_level      ret;
    u1                  flag;
    u1                  security_environment;
}
```

[**program_label**:] the value of this item indicates the program label which will be typed by the register type and the security environment.

[**register_type_size**:] the value of this item indicates how many registers are there in the register type for the current program label.

[**register_level**:] this item describes the security level of the value hold by a particular register. This item is represented as a pair of register number and its security level, and we represent them directly as a map.

[**ret**:] the value of this item will indicate what is the security level of the pseudo-register ret. In the case where it is a $\perp$, we can say that ret does not hold any value

[**flag**:] the value of this item will indicate the flag that is associated with the instruction at program_label.

[**security_environment**:] the value of this item will refer to a basic security level contained in the level_pool. The valid value will be greater or equal than 0 and less than level_pool_size.

2) Naive JVM Type Inference: This component takes as an input a file which contains the certificate without the typing for each instructions (stacktype and security environment) then reconstruct the certificate for the JVM bytecode. The inference itself (described in Algorithm 3 is simple in nature, we just repeatedly infer the stack type and security environment for the successor instruction, starting from label 0, until they converge (no more change either in stack type or security environment). Algorithm 2, describes how we implement the naive type inference. A little note to be made is that we abstract from actual program counter and use the index to the list instead. We also overload the symbol $\sqsubseteq$ to be operable between stack types as well, where the it means pairwise lub between the stacktype elements. Before we invoke the Algorithm 2, we first do a simple program flow tracing described in Algorithm 1.

Algorithm 1 Flow_Tracer(bm)

$order$:= [1]; // always start with the first instruction
$workingSet$:= [1];
while $workingSet$ is not empty **do**
  $(i :: workingSet') := workingSet$;
  if $bm.instructionAt(i) \in$ {Nop, Push c, Load x, Store x, Binop, New, Getfield, Putfield, Invoke m'} **then**
    $order.append(i+1)$;
    if $(i+1) \notin order$ **then** $workingSet'.append(i+1)$;
  else if $bm.instructionAt(i)$ = Goto t **then**
    $order.append(t)$;
    if $t \notin order$ **then** $workingSet'.append(t)$;
  else if $bm.instructionAt(i) \in$ {If t, Ifz t} **then**
    $order.append(i+1); order.append(t)$
    if $(i+1) \notin order$ **then** $workingSet'.append(i+1)$;
    if $t \notin order$ **then** $workingSet'.append(t)$;
  $workingSet := workingSet'$;
return order;

The purpose of this program flow tracing is to ease the burden of type inference so that it just need to traverse the order produced by the flow tracer without having to deal with the case where an instruction still has not been assigned a stack type and security environment yet.

We implemented this component in OCaml, in conjunction with the component to do a certificate translation. Our main consideration for doing so is because it is much easier to transfer the resulting JVM certificate directly to the next component within the program itself, rather than having a temporary output file.

3) Non-Optimizing Certificate Translation: The translation component basically implements what we have highlighted before in the translation section. We implemented the five steps in translating the JVM bytecode into DEX bytecode while carrying over the type from JVM for each instructions. The translation of the stack type also follows in that we just flatten the stack type and assign an index (starting from the number of local variables) from the bottom of the stack. The details fo the implementation itself is somehow straightforward if it just concerns the translation of the bytecode. When we translate the type, however, some complications arise.

Firstly, since the translation steps only concerns a specific bytecode under a specific policy, we have to extend that to include each security level for the object that contains this method. We then have to incorporate all the methods in the class. Since DX aggregates all the files together, we also have to extend the scope to include all the classes to be compiled to DEX bytecode. These peripheral informations are not necessarily difficult in themselves because they are mostly straightforward. For example, the policy for the field and methods can be directly transferred from JVM to DEX.

That said, the convention for the class identifier is different between JVM and DEX, so we need a bit of adjustment here. In particular, for JVM the class identifier is a "/" delimited string where the last string is the particular class name, and the strings before that is the package name. For DEX, however, the class identifier takes the form of "L" + class identifier + ";", e.g. if a class in JVM is identified with "java/lang/Object" then in DEX it is identified as "Ljava/lang/Object;". For the current implementation we also use the short method descriptor in DEX as opposed to the full method descriptor.

Algorithm 2 Type_Inference $(bm, bmc, mc, lvt, ft, order)$

$bmc.ST(1)$ = []; // Initial stack type is always empty
$bmc.ST(1) = mc.default_level$; // Initial se is always Low
for i = 1 to bm.length **do**
  $idx := order(i)$;
  switch $(bm.instructionAt(idx))$
  case (Nop)**:**
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup bmc.ST(i)$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
  case (Push c)**:**
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup (bmc.se(i) :: bmc.ST(i))$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
  case (Load x)**:**
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup ((bmc.se(i) \sqcup lvt.ka(x)) :: bmc.ST(i))$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
  case (Store x)**:**
    $(k :: st) := bmc.ST(i)$;
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup st$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
    if $bmc.se(i) \sqcup k \not\leq lvt.ka(x)$ **then return** false;
  case (Binop)**:**
    $(k_1 :: k_2 :: st) := bmc.ST(i)$;
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup ((k_1 \sqcup k_2 \sqcup bmc.se(i)) :: st)$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
  case (Goto t)**:**
    $bmc.ST(t) := bmc.ST(t) \sqcup bmc.ST(i)$;
    $bmc.se(t) := bmc.se(t) \sqcup bmc.se(i)$;
  case (If t)**:**
    $(k_a :: k_b :: st) = bmc.ST(i)$;
    $bmc.ST(idx{+}1) := bmc.ST(idx{+}1) \sqcup lift(st, k_a \sqcup k_b)$;
    $bmc.ST(t) := bmc.ST(t) \sqcup lift(st, k_a \sqcup k_b)$;
    for j in bmc.region(i) **do**
      $bmc.se(j) = bmc.se(j) \sqcup k_a \sqcup k_b$;
  case (Ifz t)**:**
    $(k :: st) = bmc.ST(i)$;
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup lift(st, k)$;
    $bmc.ST(t) := bmc.ST(t) \sqcup lift(st, k)$;
    for j in bmc.region(i) **do**
      $bmc.se(j) = bmc.se(j) \sqcup k$;
  case (Return)**: continue**;
  case (IReturn)**:**
    $(k :: st) := bmc.ST(i)$;
    if $bmc.se(i) \sqcup k \not\leq lvt.kr(0)$ **then return** false;
  case (New c)**:**
    $bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup (bmc.se(i) :: bmc.ST(i))$;
    $bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;

Secondly, there are some instructions that do not fit nicely with the framework of just bringing the types from the JVM. The instruction pair of Invoke and MoveResult in DEX use

case (Getfield f)**:**
$(k_o :: st) = bmc.ST(i)$;
$bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup ((bmc.se(i) \sqcup k_o \sqcup ft(f)) :: bmc.ST(i))$;
$bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
case (Putfield (r, r_o, f))**:**
$(k_s :: k_o :: st) = bmc.ST(i)$;
$bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup st$;
$bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
if $bmc.se(i) \sqcup k_s \sqcup k_o \not\leq ft(f)$ **then return** false;
if $k_h \not\leq ft(f)$ **then return** false;
case (Invoke m')**:**
$length(st_1) := nbArguments(m')$;
$(st_1 :: k :: st_2) := bmc.ST(i)$;
$lvt' := mc.find(m', k).policy$;
$bmc.ST(idx+1) := bmc.ST(idx+1) \sqcup ((lvt'.kr(0) \sqcup bmc.se(i) \sqcup k) :: st_2)$;
$bmc.se(idx+1) := bmc.se(idx+1) \sqcup bmc.se(i)$;
if $k \sqcup lvt.kh \sqcup bmc.se(i) \not\leq lvt'.kh$ **then return** false
if $k \not\leq lvt'.ka(0)$ **then return** false
for $j := 1$ to $length(st_1)$ **do**
if $st_1[j] \not\leq lvt'.ka[j]$) **then return** false
end for
return bmc;

Algorithm 3 Infer_Type(bm, bmc, mc, lvt, ft)

$order$:= Flow_Tracer(bm);
new_bmc := Type_Inference(bm, bmc, mc, lvt, ft);
while bmc <> new_bmc **do**
$bmc := new\_bmc$;
new_bmc := Type_Inference(bm, bmc, mc, lvt, ft);
return new_bmc;

the pseudo-register ret, which obviously is not captured in JVM. For MoveResult, firstly we just take the stack type of the successor of translate it into the registers type. Then we need to check whether the Invoke instruction immediately before is returning a value or not, and if so we remove one value from the top of the stack and put it into ret. Goto instruction is also a tricky bit that we have to handle, in that it only ever appear in the translation phase if the next block to output is not a successor. To handle this, we maintain the registers type for the start of each block, and assign them to any Goto instruction that points to that block.

Thirdly, the translation of regions and junction are not really straightforward either. We need to know the relation between the source program points and their translation. In particular, a program point in JVM may be related to 0 or more program points in DEX, but each program point in DEX will only have one relation to JVM program points (one to many) with the exception of a Goto instruction. The simplest part of the region translation is that for a program point i, all the program points in DEX that is related to any program points in $region(i)$ will also be in $region(j)$ where j is a DEX program point that is related to i. Again, MoveResult and Goto complicate this process. For MoveResult, because it does not have its corresponding instruction in JVM we have to fix it so that it relates to the Invoke instruction in JVM. Similar case can be made with the Goto instruction in that we tag the source instruction for Goto to be the same as the start of the block which becomes the target of this Goto instruction. This decision of relating the Goto instruction with the target instead of the previous instruction is mainly to handle the case where the Goto instruction is generated by a branching instruction, in which case we need to include Goto in the region as well.

Fourthly, the flag that indicates whether register zero has been used or not does not exist in JVM. The translation itself just need to assign the flags of all instructions to be zero, except the Return instruction and its associated Move and Goto instructions which will have flag of 1. A difficulty arises in that we also need to take care of the moving of parameters which occurs in the first few instructions of the bytecode. These moving parameters instructions also has a Move instruction targetting register zero. Fortunately this always happens as the first instruction in the bytecode, so we can ignore the rule about the flag if it is the first instruction in the bytecode, and just label these instructions with flag 0.

Finally, there are also some instructions that do not directly follow the translation outlined above. In JVM, all Binop instructions are using the values on the stack as the operands, regardless whether they are constants or not. In DEX, BinopConst is dedicated to deal with a binary operation with constant. The modifications to the values themselves will not change, but it has impact on the size of the instructions, i.e. the size of Binop instruction can be one short, or two shorts, but the size of BinopConst is always two shorts. The DEX translation of Dup is also not straightforward. Instead of copying the value from top of the stack and push it on top, it copy the value on top of the stack, create a register containing the copy, and then copy the value from this register to both the top of the stack and new top of the stack.

4) DEX Type Checker: This last component primarily just do a simple type checking on the Android phone. It takes the DEX file, parse the DEX file, and check it against the certificate generated by the certificate translator whether all the instructions in the bytecode satisfy the typability definition. This means that our system works independently of the Android OS, and as such there is no modification at all to the overall Android structure.

We implemented this as an Android application which takes an input a string from user which is the package name for the application that the user wants to type check. Several notes need to be made for this component:

- The DEX file in the package is contained in the predefined file name "classes.dex". We are aware of the possibility that there are multiple DEX file for a single application (e.g. when there are so many methods that it can not be contained within one DEX file), but we decided to ignore it because it is not the main focus of our work.
- We also take the certificate from a predefined file name "Certificate.cert" contained in the "assets" directory. The type checker will just check for this particular file for the certificate, will skip the type checking should a valid certificate is not found.
- There are a lot of generated additional methods and classes for Android application, e.g. "Build.config"

and "R" (and its subclasses). Obviously we could analyze each of those additional thing and provide a proper certificate for each of them, but we decided to just ignore them instead because they are not the focus of this work.

- Similar thing can be said about the methods and classes contained in the Android library. Although for this particular case, since programs will inevitably reference them, we decided to inject the certificate with these methods except that they are stripped off their bytecode instructions. This decision is mainly due to the transfer rule of method invocation which requires the policy oJVMf the target method.
- Since we need to parse the whole DEX file, the process takes a significant amount of time

Algorithmically, Algorithm 4 describe how we implement the DEX type checker for a particular bytecode instructions under a particular policy. Since we need to type check all the methods for all the policy, we just apply this algorithm repeatedly for different methods and policy.

IX. 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. 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 [19], 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. Furthermore, since our target is the relationship between bytecode, we are not writing a full compiler which will generate full DEX files. Following the Compcert project [20], [21], we would ultimately like to have a fully certified end to end compiler. We leave this as future work.

Algorithm 4 Type_check(bm, bmc, mc, lvt, ft)

for i = 1 to bm.length **do**
  $rt := bmc.RT(i)$;
  switch $(bm.instructionAt(i))$
  case (Nop)**:**
    if $bmc.RT(i) \notin bmc.RT(i+1)$ **then return** false;
  case (Const (r, c))**:**
    if r = 0 **then**
      if $bmc.flag(i+1)$ = 0 **then return** false;
    else if $r < lvt.localN$ **then**
      if $bmc.se(i) \not\leq lvt.ka(r)$ **then return** false;
    if $rt \oplus \{r \mapsto bmc.se(i)\} \notin bmc.RT(i+1)$ **then return** false;
  case (Move (r, r_s))**:**
    if r = 0 **then**
      if $bmc.flag(i+1)$ = 0 **then return** false;
    else if $r < lvt.localN$ **then**
      if r_s = 0 **then if** $bmc.flag(i)$ = 1 **then return** false;
      if $bmc.se(i) \sqcup rt(r_s) \not\leq lvt.ka(r)$ **then return** false;
    if $rt \oplus \{r \mapsto bmc.se(i) \sqcup rt(r_s)\} \notin bmc.RT(i+1)$ **then return** false;
  case (Binop (r, r_a, r_b))**:**
    if r = 0 **then**
      if $bmc.flag(i+1)$ = 0 **then return** false;
    else if $r < lvt.localN$ **then**
      if r_a = 0 **or** r_b = 0 **then if** $bmc.flag(i)$ = 1 **then return** false;
      if $bmc.se(i) \sqcup rt(r_a) \sqcup rt(r_b) \not\leq lvt.ka(r)$ **then return** false;
    if $rt \oplus \{r \mapsto bmc.se(i) \sqcup rt(r_a) \sqcup rt(r_b)\} \notin bmc.RT(i+1)$ **then return** false;
  case (Binop2Addr (r_a, r_b))**:**
    if r_a = 0 **then**
      if $bmc.flag(i+1)$ = 0 **then return** false;
    else if $r < lvt.localN$ **then**
      if r_b = 0 **then if** $bmc.flag(i)$ = 1 **then return** false;
      if $bmc.se(i) \sqcup rt(r_a) \sqcup rt(r_b) \not\leq lvt.ka(r_a)$ **then return** false;
    if $rt \oplus \{r_a \mapsto bmc.se(i) \sqcup rt(r_a) \sqcup rt(r_b)\} \notin bmc.RT(i+1)$ **then return** false;
  case (BinopConst (r, r_a, c))**:**
    if r_a = 0 **then**
      if $bmc.flag(i+1)$ = 0 **then return** false;
    else if $r < lvt.localN$ **then**
      if r_a = 0 **then if** $bmc.flag(i)$ = 1 **then return** false;
      if $bmc.se(i) \sqcup rt(r_a) \not\leq lvt.ka(r)$ **then return** false;
    if $rt \oplus \{r \mapsto bmc.se(i) \sqcup rt(r_a)\} \notin bmc.RT(i+1)$ **then return** false;

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. 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

[4] 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.

case (MoveResult (r))**:**
  if $r = 0$ **then**
    if $bmc.flag(i+1) = 0$ **then return** false;
  else if $r < lvt.localN$ **then**
    if $bmc.se(i) \sqcup rt(ret) \not\leq lvt.ka(r)$ **then return** false;
  if $rt \oplus \{r \mapsto bmc.se(i) \sqcup rt(ret)\} \not\equiv bmc.RT(i+1)$ **then return** false;
case (Goto (t))**:**
  if $rt \not\equiv bmc.RT(t)$ **then return** false;
case (If (r_a, r_b, t))**:**
  $k := bmc.se(i) \sqcup rt(r_a) \sqcup rt(r_b)$
  if $r_a = 0$ **or** $r_b = 0$ **then if** $bmc.flag(i) = 1$ **then return** false;
  if $lift(rt, k)\} \not\equiv bmc.RT(i+1)$ **then return** false;
  if $lift(rt, k)\} \not\equiv bmc.RT(t)$ **then return** false;
  for j in bmc.region(i) **do**
    if $k \not\leq bmc$.se(j) **then return** false;
case (Ifz (r, t))**:**
  $k = bmc.se(i) \sqcup rt(r)$
  if $r = 0$ **then if** $bmc.flag(i) = 1$ **then return** false;
  if $lift(rt, k)\} \not\equiv bmc.RT(i+1)$ **then return** false;
  if $lift(rt, k)\} \not\equiv bmc.RT(t)$ **then return** false;
  for j in bmc.region(i) **do**
    if $k \not\leq bmc$.se(j) **then return** false;
case (Return)**: continue**;
case (Return (r))**:**
  if $bmc.se(i) \sqcup rt(r) \not\leq lvt.kr(0)$ **then return** false;
case (NewInstance (r, c))**:**
  if $r = 0$ **then**
    if $bmc.flag(i+1) = 0$ **then return** false;
  else if $r < lvt.localN$ **then**
    if $bmc.se(i) \not\leq lvt.ka(r)$ **then return** false;
  if $rt \oplus \{r \mapsto bmc.se(i)\} \not\equiv bmc.RT(i+1)$ **then return** false;
case (Iget (r, r_o, f))**:**
  if $r = 0$ **then**
    if $bmc.flag(i+1) = 0$ **then return** false;
  else if $r < lvt.localN$ **then**
    if $r_o = 0$ **then if** $bmc.flag(i) = 1$ **then return** false;
    if $bmc.se(i) \sqcup rt(r_o) \sqcup ft(f) \not\leq lvt.ka(r)$ **then return** false;
  if $rt \oplus \{r \mapsto bmc.se(i) \sqcup rt(r_o) \sqcup ft(f)\} \not\equiv bmc.RT(i+1)$ **then return** false;
case (Iput (r, r_o, f))**:**
  $k := bmc.se(i) \sqcup rt(r_o) \sqcup rt(r)$
  if $r = 0$ **or** $r_o = 0$ **then if** $bmc.flag(i) = 1$ **then return** false;
  if $rt \not\equiv bmc.RT(i+1)$ **then return** false;
  if $k \not\leq ft(f)$ **then return** false;
  if $k_h \not\leq ft(f)$ **then return** false;
case (Invoke $(n, m', \vec{p})$)**:**
  $lvt' := mc.find(m', \vec{p}[0]).policy$;
  if $rt \oplus \{ret \mapsto lvt'.kr(0) \sqcup bmc.se(i)\} \not\equiv bmc.RT(i+1)$ **then return** false;
  if $rt(\vec{p}[0]) \sqcup lvt.kh \sqcup bmc.se(i) \not\leq lvt'.kh$ **then return** false
  for $j := 1$ to n **do**
    if $\vec{p}[j] = 0$ **then if** $bmc.flag(i) = 1$ **then return** false;
    if $rt(\vec{p}[j]) \not\leq lvt'.ka[j]$ **then return** false
end for

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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

[13] 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

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

[15] 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.

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

[17] 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.

[18] 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

[19] 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.

[20] 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.

[21] 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.

APPENDIX A
TRANSLATION APPENDIX

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 $\mathbf{BMap} : \mathcal{PP} \rightharpoonup BasicBlocks$. 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.
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{t\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = t$; and
  - $\mathbf{BMap}(t).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
- $P[i]$ is Conditional jump (**ifeq** t) : since there will be 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.
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{i+1, t\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = i + 1$;
  - $\mathbf{BMap}(i+1).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$; and
  - $\mathbf{BMap}(t).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
- $P[i]$ is **Return** : just add the return block as the current block successors, and also update the parent of return block to include the current block, i.e.
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{ret\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = ret$; and
  - $\mathbf{BMap}(ret).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
- $P[i]$ is one of the object manipulation 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.
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{i+1\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = i + 1$;
  - $\mathbf{BMap}(i+1).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$; and
  - for each exception handler j associated with i, let $intPC = maxLabel + j.\mathbf{handlerPC}$ and $hPC = j.\mathbf{handlerPC}$:
    - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{intPC\}$;
    - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{handlers} \cup \{j\}$;
    - $\mathbf{BMap}(intPC).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
    - $\mathbf{BMap}(intPC).\mathbf{succs} \cup \{hPC\}$
    - $\mathbf{BMap}(intPC).\mathbf{insn} = \{\mathbf{moveexception}\}$
    - $\mathbf{BMap}(hPC).\mathbf{parents} \cup \{intPC\}$

  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 $l = \mathbf{getAvailableLabel}$ 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.
  - $l = \mathbf{getAvailableLabel}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{l\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{pSucc} = l$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{parents} = \{i\}$;
  - $\mathbf{BMap} \oplus \{l \mapsto NewBlock\}$;
  - $\mathbf{BMap}(l).\mathbf{succs} = \{i+1\}$;
  - $\mathbf{BMap}(l).\mathbf{pSucc} = (i+1)$;
  - $\mathbf{BMap}(l).\mathbf{insn} = \{\mathbf{moveresult}\}$
  - $\mathbf{BMap}(i+1).\mathbf{parents} \cup \{l\}$; and
  - for each exception handler j associated with i, let $intPC = maxLabel + j.\mathbf{handlerPC}$ and $hPC = j.\mathbf{handlerPC}$:
    - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{intPC\}$;
    - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{handlers} \cup \{j\}$;
    - $\mathbf{BMap}(intPC).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
    - $\mathbf{BMap}(intPC).\mathbf{succs} \cup \{hPC\}$
    - $\mathbf{BMap}(intPC).\mathbf{insn} = \{\mathbf{moveexception}\}$
    - $\mathbf{BMap}(hPC).\mathbf{parents} \cup \{intPC\}$
- $P[i]$ 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 $intPC = maxLabel + j.\mathbf{handlerPC}$ and $hPC = j.\mathbf{handlerPC}$:
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{intPC\}$;
  - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{handlers} \cup \{j\}$;
  - $\mathbf{BMap}(intPC).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
  - $\mathbf{BMap}(intPC).\mathbf{succs} \cup \{hPC\}$
  - $\mathbf{BMap}(intPC).\mathbf{insn} = \{\mathbf{moveexception}\}$
  - $\mathbf{BMap}(hPC).\mathbf{parents} \cup \{intPC\}$
- $P[i]$ is any other instruction : depending whether the next instruction is a start of a block or not.
  - If the next instruction is a start of a block, then update the successor of the current block to include the block of the next instruction and the parent of the block of the next instruction to include the current block i.e.
    - $\mathbf{BMap}(\mathbf{PMap}(i)).\mathbf{succs} \cup \{i+1\}$; and
    - $\mathbf{BMap}(i+1).\mathbf{parents} \cup \{\mathbf{PMap}(i)\}$
  - If the next instruction is not start of a block, then just point the next instruction to have the same pointer as the current block, i.e. $\mathbf{PMap}(i+1) = \mathbf{PMap}(i)$

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

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 ($\textbf{PickOrder}$*)*

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 5 describes how we implement this "trace analysis".

Algorithm 5 PickOrder($blocks$)

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

- **Pick Starting Point**
  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 $\textbf{PickStartingPoint}(p)$.

Algorithm 6 PickStartingPoint($x, loop$)

```
for all p ∈ BMap(x).parents do
    if p ∈ loop then return x;
    bp = BMap(p);
    if bp.pSucc = x and bp.order = −1 then
        loop = loop ∪ {p};
        return PickStartingPoint(p, loop)
return order;
```

- **Trace Successors**
  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 $\textbf{TraceSuccessors}$ 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 7 TraceSuccessors($x, order$)

```
BMap(x).order = order;
if BMap(x).psucc ≠ −1 then
    pSucc = BMap(x).pSucc;
    if BMap(pSucc).order = −1 then return
    TraceSuccessors(pSucc, order + 1);
    for all s ∈ BMap(x).succs do
        if BMap(pSucc).order = −1 then return
        TraceSuccessors(s, order + 1);
return order;
```

E. Output DEX Instructions (**Output**)

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 $\mathbf{newHandler}(cS, cE, hPC, t)$ 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 8 makeHandlerEntry(cH, cS, cE)

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

The only information that are needed to produce the information about exception handlers in DEX is the basic blocks contained in **BMap**. The procedure **translateExceptionHandlers** (Algorithm 9) 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 $\mathbf{ifeq}(r, t)$ and returns its opposite $\mathbf{ifneq}(r, t)$. Finally, we assume that the target of jump instruction can be accessed using the field **target**, e.g. $\mathbf{ifeq}(r, t).\mathbf{target} = t$. The details of the steps in this phase is contained in Algorithm 10.

Algorithm 9 translateExceptionHandlers($blocks$)

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

Algorithm 10 output

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

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	$\frac{P[i] = \mathbf{Push}v}{se, i \vdash^{\mathrm{Norm}} st \Rightarrow se(i) :: st}$	$\frac{P[i] = \mathbf{Const}(r, n) \quad r \notin locR}{se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i)\}\rangle}$
Pop	None	$\frac{P[i] = \mathbf{Pop}}{i \vdash st \Rightarrow st}$	None
Load	Move	$\frac{P[i] = \mathbf{Load}x}{se, i \vdash^{\mathrm{Norm}} st \Rightarrow \big(se(i) \sqcup \vec{k_a}(x)\big) :: st}$	$\frac{P[i] = \mathbf{Move}(r, r_s) \quad r \notin locR}{se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(se(i) \sqcup sec(r_s)\big)\}\rangle}$
Store	Move	$\frac{P[i] = \mathbf{Store}x \quad k \sqcup se(i) \leq \vec{k_a}(x)}{\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow st}$	$\frac{P[i] = \mathbf{Move}(r, r_s) \quad sec(r_s) \sqcup se(i) \leq \vec{k_a}(r)}{\vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i) \sqcup sec(r_s)\}\rangle}$
Binop	Binop	$\frac{P[i] = \mathbf{Binop}}{se, i \vdash^{\mathrm{Norm}} a :: b :: st \Rightarrow \big(se(i) \sqcup a \sqcup b\big) :: st}$	$\frac{P[i] = \mathbf{Binop}(r, r_a, r_b) \quad r \notin locR}{se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(se(i) \sqcup sec(r_a) \sqcup sec(r_b)\big)\}\rangle}$
Swap	Move	$\frac{P[i] = \mathbf{Swap}}{i \vdash^{\mathrm{Norm}} k_1 :: k_2 :: st \Rightarrow k_2 :: k_1 :: st}$	$\frac{P[i] = \mathbf{Move}(r, r_s) \quad r \notin locR}{se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(se(i) \sqcup sec(r_s)\big)\}}$
Goto	Goto	$\frac{P[i] = \text{Goto } t}{i \vdash st \Rightarrow st}$	$\frac{P[i] = \text{Goto } t}{i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt\rangle}$ *) Not directly translated
Ifeq	Ifeq	$\frac{P[i] = \mathbf{ifeq}t \quad \forall j' \in \mathbf{region}(i), k \leq se(j')}{\mathbf{reigon}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow \mathrm{lift}_k(st)}$	$\frac{P[i] = \mathbf{ifeq}(r, t) \quad \forall j' \in \mathrm{region}(i), se(i) \sqcup rt(r) \leq se(j')}{\mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow \mathbf{lift}_{\mathbf{sec(r)}}(rt)}$
	Ifneq	Ifeq may be translated into Ifneq on certain condition	$\frac{P[i] = \mathbf{ifneq}(r, t) \quad \forall j' \in \mathrm{region}(i), se(i) \sqcup rt(r) \leq se(j')}{\mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow \mathbf{lift}_{\mathbf{sec(r)}}(rt)}$
New	New	$\frac{P[i] = \mathbf{new}C}{se, i \vdash^{\mathrm{Norm}} st \Rightarrow se(i) :: st}$	$\frac{P[i] = \mathbf{new}(r, c) \quad r \notin locR}{se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i)\}\rangle}$
Getfield	Iget	$\frac{P[i] = \mathbf{getfield}f \quad k \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \leq se(j)}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} k :: st \Rightarrow \mathbf{lift_k}((k \sqcup \mathbf{ft}(f) \sqcup se(i)) :: st)}$ $\frac{P[i] = \mathbf{getfield}f \quad k \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$ $\frac{P[i] = \mathbf{getfield}f \quad k \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad k \leq \vec{k_r}[\mathrm{np}]}{\mathbf{ft}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k :: st \Rightarrow}$	$\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad r \notin locR \quad sec(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_o) \leq se(j)}{\mathbf{ft}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto sec(r_o) \sqcup \mathbf{ft}(f) \sqcup se(i)\}\rangle}$ $\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad r \notin locR \quad sec(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), sec(r_o) \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{ft}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto sec(r_o) \sqcup se(i)\}\rangle}$ $\frac{P[i] = \mathbf{iget}(r, r_o, f) \quad r \notin locR \quad sec(r_o) \in \mathcal{S} \quad sec(r_o) \leq \vec{k_r}[\mathrm{np}] \quad \forall j \in \mathbf{region}(i, \mathrm{np}), sec(r_o) \leq se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow}$
Putfield	Iput	$\frac{P[i] = \mathbf{putfield}f \quad k_h \leq \mathbf{ft}(f) \quad k_1 \sqcup se(i) \sqcup k_2 \leq \mathbf{ft}(f) \quad k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k_2 \leq se(j)}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} k_1 :: k_2 :: st \Rightarrow \mathbf{lift_{k_2}}(st)}$ $\frac{P[i] = \mathbf{putfield}f \quad k_1 \sqcup se(i) \sqcup k_2 \leq \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) = t \quad k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k_2 \leq se(j)}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$ $\frac{P[i] = \mathbf{putfield}f \quad k_1 \sqcup se(i) \sqcup k_2 \leq \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad k_1 \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k_2 \leq se(j) \quad k_2 \leq \vec{k_r}[\mathrm{np}]}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: st \Rightarrow}$	$\frac{P[i] = \mathbf{iput}(r_s, r_o, f) \quad k_h \leq \mathbf{ft}(f) \quad sec(r_o) \in \mathcal{S} \quad sec(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad sec(r_o) \sqcup se(i) \sqcup sec(r_s) \leq \mathbf{ft}(f) \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_o) \leq se(j)}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec(r_o)}}(rt)\rangle}$ $\frac{P[i] = \mathbf{iput}(r_s, r_o, f) \quad k_h \leq \mathbf{ft}(f) \quad sec(r_o) \in \mathcal{S} \quad sec(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad sec(r_o) \sqcup se(i) \sqcup sec(r_s) \leq \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) = t \quad \forall j \in \mathbf{region}(i, \mathrm{np}), sec(r_o) \leq se(j)}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto sec(r_o) \sqcup se(i)\}\rangle}$ $\frac{P[i] = \mathbf{iput}(r_s, r_o, f) \quad k_h \leq \mathbf{ft}(f) \quad sec(r_o) \in \mathcal{S} \quad sec(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad sec(r_o) \sqcup se(i) \sqcup sec(r_s) \leq \mathbf{ft}(f) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad \forall j \in \mathbf{region}(i, \mathrm{np}), sec(r_o) \leq se(j) \quad sec(r_o) \leq \vec{k_r}[\mathrm{np}]}{\mathbf{ft}, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow}$

JVM	DEX	Original Typing Rule	Related DEX Typing Rule
Newarray	Newarray	$\dfrac{P[i] = \mathbf{newarray}t \quad k \in \mathcal{S}}{i \vdash^{\mathrm{Norm}} k :: st \Rightarrow k[\mathbf{at}(i)] :: st}$	$\dfrac{P[i] = \mathbf{newarray}(r, r_l, t) \quad sec(r_l) \in \mathcal{S} \quad r \notin locR}{i \vdash^{\mathrm{Norm}} rt \Rightarrow rt \oplus \{r \mapsto sec(r_l)[\mathbf{at}(i)]\}}$
Arraylength	Arraylength	$\dfrac{P[i] = \mathbf{arraylength} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \le se(j) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}}}{\mathbf{region}, se, i \vdash^{\mathrm{Norm}} k[k_c] :: st \Rightarrow \mathbf{lift_k}(k :: st)}$ $\dfrac{P[i] = \mathbf{arraylength} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(j) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{region}, se, i \vdash^{\mathrm{np}} k[k_c] :: st \Rightarrow (k \sqcup se(i)) :: \epsilon}$ $\dfrac{P[i] = \mathbf{arraylength} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(j) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad k \le \vec{k_r}[\mathrm{np}]}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{np}} k[k_c] :: st \Rightarrow}$	$\dfrac{P[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \le se(j)}{\mathbf{region}, se, i \vdash^{norm} \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift_k}(rt \oplus \{r \mapsto k\})\rangle}$ $\dfrac{P[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{region}, se, i \vdash^{np} \langle f, rt\rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}\rangle}$ $\dfrac{P[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad k \le \vec{k_a}[\mathrm{np}]}{\vec{k_a} \to \vec{k_r}, \mathbf{region}, se, i \vdash^{np} \langle f, rt\rangle \Rightarrow}$
Arrayload	Aget	$\dfrac{P[i] = \mathbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}) k_2 \le se(j)}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} k_1 :: k_2[k_c] :: st \Rightarrow \mathbf{lift_{k_2}}\big((k_1 \sqcup k_2) \sqcup^{\mathrm{ext}} k_c\big) :: st)}$ $\dfrac{P[i] = \mathbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{np}) k_2 \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2[k_c] :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$ $\dfrac{P[i] = \mathbf{arrayload} \quad k_1, k_2 \in \mathcal{S} \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad k_2 \le \vec{k_r}[\mathrm{np}] \quad \forall j \in \mathbf{region}(i, \mathrm{np}) k_2 \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2[k_c] :: st \Rightarrow}$	$\dfrac{P[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad k, sec(r_i) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \le se(j)}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{norm} \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift_k}(rt \oplus \{r \mapsto ((k \sqcup sec(r_i)) \sqcup^{\mathrm{ext}} k_c)\})\rangle}$ $\dfrac{P[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad k, sec(r_i) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{region}, se, i \vdash^{np} \langle f, rt\rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}\rangle}$ $\dfrac{P[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k_c \in \mathcal{S}^{\mathrm{ext}} \quad k, sec(r_i) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad k \le \vec{k_r}[\mathrm{np}]}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{np} \langle f, rt\rangle \Rightarrow}$
Arraystore	Aput	$\dfrac{P[i] = \mathbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\mathrm{ext}} \quad k_2, k_3 \in \mathcal{S} \quad ((k_2 \sqcup k_3) \sqcup^{\mathrm{ext}} k_1) \le^{\mathrm{ext}} k_c \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k_2 \le se(j)}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow \mathbf{lift_{k_2}}(st)}$ $\dfrac{P[i] = \mathbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\mathrm{ext}} \quad ((k_2 \sqcup k_3) \sqcup^{\mathrm{ext}} k_1) \le^{\mathrm{ext}} k_c \quad k_2, k_3 \in \mathcal{S} \quad \mathbf{Handler}(i, \mathrm{np}) = t \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k_2 \le se(j)}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow (k_2 \sqcup se(i)) :: \epsilon}$ $\dfrac{P[i] = \mathbf{arraystore} \quad k_1, k_c \in \mathcal{S}^{\mathrm{ext}} \quad k_2, k_3 \in \mathcal{S} \quad ((k_2 \sqcup k_3) \sqcup^{\mathrm{ext}} k_1) \le^{\mathrm{ext}} k_c \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k_2 \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad k_2 \le \vec{k_r}[\mathrm{np}]}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{np}} k_1 :: k_2 :: k_3[k_c] :: st \Rightarrow}$	$\dfrac{P[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad ((k \sqcup sec(r_i)) \sqcup^{\mathrm{ext}} sec(r_s)) \le^{\mathrm{ext}} k_c \quad k_c, sec(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \le se(i)}{\vec{k_a} \to k_r, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} rt \Rightarrow \mathbf{lift_k}(rt)}$ $\dfrac{P[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad ((k \sqcup sec(r_i)) \sqcup^{\mathrm{ext}} sec(r_s)) \le^{\mathrm{ext}} k_c \quad k_c, sec(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(i) \quad \mathbf{Handler}(i, \mathrm{np}) = t}{\mathbf{region}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto k \sqcup se(i)\}\rangle}$ $\dfrac{P[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad ((k \sqcup sec(r_i)) \sqcup^{\mathrm{ext}} sec(r_s)) \le^{\mathrm{ext}} k_c \quad k_c, sec(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad \forall j \in \mathbf{region}(i, \mathrm{np}), k \le se(i) \quad \mathbf{Handler}(i, \mathrm{np}) = t \quad k \le \vec{k_r}[\mathrm{np}]}{\vec{k_a} \to \vec{k_r}\mathbf{region}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow}$
Return	Move **and** Goto **or** Return	$\dfrac{P[i] = \mathbf{return} \quad se(i) \sqcup k \le k_r}{\vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash k :: st \Rightarrow}$	$\dfrac{P[i] = \mathbf{Move}(r_0, r_s)}{se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 1, rt \oplus \{r \mapsto \big(se(i) \sqcup rt(r_s)\big)\}\rangle}$ $\dfrac{P[i] = \mathbf{goto}(t)}{i \vdash rt \Rightarrow rt}$ $\dfrac{P[i] = \mathbf{return}(r_s) \; se(i) \sqcup rt(r_s) \le k_r}{\vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash rt \Rightarrow}$

JVM	DEX	Original Typing Rule	Related DEX Typing Rule
Invoke	Invoke	$\frac{P_m[i] = \mathbf{invoke} m_{\mathrm{ID}} \quad \mathbf{length}(st_1) = \mathrm{nbArguments}(m_I D) \quad k \leq \vec{k_a}[0] \quad \forall i \in [0, \mathbf{length}(st_1) - 1] st_1[i] \leq \vec{k'_a}[i+1] \quad k \sqcup k_h \sqcup se(i) \leq k'_h \quad k_e = \bigsqcup\{\vec{k'_r}[e] \mid e \in \mathbf{excAnalysis}(m_{\mathrm{ID}})\} \quad \Gamma_{m_{ID}}[k] = \vec{k'_a} \overset{k'_h}{\rightarrow} k'_r \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), k \sqcup k_e \leq se(j)}{\Gamma, \mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, i \vdash^{\mathrm{Norm}} st_1 :: k :: st_2 \Rightarrow \mathbf{lift}_{\mathbf{k} \sqcup \mathbf{ke}}((k'_r \sqcup k \sqcup se(i)) :: st_2)}$	$\frac{P_m[i] = \text{invoke } (n, m, \vec{p}) \quad \Gamma_{m'}[sec(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\rightarrow} k'_r \quad sec(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall_{0 \leq j < n} sec(\vec{p}[j]) \leq \vec{k'_a}[j] \quad k_e = \bigsqcup\{\vec{k'_r}[e] \mid e \in \mathbf{excAnalysis}(m')\} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(\vec{p}[0]) \sqcup k_e \leq se(j)}{\Gamma, \mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} k_r, i \vdash^{\mathrm{Norm}} \langle f, rt \rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec(\vec{p}[0]) \sqcup k_2}}(rt \oplus \{ret \mapsto k'_r[n] \sqcup sec(\vec{p}[0]) \sqcup se(i)\})\rangle}$
		$\frac{P_m[i] = \mathbf{invoke} m_{\mathrm{ID}} \quad \mathbf{length}(st_1) = \mathrm{nbArguments}(m_I D) \quad k \leq \vec{k_a}[0] \quad \forall i \in [0, \mathbf{length}(st_1) - 1] st_1[i] \leq \vec{k'_a}[i+1] \quad k \sqcup k_h \sqcup se(i) \leq k'_h \quad e \in \mathbf{excAnalysis}(m_{\mathrm{ID}}) \cup \{\mathrm{np}\} \quad \Gamma_{m_{ID}}[k] = \vec{k'_a} \overset{k'_h}{\rightarrow} k'_r \quad \forall j \in \mathbf{region}(i, e), k \sqcup k_e \leq se(j) \quad \mathbf{Handler}(i, e) = t}{\Gamma, \mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow (k \sqcup \vec{k'_r}[e]) :: \epsilon}$	$\frac{P_m[i] = \text{invoke } (n, m, \vec{p}) \quad \Gamma_{m'}[sec(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\rightarrow} k'_r \quad sec(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall_{0 \leq j < n} sec(\vec{p}[j]) \leq \vec{k'_a}[j] \quad e \in \mathbf{excAnalysis}(m') \cup \{\mathrm{np}\}\} \quad \mathbf{Handler}(i, \mathrm{np}) = t \quad \forall j \in \mathbf{region}(i, e), sec(\vec{p}[0]) \sqcup \vec{k'_r}[e] \leq se(j)}{\Gamma, \mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} k_r, i \vdash^{e} \langle f, rt \rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto k'_r[e] \sqcup sec(\vec{p}[0])\}\rangle}$
		$\frac{P_m[i] = \mathbf{invoke} m_{\mathrm{ID}} \quad \mathbf{length}(st_1) = \mathrm{nbArguments}(m_I D) \quad k \leq \vec{k_a}[0] \quad \forall i \in [0, \mathbf{length}(st_1) - 1] st_1[i] \leq \vec{k'_a}[i+1] \quad k \sqcup k_h \sqcup se(i) \leq k'_h \quad e \in \mathbf{excAnalysis}(m_{\mathrm{ID}}) \cup \{\mathrm{np}\} \quad \Gamma_{m_{ID}}[k] = \vec{k'_a} \overset{k'_h}{\rightarrow} k'_r \quad \forall j \in \mathbf{region}(i, e), k \sqcup k_e \leq se(j) \quad \mathbf{Handler}(i, e) \uparrow \quad k \sqcup se(i) \sqcup \vec{k'_r}[e] \leq \vec{k_r}[e]}{\Gamma, \mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} \vec{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow}$	$\frac{P_m[i] = \text{invoke } (n, m, \vec{p}) \quad \Gamma_{m'}[sec(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\rightarrow} k'_r \quad sec(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall_{0 \leq j < n} sec(\vec{p}[j]) \leq \vec{k'_a}[j] \quad e \in \mathbf{excAnalysis}(m') \cup \{\mathrm{np}\}\} \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad \forall j \in \mathbf{region}(i, e), sec(\vec{p}[0]) \sqcup k'_r[e] \leq se(j) \quad \mathrm{sec}(\vec{p}[0]) \sqcup se(i) \sqcup \vec{k'_r}[e] \leq \vec{k_r}[e]}{\Gamma, \mathbf{region}, se, \vec{k_a} \overset{k_h}{\rightarrow} k_r, i \vdash^{e} \langle f, rt \rangle \Rightarrow}$
	Moveresult		$\frac{P_m[i] = \mathbf{moveresult}(r) \quad r \notin locR}{i \vdash^{\mathrm{Norm}} \langle f, rt \rangle \Rightarrow \langle f, rt \oplus \{r \mapsto rt(ret)\}\rangle}$

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 $a \mapsto^{\mathrm{Norm}} b$, then either*

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

in $[\![P]\!]$.

Proof: To prove this lemma, we first unfold the definition of compilation. Using the information that $a \mapsto b$, there are several possible cases to output the block depending whether what instruction is Ins_a and where a and b are located. Either:

- $\llparenthesis b \rrparenthesis$ are placed directly after $\llparenthesis a \rrparenthesis$ and $\llparenthesis a \rrparenthesis [n-1]$ is sequential instruction;
  In this case, appealing to the definition of successor relation this trivially holds as the first case.
- $\llparenthesis Ins_a \rrparenthesis$ ends in a sequential instruction and will have a **goto** instruction appended that points to $\llbracket \llparenthesis b \rrparenthesis [0] \rrbracket$; Again appealing to the definition of successor relation this trivially holds as the second case, where $P_{\mathrm{DEX}}[(\llbracket \llparenthesis a \rrparenthesis [n-1] \rrbracket + 1)] = \mathbf{goto}(\llbracket \llparenthesis b \rrparenthesis [0] \rrbracket)$.
- $\llparenthesis Ins_a \rrparenthesis [n-1]$ is a branching instruction and b is one of its child, and $\llparenthesis b \rrparenthesis$ is placed directly after $\llparenthesis a \rrparenthesis$ 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.
- $\llparenthesis Ins_a \rrparenthesis [n-1]$ is a branching instruction and b is one of its child, nevertheless $\llparenthesis b \rrparenthesis$ is not placed directly after $\llparenthesis a \rrparenthesis$ nor is pointed to by the branching instruction;
  In this case, according to the **Output** phase, a **goto** instruction will be appended in $(\llbracket \llparenthesis a \rrparenthesis [n-1] \rrbracket + 1)$ 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 τ. Let e be the index to*

the instructions within $\llbracket a \rrbracket$ *that throws exception. If* $a \mapsto^{\tau} b$, *then* $\llbracket\llbracket a \rrbracket[e]\rrbracket \mapsto^{\tau} \llbracket\llbracket h \rrbracket\rrbracket$ *and* $\llbracket\llbracket h \rrbracket\rrbracket \mapsto^{\text{Norm}} \llbracket\llbracket b \rrbracket[0]\rrbracket$

Proof: Trivial based on the unfolding definition of the compiler, where there is a block containing sole instruction **moveexception**, which will be pointed by the exception handler in DEX, between possibly throwing instruction and its handler for particular exception class τ. 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* $a \mapsto^{\text{Norm}} b$, *then* $\llbracket\llbracket a \rrbracket[0]\rrbracket \mapsto^{\text{Norm}} \llbracket\llbracket a \rrbracket[1]\rrbracket$ *and* $\llbracket\llbracket a \rrbracket[1]\rrbracket \mapsto^{\text{Norm}} \llbracket\llbracket b \rrbracket[0]\rrbracket$

Proof: This is trivial based on the unfolding definition of the compiler since the primary successor of $\llbracket\llbracket a \rrbracket[0]\rrbracket$ is $\llbracket\llbracket a \rrbracket[1]\rrbracket$, where $\llbracket P \rrbracket[\llbracket\llbracket a \rrbracket[1]\rrbracket] =$ **moveresult**, and the primary successor of $\llbracket\llbracket a \rrbracket[1]\rrbracket$ is $\llbracket\llbracket b \rrbracket[0]\rrbracket$. 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* $\llbracket P[s] \rrbracket$ *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*

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

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 $\llbracket P[k] \rrbracket$ be the translation of instruction $P[k]$ and k_1 is the address of its first instruction ($\llbracket P[k] \rrbracket[0]$). 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 $\llbracket P[j] \rrbracket$. Denote i_b as the program point in the sequence $\llbracket i \rrbracket$ and is a branching point. Let $\llbracket P[k] \rrbracket$ be the translation of instruction $P[k]$ and k_1 is the address of its first instruction ($\llbracket P[k] \rrbracket[0]$). 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 $\llbracket i \rrbracket$ that branch and k_{11} and k_{21} be the first instruction in $\llbracket P[k_1] \rrbracket$ and $\llbracket P[k_2] \rrbracket$ 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 τ' 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 $\llbracket a \rrbracket$ also has more codes than $\llbracket b \rrbracket$.

■

B. Proof that Translation Preserves Typability

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 $rt \sqsubseteq \llbracket S_1 \rrbracket$ and $S_1 \sqsubseteq S_2$, then $rt \sqsubseteq \llbracket S_2 \rrbracket$ as well.*

Proof: Trivial based on the definition of $\llbracket . \rrbracket$ and the $\sqsubseteq$ for register types. ∎

Lemma (VII.2)**.** *For any JVM program P with instruction Ins at address i and tag* Norm*, let the length of $\llbracket Ins \rrbracket$ denoted by n. Let $RT_{\llbracket i \rrbracket[0]} = \llbracket S_i \rrbracket$. If according to the transfer rule for $P[i] = ins$ exists st s.t. $i \vdash^{\text{Norm}} S_i \Rightarrow st$ then*

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

according to the transfer rule(s) of $\llbracket Ins \rrbracket$

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 $se(\llbracket i \rrbracket)$ 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

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

  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 $\llbracket st \rrbracket = rt$ because $RT_{\llbracket i \rrbracket[0]} = \llbracket S_i \rrbracket$ and the rt we have is the same as $\llbracket se(i) :: S_i \rrbracket$ thus $rt \sqsubseteq \llbracket st \rrbracket$
- **Pop**
  In this case, since the instruction does not get translated, this instruction does not affect the lemma.
- **Load** x
  Similar to **Push** except that the security value pushed on top of the stack is $se(\llbracket i \rrbracket[0]) \sqcup \vec{k_a}(x)$. And although there are several transfer rules for **move**, there is only one applicable because the source register comes from local variable register, and the target register is one of the stack space. Using this transfer rule, we can trivially show that $rt = \llbracket st \rrbracket$ where $st = \left(se(i) \sqcup \vec{k_a}(x)\right) :: S_i$ and $rt = RT_{\llbracket i \rrbracket[0]} \oplus \{r(TS_i) \mapsto se(\llbracket i \rrbracket[0]) \sqcup \vec{k_a}(x)\}$, thus $rt \sqcup \llbracket st \rrbracket$.
- **Store** x
  This instruction is also translated as **move** except that the source register is the top of the stack and the target register is one of the local variable register. The rt in this case will be

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

  This rt coincides with the transfer rule for **move** where the target register is a register used to contain local variable. Since we know that the x is in the range of local variable, we will have that $rt \sqsubseteq \llbracket st \rrbracket$ based on the definition of $\sqsubseteq$, $\llbracket . \rrbracket$ 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. Let

  $$rt = \mathbf{lift}_{\mathbf{RT}_{\llbracket i \rrbracket[0]}\left(\mathbf{r(TS_i - 1)}\right)}\left(RT_{\llbracket i \rrbracket[0]}\right)$$

  Since the **lift** does not affect registers in the local variable side, we know that for all registers $r \in locR$, $rt(r) = \vec{k_a}(r)$ so they coincide. For registers in the stack space, since they are the lub of values originally the same $RT_{\llbracket i \rrbracket[0]} = \llbracket S_i \rrbracket$, with a value that is the same for both $(RT_{\llbracket i \rrbracket[0]}\big(r(TS_i - 1)\big) = S_i[0])$, therefore the resulting registers type will also be the same, i.e. $rt \sqsubseteq \llbracket \mathbf{lift_k}(st') \rrbracket$ where $S_i = k :: st'$.
- **Binop**
  Translated as a DEX instruction for specified binary operator with the source taken from the top two values from the stack, and then put the resulting value in the then would be top of the stack. Let rt in this case comes from

  $$\begin{aligned} rt = & RT_{\llbracket i \rrbracket[0]} \oplus \{r(TS_i - 1) \mapsto se(\llbracket i \rrbracket[0]) \sqcup \\ & RT_{\llbracket i \rrbracket[0]}\big(r(TS_i - 1)\big) \sqcup \big(RT_{\llbracket i \rrbracket[0]}\big(r(TS_i - 2)\big)\big)\} \end{aligned}$$

  This rt corresponds to the scheme of DEX transfer rule for binary operation. Then we will have that $rt \sqsubseteq \llbracket st \rrbracket$ where $st = se(i) \sqcup k_a \sqcup k_b :: st'$ and $S_i = k_a :: k_b :: st'$
- **Swap**
  In dx tool, this instruction is translated into 4 **move** instructions. In this case, such rt is

  $$\begin{aligned} rt = RT_{\llbracket i \rrbracket[0]} \oplus \{ & \\ r(TS_i) & \mapsto se(\llbracket i \rrbracket[0]) \sqcup RT_{\llbracket i \rrbracket[0]}\big(r(TS_i - 2)\big), \\ r(TS_i + 1) & \mapsto se(\llbracket i \rrbracket[1]) \sqcup RT_{\llbracket i \rrbracket[1]}\big(r(TS_i - 1)\big), \\ r(TS_i - 2) & \mapsto se(\llbracket i \rrbracket[2]) \sqcup RT_{\llbracket i \rrbracket[2]}\big(r(TS_i - 1)\big), \\ r(TS_i - 1) & \mapsto se(\llbracket i \rrbracket[3]) \sqcup RT_{\llbracket i \rrbracket[3]}\big(r(TS_i - 2)\big)\} \end{aligned}$$

  justified by applying transfer rule for **move** 4 times. As before, appealing to the definition of $\sqsubseteq$ to establish

that this $rt \sqsubseteq \lfloor\!\lfloor st \rfloor\!\rfloor$ 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

$$\begin{aligned} RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[1]} &= RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]} \oplus \{r(TS_i) \mapsto \\ &\quad se(\lfloor\!\lfloor i \rfloor\!\rfloor[0]) \sqcup RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]}\big(r(TS_i - 2)\big)\} \\ RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[2]} &= RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[1]} \oplus \{r(TS_i + 1) \mapsto \\ &\quad se(\lfloor\!\lfloor i \rfloor\!\rfloor[1]) \sqcup RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[1]}\big(r(TS_i - 1)\big)\} \\ RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[3]} &= RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[2]} \oplus \{r(TS_i - 2) \mapsto \\ &\quad se(\lfloor\!\lfloor i \rfloor\!\rfloor[2]) \sqcup RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[3]}\big(r(TS_i - 1)\big)\} \end{aligned}$$

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

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

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

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

- **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 $rt = \lfloor\!\lfloor S_i \rfloor\!\rfloor$, then by the definition of $\sqsubseteq$ we'll have $rt \sqsubseteq \lfloor\!\lfloor st \rfloor\!\rfloor$ since $S_i = k_o :: k_v :: st$.

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

$$\begin{aligned} rt = RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]} \quad &\oplus \{r(TS_i - 1) \mapsto \\ &RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]}\big(r(TS_i - 1)\big)[\mathbf{at}(\lfloor\!\lfloor i \rfloor\!\rfloor[0])]\} \end{aligned}$$

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

- **Arraylength**
Let $k[k_c] = RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]}\big(r(TS_i - 1)\big) = S_i[0]$. In this case $rt = RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]} \oplus \{r(TS_i - 1) \mapsto k\} = \lfloor\!\lfloor st \rfloor\!\rfloor$ then we will have $rt = \lfloor\!\lfloor st \rfloor\!\rfloor$ where $st = k :: st'$ and $S_i = k[k_c] :: st'$ which will give us $rt \sqsubseteq \lfloor\!\lfloor st \rfloor\!\rfloor$.

- **Arrayload**
Let $k[k_c] = RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]}\big(r(TS_i - 2)\big) = S_i[1]$. In this case

$$\begin{aligned} rt = &RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]} \oplus \{r(TS_i - 2) \mapsto \\ &\big(se(i) \sqcup k \sqcup RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]}(r(TS_i - 1))\big) \sqcup^{\text{ext}} k_c\} \end{aligned}$$

which coincides with $\lfloor\!\lfloor st \rfloor\!\rfloor$ 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 $rt \sqsubseteq \lfloor\!\lfloor st \rfloor\!\rfloor$.

- **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 $\vec{k'_a} \xrightarrow{k'_h} \vec{k'_r}$ 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 $k_e = \bigsqcup\{\vec{k'_r}[e] \mid e \in \mathbf{excAnalysis}(m_{\mathrm{ID}})$. 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

$$\begin{aligned} \mathbf{lift}_{\mathbf{k} \sqcup \mathbf{k_e}}(RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]} \oplus \{ \quad &ret \mapsto \big(\vec{k'_r}[n] \sqcup se(\lfloor\!\lfloor i \rfloor\!\rfloor[0])\big), \\ &r_t \mapsto \big(\vec{k'_r}[n] \sqcup se(\lfloor\!\lfloor i \rfloor\!\rfloor[1])\big)\}) \end{aligned}$$

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

$$\mathbf{lift}_{\mathbf{k} \sqcup \mathbf{k_e}}(RT_{\lfloor\!\lfloor i \rfloor\!\rfloor[0]} \oplus \{ret \mapsto \big(\vec{k'_r}[n] \sqcup se(\lfloor\!\lfloor i \rfloor\!\rfloor[0])\big)\})$$

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 $\tau \neq \mathrm{Norm}$, but if the exception is uncaught then the instruction is a return point.

■

Lemma (VII.3)**.** *For any JVM program P with instruction Ins at address i and tag $\tau \neq \text{Norm}$ with exception handler at address i_e. Let the length of $[\![Ins]\!]$ until the instruction that throw exception τ 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*

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

according to the transfer rule(s) of first n instruction in $[\![Ins]\!]$ 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 $rt \sqsubseteq [\![st]\!]$, where $st = (k \sqcup \vec{k'_r}[e]) :: \epsilon$. Such rt is obtained from the transfer rule for **invoke** where an exception of tag τ 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
  $$rt = \{\vec{k_a}, ex \mapsto \big(k \sqcup se([\![i]\!][0])\big), \\ r(locN) \mapsto \big(k \sqcup se([\![i]\!][0])\big)\}$$
  and it will make the relation $rt \sqsubseteq [\![st]\!]$ 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
  $$RT_{(be,0)}\{\vec{k_a}, ex \mapsto (k \sqcup se([\![i]\!][0])\}$$
  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.4)**.** *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 $[\![ins]\!]$. Let $RT_{[\![i]\!][0]} = [\![S_i]\!]$, $RT_{[\![j]\!][0]} = [\![S_j]\!]$ and rt is obtained from the transfer rules involved in $[\![ins]\!]$. Then $rt \sqsubseteq RT_{[\![j]\!][0]}$.*

Proof: Using Lemma VII.2 and Lemma VII.3 to establish that we have $rt \sqsubseteq [\![st]\!]$. Then we conclude by using Lemma B.5 to establish that $rt \sqsubseteq RT_{[\![j]\!][0]}$ because $st \sqsubseteq S_j$. ■

Lemma (VII.5)**.** *Let Ins be instruction at program point i, S_i its corresponding stack types, and let $RT_{[\![i]\!][0]} = [\![S_i]\!]$. If $P[i]$ satisfy the typing constraint for Ins with the stack type S_i, then $\forall (bj, j) \in [\![i]\!].P_{DEX}[bj, j]$ will also satisfy the typing constraints for all instructions in $[\![Ins]\!]$ with the initial registers typing $RT_{[\![i]\!][0]}$.*

Proof: We do this by case by case instruction:

- **Push**
  This instruction is translated into **Const**. There are several transfer rules applicable to **Const** depending on the target register. Since the translation will only yield **Const** that operates on the stack space, the only transfer rule that is applicable is the transfer rule where there is no constraint involved.
- **Pop**: does not get translated.
- **Load** x
  Although the translated instruction is **move**, but its target register is in the stack space which means that the translated instruction does not have any constraint.
- **Store** x
  This time, there is a constraint on the instruction although it is the same **move** instruction due to the target of instruction which resides on the local variable side. Since $se(i)$ is translated directly, basically we only need to show that $rt\big(r(TS_i - 1)\big) \leq \vec{k_a}(x)$. Since we know from JVM transfer rule satisfaction that we have $S_i = k :: st, k \leq \vec{k_a}(x)$ and the fact that the policy on local variable gets translated as it is, we have $rt\big(r(TS_i)\big) \leq \vec{k_a}(x)$ because we have $rt\big(r(TS_i)\big) = k$ from the definition of $[\![.]\!]$ on stack types.
- **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 $\forall j' \in \textbf{region}(i, \text{Norm}), rt\big(r(TS_i - 1)\big) \leq se(j')$, and we know that in the JVM bytecode the constraint $\forall j' \in$

$\mathbf{region}(i, \mathrm{Norm}), k \leq se(j')$ is fulfilled. Based on the definition of $[\![.]\!]$, we will have $k = rt\big(r(TS_i - 1)\big)$. Thus we only need to prove that the difference in region will still preserve the constraint satisfaction. We do this by proof by contradiction. Suppose there exists such instruction at address $(bj, j) \in \mathbf{region}([\![i]\!][n])$ such that $k \not\leq se(bj, j)$. But according to definition VII.2, such instruction will come from an instruction at address i' s.t. $i' \in \mathbf{region}(i)$ thus it will satisfy $k \leq se(i')$. By definition VII.7, $se(bj, j) = se(i')$, thus we will have $k \leq se(bj, j)$. A plain contradiction.

- **Binop**
  Although there are transfer rule for **binop** where constraints is involved, since the translation from JVM to DEX yields binop that only operates on the stack space, there is no constraint involved in the instruction.

- **Swap**
  Trivially holds as well because all the 4 **move** instructions translated from **swap** will operate on the stack space, so there is no constraint involved.

- **New**
  Trivially holds as the object reference created will be put in the stack space, so there is no constraint involved.

- **Getfield**
  The value obtained from the object's field is put into the stack, so we do not have to deal with the constraint against $\vec{k_a}$. 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 $sec(r_o) \in \mathcal{S}$ and $\forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_o) \leq se(j)$. 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 $sec(r_o) = k$, therefore we can conclude that $sec(r_o) \in \mathcal{S}$. 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 $sec(r_o) \leq \vec{k_r}[\mathrm{np}]$. 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 $sec(r_o) = k$. 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 $sec(r_o) \in \mathcal{S}$ and $sec(r_s) \in \mathcal{S}^{\mathrm{ext}}$ 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, $(sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} sec(r_s) \leq \mathbf{ft}(f)$ 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 $k_1 = sec(r_s)$, $k_2 = sec(r_o)$, and se stays the same, therefore the constraint $(sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} sec(r_s) \leq \mathbf{ft}(f)$ 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 reference to the new array created is put into the stack space, thus there is only one constraint involved which is proved by the definition of $[\![.]\!]$.

- **Arraylength**
  Since the length of the array is put into the stack space, we do not need to concern ourselves with the constraint that restrict the value to put into the local variables. 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 $sec(r_a) = k[k_c]$. 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**
  Since the length of the array is put into the stack space, we do not need to concern ourselves with the constraint that restrict the value to put into the local variables. We first deal with the constraints $k, sec(r_i) \in \mathcal{S}$ and $k_c \in \mathcal{S}^{\mathrm{ext}}$. From the definition of $[\![.]\!]$, we know that $sec(r_a) = k_2[k_c]$ and $sec(r_i) = k_1$. Since we know that JVM typing satisfies all the constraint, we know that $sec(r_i) \in \mathcal{S}$ 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 $k', sec(r_i) \in \mathcal{S}$ and $k_1, k_c \in \mathcal{S}^{\text{ext}}$ gives us $k'_c, sec(r_s) \in \mathcal{S}^{\text{ext}}$. In this setting as well, it is easy to show that DEX typing satisfies $((k' \sqcup sec(r_i)) \sqcup^{\text{ext}} sec(r_s)) \leq^{\text{ext}} k'_c$ because JVM typing satisfies $((k_2 \sqcup k_3) \sqcup^{\text{ext}} k_1) \leq^{\text{ext}} k_c$. For the rest of the constraints refer to the proof in **Getfield** as they are essentially the same (the constraint for 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 $sec(\vec{p}[0]) = k$, 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 $sec(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h$ since we know that the original JVM instruction satisfy $k \sqcup k_h \sqcup se(i) \leq k'_h$. We also know that $sec(\vec{p}[0]) \leq \vec{k'_a}[0]$ since $k \leq k'_a[0]$. 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 $\forall 0 \leq i \leq n.sec(\vec{p}[i]) \leq \vec{k'_a}[i]$. For the last constraint, we know that $\mathbf{excAnalysis}$ 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 $sec(\vec{p}[0]) \sqcup k_e = k \sqcup k_e$ which is the case in our setting. Therefore, we will have that constraint $\forall j \in \mathbf{region}(i, \text{Norm}).sec(\vec{p}[0]) \sqcup k_e \leq se(j)$ 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 $\forall j \in \mathbf{region}(i, e)$. $sec(\vec{p}[0]) \sqcup k'_r[e] \leq se(j)$. 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 $\mathbf{excAnalysis}$. Again as the class stays the same and that $\mathbf{excAnalysis}$ is the same, the satisfaction of $e \in \mathbf{excAnalysis}(m_{\text{ID}}) \sqcup \{\text{np}\}$ in JVM side implies the satisfaction of $e \in \mathbf{excAnalysis}(m') \sqcup \{\text{np}\}$ 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 $sec(\vec{p}[0]) \sqcup se(i) \sqcup \vec{k'_r}[e] \leq \vec{k_r}[e]$ in the DEX side since $sec(\vec{p}[0]) = k$ and everything else is the same.
  Actually there is a possibility that there is addition of $\mathbf{moveresult}$ and/or $\mathbf{moveexception}$, 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 $\mathbf{Invokevirtual}$ addressing the similar form of the constraints. In the case of caught exception case, the constraint $e \in \mathbf{classAnalysis}(i) \cup \{\text{np}\}$ is satisfied because, as before, $\mathbf{classAnalysis}$ and classes (e) are the same. So, if JVM program satisfy the constraint the translated DEX program will also satisfy it. The same with $\forall j \in \mathbf{region}(i, e) sec(r) \leq se(j)$ since $sec(r) = k$.
  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 $sec(r) \leq \vec{k_r}[e]$ since $sec(r) = k$ and policies stay the same.

■

Lemma (VII.6)**.** *Instructions within a block (sequential instructions) does not modify the flag, i.e.*

$$\forall bi, \{i, j\} \in bi \text{ s.t. } (bi, i) \mapsto (bi, j),$$
$$\exists rt, rt'.(bi, i) \vdash \langle F_{(bi,i)}, rt\rangle \Rightarrow \langle F_{(bi,i)}, rt'\rangle$$

Proof: There are a lot of instructions in DEX side which may target r_0 thus modifying the flag, but they are not translatable from JVM instructions. The only JVM instruction that may cause this modifying of the flag is $\mathbf{Store}$ instruction, which will be translated to $\mathbf{move}$ instruction. Since we already have the assumption that JVM instructions will not modify self-reference for any object, there will be no case such that $\mathbf{store}(0)$ ever appear, thus there will be no $\mathbf{move}(r_0, r_s)$ ever occur as well. ■

Lemma (VII.7)**.** *All blocks (except return block) will have flag* 0 *at its starting instruction, i.e.* $\forall bi \backslash \{Ret\}.F_{(bi,0)} = 0$ *where* Ret *indicates return block.*

Proof: We do this by induction on reachable blocks from block 0 because we know that block 0 will always start with the flag 0 ($F_{(0,0)} = 0$). Define block$_n$ as blocks that are minimally reachable with n steps. The base case is trivial, that is there are no other reachable block from block 0 apart from Ret.

For the induction step, we assume that all block$_n$ (excluding return block) have 0 as the starting flag (Induction Hypothesis). Let us focus our attention on a particular block in block$_n$. We proceed by using the previous lemma VII.6 and consider the possibility of the ending instructions. There are several possible instructions that can modify the flag, but the only possible instruction translated from JVM instructions which can modify the flag is those translated from $\mathbf{return}$ instruction. (in this case, we know that the successor block will be return block, which as we already assumed is not in the scope). Using lemma VII.6 and considering possible ending instruction we can establish that block$_n$ will not modify its starting flag so for block$_{n+1}$ the starting flag will be the same as that of block$_n$. Since by IH we have block$_n$'s starting flag is 0, we can establish that the starting flag of block$_{n+1}$ will be 0 as well. ■

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

Lemma (VII.8)**.** *Let* P *be a JVM program such that*

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

Then $[\![P]\!]$ *will 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

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

Proof: For the first property, they are mainly proved using Lemma VII.4 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.4, 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.2, and for throwing instructions we use the first case of Lemma VII.3.

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.2. Otherwise, we use Lemma VII.4.

■

Before we proceed to the proof of Lemma VII.9, 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.9)**.** *Let* $[\![P]\!]$ *be a typable DEX blocks resulted from translation of JVM instruction still in the block form, i.e.*

$$[\![P]\!] = \mathbf{Translate}(\mathbf{TraceParentChild}(\mathbf{StartBlock}(P)))$$

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

The operator $\oplus$ denotes the function where $\rho \oplus \{r \mapsto v\}$ means a new function ρ' 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_{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 $\mathbf{lookup_P}$ 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 $\mathbf{classAnalysis}$ 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 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 Γ 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 $\vec{k_a} \xrightarrow{k_h} \vec{k_r}$, 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 $\rightsquigarrow$, e.g. we may write $\rightsquigarrow$ instead of $\rightsquigarrow_{m,\text{Norm}}$ to mean the same thing. In the table of transfer rules, we may drop the superscript 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 $\Gamma, \mathbf{ft}, \mathbf{region}, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, se, i \vdash^{\tau} st \Rightarrow st'$.

$$\frac{P_m[i] = \mathbf{push}\ n}{\langle i, \rho, os\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, n :: os\rangle}$$

$$\frac{P_m[i] = \mathbf{pop}}{\langle i, \rho, v :: os\rangle \rightsquigarrow_{m,\text{Norm}} \langle i+1, \rho, os\rangle}$$

$$\frac{P_m[i] = \mathbf{return}}{\langle i, \rho, v :: os\rangle \rightsquigarrow_{m,\text{Norm}} v, h}$$

$$\frac{P_m[i] = \mathbf{goto}\ j}{\langle i, \rho, os\rangle \rightsquigarrow_{m,\text{Norm}} \langle j, \rho, os\rangle}$$

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

$$\frac{P_m[i] = \mathbf{ifeq}\ j \quad n = 0}{\langle i, \rho, n :: os\rangle \rightsquigarrow_{m,\text{Norm}} \langle j, \rho, os\rangle}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 9: Full JVM Operational Semantic

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 10: JVM Transfer Rule

APPENDIX D
FULL DEX OPERATIONAL SEMANTICS AND TRANSFER RULES

The following figure 11 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) ρ. The function $\mathbf{fresh} : \text{Heap} \to \mathcal{L}$ is an allocator function that given a heap returns the location for that object. The function $\mathbf{default} : \mathcal{C} \to \mathcal{O}$ returns for each class a default object of that class. For every field of that default object, the value will be 0 if the field is 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 $\rightsquigarrow$ relation which defines transition between state is $\rightsquigarrow \subseteq \mathbf{State} \times (\mathbf{State} + \mathcal{V} \times \mathbf{Heap})$.

The operator $\oplus$ denotes the function where $\rho \oplus \{r \mapsto v\}$ means a new function ρ' 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 12 is the full version of figure 6 in section IV. The full typing judgement takes the form of $\Gamma, \mathbf{ft}, \mathbf{region}, sgn, se, i \vdash^{\tau} \langle f, rt\rangle \Rightarrow \langle f', st'\rangle$ where Γ 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, $\langle f, rt\rangle$ is the flag and register typing for the current instruction, $\langle f', rt'\rangle$ is the flag and register typing after the instruction is executed.

The flag in the typing judgement is used to indicate whether r_0 has been used or not. It is mainly used because of how DEX treats return instruction. This introduction of flag causes some complexity in the transfer rules in that we have to also take into account when r_0 is used either as a target register or one of the sources registers. Some of the transfer rules will seem to be duplicate with each other because we also need to take care where does r_0 appear in the instruction (if it does appear).

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 $\rightsquigarrow$, e.g. we may write $\rightsquigarrow$ instead of $\rightsquigarrow_{m,\text{Norm}}$ to mean the same thing. In the table of transfer rules, we may drop the superscript 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 $\Gamma, \mathbf{ft}, \mathbf{region}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash^{\tau} st \Rightarrow st'$.

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

$$\frac{P[i]_m = \mathbf{ifeq}(r, j) \quad \rho(r) = 0}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \langle t, \rho, h\rangle}$$

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

$$\frac{P[i]_m = \mathbf{return}(r_s) \quad r_s \in \mathbf{dom}(\rho)}{\langle i, \rho, h\rangle \rightsquigarrow_{m,\text{Norm}} \rho(r_s), h}$$

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

$$\frac{P_m[i] = \mathbf{goto}(t)}{\langle i, \rho, h\rangle \rightsquigarrow \langle t, \rho, h\rangle}$$

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

$$\frac{P_m[i] = \mathbf{new}(r, c) \quad l = \mathbf{fresh}(h)}{\langle i, \rho, h\rangle \rightsquigarrow \langle i+1, \rho \oplus \{r \mapsto l\}, h \oplus \{l \mapsto \mathbf{default}(c)\}\rangle}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\mathbf{RuntimeExcHandling} : \mathbf{Heap} \times \mathcal{L} \times \mathcal{C} \times \mathcal{PP} \times (\mathcal{R} \rightharpoonup \mathcal{V}) \to \mathbf{State} + (\mathcal{L} \times \mathbf{Heap})$ 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. 11: DEX Operational Semantic

$$\frac{P_m[i] = \mathbf{const}(r, v) \quad r \notin locR}{se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i)\}\rangle} \qquad \frac{P_m[i] = \mathbf{const}(r_0, v)}{se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 1, rt \oplus \{r_0 \mapsto se(i)\}\rangle} \qquad \frac{P_m[i] = \mathbf{goto}(j)}{se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt\rangle}$$

$$\frac{P_m[i] = \mathbf{const}(r, v) \quad r \in locR\backslash\{r_0\} \quad se(i) \le \vec{k_a}(r)}{se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i)\}\rangle} \qquad \frac{P_m[i] = \mathbf{return}(r_s) \quad se(i) \sqcup rt(r_s) \le \vec{k_r}[n]}{\vec{k_a} \overset{k_h}{\to} \vec{k_r}, se, i \vdash \langle f, rt\rangle \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{move}(r_0, r_s) \quad r_s \neq r_0}{se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 1, rt \oplus \{r_0 \mapsto \big(sec(r_s) \sqcup se(i)\big)\}\rangle} \qquad \frac{P_m[i] = \mathbf{move}(r_0, r_0)}{se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r_0 \mapsto \big(sec(r_0) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{move}(r, r_0) \quad r \in locR\backslash\{r_0\} \quad sec(r_0) \sqcup se(i) \le \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 0, rt \oplus \{r \mapsto \big(sec(r_0) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{move}(r, r_s) \quad r \in locR\backslash\{r_0\} \quad r_s \neq r_0 \quad sec(r_s) \sqcup se(i) \le \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(sec(r_s) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{move}(r, r_0) \quad r \notin locR}{se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 0, rt \oplus \{r \mapsto \big(sec(r_0) \sqcup se(i)\big)\}\rangle} \qquad \frac{P[i] = \mathbf{move}(r, r_s) \quad r \notin locR \quad r_s \neq r_0}{se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(sec(r_s) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{binop}(op, r_0, r_a, r_b)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 1, rt \oplus \{r_0 \mapsto \big(sec(r_a) \sqcup sec(r_b) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b) \quad r_a = r_0 \ \mathbf{or} \ r_b = r_0 \quad r \in locR\backslash\{r_0\} \quad sec(r_a) \sqcup sec(r_b) \sqcup se(i) \le \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 0, rt \oplus \{r \mapsto \big(sec(r_a) \sqcup sec(r_b) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b) \quad r_a \neq r_0 \ \mathbf{and} \ r_b \neq r_0 \quad r \in locR\backslash\{r_0\} \quad sec(r_a) \sqcup sec(r_b) \sqcup se(i) \le \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(sec(r_a) \sqcup sec(r_b) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b) \quad r \notin locR \quad r_a = r_0 \ \mathbf{or} \ r_b = r_0}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 0, rt \oplus \{r \mapsto \big(sec(r_a) \sqcup sec(r_b) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{binop}(op, r, r_a, r_b) \quad r \notin locR \quad r_a \neq r_0 \ \mathbf{and} \ r_b \neq r_0}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto \big(sec(r_a) \sqcup sec(r_b) \sqcup se(i)\big)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{ifeq}(r_0, t) \quad \forall j' \in \mathbf{region}(i, \mathrm{Norm}), se(i) \sqcup sec(r) \le se(j')}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 0, \mathbf{lift}_{\mathbf{sec(r_0)}}(rt)\rangle}$$

$$\frac{P_m[i] = \mathbf{ifeq}(r, t) \quad r \neq r_0 \quad \forall j' \in \mathbf{region}(i, \mathrm{Norm}), se(i) \sqcup sec(r) \le se(j')}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec(r)}}(rt)\rangle}$$

$$\frac{P_m[i] = \mathbf{ifneq}(r_0, t) \quad \forall j' \in \mathbf{region}(i, \mathrm{Norm}), se(i) \sqcup sec(r) \le se(j')}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle 0, rt\rangle \Rightarrow \langle 0, \mathbf{lift}_{\mathbf{sec(r_0)}}(rt)\rangle}$$

$$\frac{P_m[i] = \mathbf{ifneq}(r, t) \quad r \neq r_0 \quad \forall j' \in \mathbf{region}(i, \mathrm{Norm}), se(i) \sqcup sec(r) \le se(j')}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec(r)}}(rt)\rangle}$$

$$\frac{P_m[i] = \mathbf{new}(r_0, c)}{se, i \vdash^{\mathrm{Norm}} \langle 0, rt\rangle \Rightarrow \langle 1, rt \oplus \{r_0 \mapsto se(i)\}\rangle} \qquad \frac{P_m[i] = \mathbf{new}(r, c) \quad r \notin locR}{se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{new}(r, c) \quad r \in locR \backslash \{r_0\} \quad se(i) \le \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i)\}\rangle}$$

$$\frac{P_m[i] = \mathbf{iget}(r_0, r_o, f) \quad sec(r_o) \in \mathcal{S} \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_o) \le se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle 0, rt\rangle \Rightarrow \langle 1, \mathbf{lift}_{\mathbf{sec(r_o)}}(rt \oplus \{r_0 \mapsto \big((sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f)\big)\})\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathbf{iget}(r, r_0, f) \quad sec(r_0) \in \mathcal{S} \quad r \in locR \backslash r_0 \quad (sec(r_0) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f) \le \vec{k_a}(r) \\ \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_0) \le se(j)\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle 0, rt\rangle \Rightarrow \langle 0, \mathbf{lift}_{\mathbf{sec(r_0)}}(rt \oplus \{r \mapsto \big((sec(r_0) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f)\big)\})\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathbf{iget}(r, r_o, f) \quad sec(r_o) \in \mathcal{S} \quad r \in locR \backslash \{r_0\} \quad r_o \ne r_0 \quad (sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f) \le \vec{k_a}(r) \\ \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_o) \le se(j)\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec(r_o)}}(rt \oplus \{r \mapsto \big((sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f)\big)\})\rangle}$$

$$\frac{P_m[i] = \mathbf{iget}(r, r_0, f) \quad sec(r_0) \in \mathcal{S} \quad r \notin locR \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_0) \le se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle 0, rt\rangle \Rightarrow \langle 0, \mathbf{lift}_{\mathbf{sec(r_0)}}(rt \oplus \{r \mapsto \big((sec(r_0) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f)\big)\})\rangle}$$

$$\frac{P_m[i] = \mathbf{iget}(r, r_o, f) \quad sec(r_o) \in \mathcal{S} \quad r \notin locR \quad r_o \ne r_0 \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_o) \le se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec(r_o)}}(rt \oplus \{r \mapsto \big((sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f)\big)\})\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathbf{iget}(r, r_o, f) \quad sec(r_o) \in \mathcal{S} \quad r \notin locR \ \mathbf{or}\ r = r_0 \quad \forall j \in \mathbf{region}(i, \mathrm{np}), sec(r_o) \le se(j) \\ \mathbf{Handler}(i, \mathrm{np}) = t\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto (sec(r_o) \sqcup se(i))\}\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathbf{iget}(r, r_o, f) \quad sec(r_o) \in \mathcal{S} \quad r \in locR \backslash \{r_0\} \quad (sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f) \le \vec{k_a}(r) \\ \forall j \in \mathbf{region}(i, \mathrm{np}), sec(r_o) \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) = t\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto (sec(r_o) \sqcup se(i))\}\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathbf{iget}(r, r_o, f) \quad sec(r_o) \in \mathcal{S} \quad r \notin locR \ \mathbf{or}\ r = r_0 \quad \forall j \in \mathbf{region}(i, \mathrm{np}), sec(r_o) \le se(j) \\ \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad sec(r_o) \le \vec{k_r}[\mathrm{np}]\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow}$$

$$\frac{\begin{array}{c}P_m[i] = \mathbf{iget}(r, r_o, f) \quad sec(r_o) \in \mathcal{S} \quad r \in locR \backslash \{r_0\} \quad (sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} \mathbf{ft}(f) \le \vec{k_a}(r) \\ \forall j \in \mathbf{region}(i, \mathrm{np}), sec(r_o) \le se(j) \quad \mathbf{Handler}(i, \mathrm{np}) \uparrow \quad sec(r_o) \le \vec{k_r}[\mathrm{np}]\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{np}} \langle f, rt\rangle \Rightarrow}$$

$$\frac{\begin{array}{c}P_m[i] = \mathbf{iput}(r_0, r_o, f) \quad sec(r_0) \in \mathcal{S}^{\mathrm{ext}} \quad sec(r_o) \in \mathcal{S} \quad (sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} sec(r_0) \le \mathbf{ft}(f) \quad k_h \le \mathbf{ft}(f) \\ \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_o) \le se(j)\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle 0, rt\rangle \Rightarrow \langle 0, \mathbf{lift}_{\mathbf{sec(r_o)}}(rt)\rangle}$$

$$\frac{\begin{array}{c}P_m[i] = \mathbf{iput}(r_s, r_o, f) \quad sec(r_s) \in \mathcal{S}^{\mathrm{ext}} \quad sec(r_o) \in \mathcal{S} \quad (sec(r_o) \sqcup se(i)) \sqcup^{\mathrm{ext}} sec(r_s) \le \mathbf{ft}(f) \quad k_h \le \mathbf{ft}(f) \\ r_s \ne r_0 \quad \forall j \in \mathbf{region}(i, \mathrm{Norm}), sec(r_o) \le se(j)\end{array}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec(r_o)}}(rt)\rangle}$$

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

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

$$\frac{P_m[i] = \mathbf{newarray}(r_0, r_l, t) \quad sec(r_l) \in \mathcal{S} \quad r \in locR \quad sec(r_l)[\mathbf{at}(i)] \leq \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt\rangle \Rightarrow \langle 1, rt \oplus \{r_0 \mapsto sec(r_l)[\mathbf{at}(i)]\}\rangle}$$

$$\frac{P_m[i] = \mathbf{newarray}(r, r_0, t) \quad sec(r_l) \in \mathcal{S} \quad r \in locR \backslash \{r_0\} \quad sec(r_0)[\mathbf{at}(i)] \leq \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt\rangle \Rightarrow \langle 0, rt \oplus \{r \mapsto sec(r_0)[\mathbf{at}(i)]\}\rangle}$$

$$\frac{P_m[i] = \mathbf{newarray}(r, r_l, t) \quad sec(r_l) \in \mathcal{S} \quad r \in locR \backslash \{r_0\} \quad sec(r_l)[\mathbf{at}(i)] \leq \vec{k_a}(r) \quad r_l \neq r_0}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto sec(r_l)[\mathbf{at}(i)]\}\rangle}$$

$$\frac{P_m[i] = \mathbf{newarray}(r, r_0, t) \quad sec(r_l) \in \mathcal{S} \quad r \notin locR}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt\rangle \Rightarrow \langle 0, rt \oplus \{r \mapsto sec(r_l)[\mathbf{at}(i)]\}\rangle}$$

$$\frac{P_m[i] = \mathbf{newarray}(r, r_l, t) \quad sec(r_l) \in \mathcal{S} \quad r \notin locR \quad r_l \neq r_0}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto sec(r_l)[\mathbf{at}(i)]\}\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r_0, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt\rangle \Rightarrow \langle 1, \mathbf{lift_k}(rt \oplus \{r_0 \mapsto k\})\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_0) \quad k[k_c] = sec(r_0) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \in locR \backslash \{r_0\} \quad k \leq \vec{k_a}(r) \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt\rangle \Rightarrow \langle 0, \mathbf{lift_k}(rt \oplus \{r \mapsto k\})\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \in locR \backslash \{r_0\} \quad k \leq \vec{k_a}(r) \quad r_a \neq r_0 \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift_k}(rt \oplus \{r \mapsto k\})\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_0) \quad k[k_c] = sec(r_0) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \notin locR \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt\rangle \Rightarrow \langle 0, \mathbf{lift_k}(rt \oplus \{r \mapsto k\})\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \notin locR \quad r_a \neq r_0 \quad \forall j \in \mathbf{region}(i, \text{Norm}), sec(r_a) \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt\rangle \Rightarrow \langle f, \mathbf{lift_k}(rt \oplus \{r \mapsto k\})\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \in locR \backslash \{r_0\} \quad k \leq \vec{k_a}(r) \quad r_a \neq r_0 \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto (k \sqcup se(i))\}\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \notin locR \textbf{ or } r = r_0 \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto (k \sqcup se(i))\}\rangle}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \in locR \backslash \{r_0\} \quad k \leq \vec{k_a}(r) \quad r_a \neq r_0 \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) \uparrow \quad k \leq \vec{k_a}[\text{np}]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{arraylength}(r, r_a) \quad k[k_c] = sec(r_a) \quad k \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \notin locR \textbf{ or } r = r_0 \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) \uparrow \quad k \leq \vec{k_a}[\text{np}]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{aget}(r_0, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt \rangle \Rightarrow \langle 1, \mathbf{lift_k}(rt \oplus \{r_0 \mapsto \big((se(i) \sqcup k \sqcup sec(r_i)\big) \sqcup^{\text{ext}} k_c)\})\rangle}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \in locR \backslash \{r_0\} \quad ((k \sqcup sec(r_i)) \sqcup^{\text{ext}} k_c) \leq \vec{k_a}(r) \quad r_a = r_0 \textbf{ or } r_i = r_0 \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt \rangle \Rightarrow \langle 0, \mathbf{lift_k}(rt \oplus \{r \mapsto \big((se(i) \sqcup k \sqcup sec(r_i)\big) \sqcup^{\text{ext}} k_c)\})\rangle}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \in locR \backslash \{r_0\} \quad ((k \sqcup sec(r_i)) \sqcup^{\text{ext}} k_c) \leq \vec{k_a}(r) \quad r_a \neq r_0 \textbf{ and } r_i \neq r_0 \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt \rangle \Rightarrow \langle f, \mathbf{lift_k}(rt \oplus \{r \mapsto \big((se(i) \sqcup k \sqcup sec(r_i)\big) \sqcup^{\text{ext}} k_c)\})\rangle}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \notin locR \quad r_a = r_0 \textbf{ or } r_i = r_0 \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt \rangle \Rightarrow \langle 0, \mathbf{lift_k}(rt \oplus \{r \mapsto \big((se(i) \sqcup k \sqcup sec(r_i)\big) \sqcup^{\text{ext}} k_c)\})\rangle}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \notin locR \quad r_a \neq r_0 \textbf{ and } r_i \neq r_0 \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt \rangle \Rightarrow \langle f, \mathbf{lift_k}(rt \oplus \{r \mapsto \big((se(i) \sqcup k \sqcup sec(r_i)\big) \sqcup^{\text{ext}} k_c)\})\rangle}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \notin locR \textbf{ or } r = r_0 \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto (k \sqcup se(i))\}\rangle}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \in locR \backslash \{r_0\} \quad ((k \sqcup sec(r_i)) \sqcup^{\text{ext}} k_c) \leq \vec{k_a}(r) \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto (k \sqcup se(i))\}\rangle}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \notin locR \text{ or } r = r_0 \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) \uparrow \quad k \leq \vec{k_r}[\text{np}]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{aget}(r, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c \in \mathcal{S}^{\text{ext}} \quad r \in locR \backslash \{r_0\} \quad ((k \sqcup sec(r_i)) \sqcup^{\text{ext}} k_c) \leq \vec{k_a}(r) \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) \uparrow \quad k \leq \vec{k_r}[\text{np}]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c, sec(r_s) \in \mathcal{S}^{\text{ext}} \quad r_s = r_0 \text{ or } r_i = r_0 \quad ((k \sqcup sec(r_i)) \sqcup^{\text{ext}} sec(r_s)) \leq^{\text{ext}} k_c \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt \rangle \Rightarrow \langle 0, \mathbf{lift_k}(rt) \rangle}$$

$$\frac{P_m[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c, sec(r_s) \in \mathcal{S}^{\text{ext}} \quad r_s \neq r_0 \text{ and } r_i \neq r_0 \quad ((k \sqcup sec(r_i)) \sqcup^{\text{ext}} sec(r_s)) \leq^{\text{ext}} k_c \quad \forall j \in \mathbf{region}(i, \text{Norm}), k \leq se(j)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt \rangle \Rightarrow \langle f, \mathbf{lift_k}(rt) \rangle}$$

$$\frac{P_m[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c, sec(r_s) \in \mathcal{S}^{\text{ext}} \quad r_s = r_0 \text{ or } r_i = r_0 \quad ((k \sqcup sec(r_i)) \sqcup^{\text{ext}} sec(r_s)) \leq^{\text{ext}} k_c \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow \langle f, \vec{k_a} \oplus \{ex \mapsto (k \sqcup se(i))\} \rangle}$$

$$\frac{P_m[i] = \mathbf{aput}(r_s, r_a, r_i) \quad k[k_c] = sec(r_a) \quad k, sec(r_i) \in \mathcal{S} \quad k_c, sec(r_s) \in \mathcal{S}^{\text{ext}} \quad r_s = r_0 \text{ or } r_i = r_0 \quad ((k \sqcup sec(r_i)) \sqcup^{\text{ext}} sec(r_s)) \leq^{\text{ext}} k_c \quad \forall j \in \mathbf{region}(i, \text{np}), k \leq se(j) \quad \mathbf{Handler}(i, \text{np}) \uparrow \quad k \leq \vec{k_r}[\text{np}]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{np}} \langle f, rt \rangle \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{moveresult}(r_0)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt \rangle \Rightarrow \langle 1, rt \oplus \{r_0 \mapsto se(i) \sqcup rt(ret)\} \rangle}$$

$$\frac{P_m[i] = \mathbf{moveresult}(r) \quad r \in locR \backslash \{r_0\} \quad se(i) \sqcup rt(ret) \leq \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt \rangle \Rightarrow \langle f, rt \oplus \{r \mapsto se(i) \sqcup rt(ret)\} \rangle}$$

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

$$\frac{P_m[i] = \mathbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[sec(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\to} \vec{k'_r} \quad sec(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall 0 \leq i < n. sec(\vec{p}[i]) \leq \vec{k'_a}[i] \quad k_e = \bigsqcup \{\vec{k'_r}[e] \mid e \in \mathbf{excAnalysis}(m')\} \quad \forall j \in \mathbf{region}(i, \text{Norm}), sec(\vec{p}[0]) \sqcup k_e \leq se(j) \quad r_0 \in \vec{p}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle 0, rt \rangle \Rightarrow \langle 0, \mathbf{lift}_{\mathbf{sec}(\tilde{\mathbf{p}}[\mathbf{0}]) \sqcup \mathbf{k_e}}(rt \oplus \{ret \mapsto \vec{k'_r}[n] \sqcup se(i)\}) \rangle}$$

$$\frac{P_m[i] = \mathbf{invoke}(n, m', \vec{p}) \quad \Gamma_{m'}[sec(\vec{p}[0])] = \vec{k'_a} \overset{k'_h}{\to} \vec{k'_r} \quad sec(\vec{p}[0]) \sqcup k_h \sqcup se(i) \leq k'_h \quad \forall 0 \leq i < n. sec(\vec{p}[i]) \leq \vec{k'_a}[i] \quad k_e = \bigsqcup \{\vec{k'_r}[e] \mid e \in \mathbf{excAnalysis}(m')\} \quad \forall j \in \mathbf{region}(i, \text{Norm}), sec(\vec{p}[0]) \sqcup k_e \leq se(j) \quad r_0 \notin \vec{p}}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\text{Norm}} \langle f, rt \rangle \Rightarrow \langle f, \mathbf{lift}_{\mathbf{sec}(\tilde{\mathbf{p}}[\mathbf{0}]) \sqcup \mathbf{k_e}}(rt \oplus \{ret \mapsto \vec{k'_r}[n] \sqcup se(i)\}) \rangle}$$

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

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

$$\frac{P_m[i] = \mathbf{throw}(r_0) \quad e \in \mathbf{classAnalysis}(i) \cup \{\mathrm{np}\} \quad \forall j \in \mathbf{region}(i,e), sec(r) \le se(j) \quad \mathbf{Handler}(i,e) = t}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^e \langle 0, rt\rangle \Rightarrow \langle 0, rt \oplus \{ex \mapsto (sec(r) \sqcup se(i))\}\rangle}$$

$$\frac{P_m[i] = \mathbf{throw}(r) \quad e \in \mathbf{classAnalysis}(i) \cup \{\mathrm{np}\} \quad \forall j \in \mathbf{region}(i,e), sec(r) \le se(j) \quad \mathbf{Handler}(i,e) = t \quad r \ne r_0}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^e \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{ex \mapsto (sec(r) \sqcup se(i))\}\rangle}$$

$$\frac{P_m[i] = \mathbf{throw}(r_0) \quad e \in \mathbf{classAnalysis}(i) \cup \{\mathrm{np}\} \quad \forall j \in \mathbf{region}(i,e), sec(r) \le se(j) \quad \mathbf{Handler}(i,e) \uparrow \quad sec(r) \le \vec{k_r}[e]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^e \langle 0, rt\rangle \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{throw}(r) \quad e \in \mathbf{classAnalysis}(i) \cup \{\mathrm{np}\} \quad \forall j \in \mathbf{region}(i,e), sec(r) \le se(j) \quad \mathbf{Handler}(i,e) \uparrow \quad r \ne r_0 \quad sec(r) \le \vec{k_r}[e]}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^e \langle f, rt\rangle \Rightarrow}$$

$$\frac{P_m[i] = \mathbf{moveexception}(r_0)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle 0, rt\rangle \Rightarrow \langle 1, rt \oplus \{r \mapsto (rt(ex) \sqcup se(i))\}\rangle}$$

$$\frac{P_m[i] = \mathbf{moveexception}(r) \quad r \in locR \backslash \{r_0\} \quad se(i) \sqcup rt(ex) \le \vec{k_a}(r)}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto (rt(ex) \sqcup se(i))\}\rangle}$$

$$\frac{P_m[i] = \mathbf{moveexception}(r) \quad r \notin locR}{\Gamma, \mathbf{ft}, \vec{k_a} \overset{k_h}{\to} \vec{k_r}, \mathbf{region}, se, i \vdash^{\mathrm{Norm}} \langle f, rt\rangle \Rightarrow \langle f, rt \oplus \{r \mapsto (rt(ex) \sqcup se(i))\}\rangle}$$

Fig. 12: 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 $h \in \text{Heap}$ 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 $h, h_0 \in \text{Heap}$, 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 $h, h_0 \in \text{Heap}$ and $\mathbf{ft}(f) \not\leq k_{\text{obs}}$, $h \sim_\beta h_0$ implies $h \oplus \{l \mapsto h(l) \oplus \{f \mapsto v\}\} \sim_\beta h_0$*

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

Lemma E.5. *For all heap $h, h', h_0 \in \text{Heap}$, $k \not\leq k_{\text{obs}}$, 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 $rt_1(r) = rt_2(r), rt_1(r) \leq k_{\text{obs}}$ and $\rho_1(r) \sim_\beta \rho_2(r)$*
- *or $rt_1(r) \not\leq k_{\text{obs}}$ and $rt_2(r) \not\leq k_{\text{obs}}$*

APPENDIX F
PROOF THAT TYPABLE $\text{DEX}_\mathcal{I}$ 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 ($\text{DEX}_\mathcal{I}$), object and array instructions ($\text{DEX}_\mathcal{O}$), method invocation ($\text{DEX}_\mathcal{C}$), and exception mechanism ($\text{DEX}_\mathcal{G}$).

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 $rt \in (\mathcal{R} \rightharpoonup \mathcal{S})$ be a registers typing; we write $\mathbf{high}(\rho, rt)$ if ρ and rt have the same domain and $rt(x) \not\leq k_{\text{obs}}$ for every $x \notin locR$.

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 β 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. $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$, 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 $\text{DEX}_\mathcal{I}$ states at the same program point i and let two registers types $rt_1, rt_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ such that $s_1 \sim_{\vec{k_a}, rt_1, rt_2} s_2$.*

- *Let $s_1', s_2' \in \text{State}_I$ and $rt_1', rt_2' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ 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 $k_r \leq k_{\text{obs}}$ 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 ρ_1 and ρ_2 to have different values. We will do this by case for possible instructions :

- $\mathbf{move}(r, r_s)$. This instruction itself also depends whether the target register is a local variable or not.
  - $r \in locR \backslash \{r_0\}$. This case can be a problem only if the instruction tries to move high values ($secr_s \not\leq k_{\text{obs}}$) to register with low security level ($\vec{k_a}(r) \leq k_{\text{obs}}$). But the transfer rule already forbid it ($sec(r_s) \leq \vec{k_a}(r)$). In another words since the instruction at program point i can never cause $\rho_{s_1'} \not\sim \rho_{s_2'}$, therefore a plain contradiction.
  - $r \notin locR$ or $r = r_0$. 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 $sec(r_s) \sqcup se(i)$, where $sec(r_s) \not\leq k_{\text{obs}}$), thus preserving the indistinguishability.
- $\mathbf{binop}(r, r_a, r_b)$. Following the argument from **move**, there are two main cases whether the target register is a local variable or not.
  - $r \in locR \backslash \{r_0\}$. This case can be a problem only if the instruction tries do a binary operation on distinct values and put the result on a low register, i.e. either $\rho_1(r_a) \neq \rho_2(r_a)$ or $\rho_2(r_a) \neq \rho_2(r_b)$ and $sec(r) \leq k_{\text{obs}}$. Assume that this is the case. Since we already assumed that $\rho_1 \sim \rho_2$, this means that either $sec(r_a) \leq k_{\text{obs}}$ and $sec(r_b) \not\leq k_{\text{obs}}$, or $sec(r_a) \not\leq k_{\text{obs}}$ and $sec(r_b) \leq k_{\text{obs}}$. The transfer rule that applies to this case have a constraint $sec(r_a) \sqcup sec(r_b) \leq sec(r)$, therefore $sec(r)$ can not be low ($sec(r) \not\leq k_{\text{obs}}$). A contradiction.

- $r \notin locR$ or $r = r_0$. This case is trivial, as 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 $sec(r_a) \sqcup sec(r_b) \sqcup se(i)$, where $sec(r_a) \not\leq k_{\text{obs}}$ and/or $sec(r_b) \not\leq k_{\text{obs}}$), thus preserving the indistinguishability.

- $\mathbf{const}(r, v)$. 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.

- $\mathbf{goto}(j)$. Nothing to prove here, as the instruction only modify the program counter.

- $\mathbf{return}(r_s)$. 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 ($sec(r_s) \not\leq k_{\text{obs}}$), but the security level of the return value of the method is low ($k_r \leq k_{\text{obs}}$). But this is already taken care of by the transfer rule which state $sec(r_s) \leq k_r$. Therefore, a contradiction.

- $\mathbf{ifeq}(r, t)$. 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 ($sec(r) \leq k_{\text{obs}}$), 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 ($sec(r) \not\leq k_{\text{obs}}$), we know by the transfer rule that rt'_1 and rt'_2 will be lifted to the upper bound of the two registers ($\mathbf{lift_{sec(r)}}$), which is high thus yielding $\mathbf{high}(\rho, rt)$) and $\mathbf{high}(\rho', rt')$. Therefore, register wise these two states are indistinguishable.

■

Lemma F.2 (Step Consistent)**.** *Let $\langle i, \rho\rangle, s_0 \in \text{State}_I$ be two DEX$_\mathcal{I}$ states and two registers types $rt, rt_0 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ such that $\langle i, \rho\rangle \sim_{\vec{k_a}, rt, rt_0} s_0$, $se(i) \not\leq k_{\text{obs}}$ and $\mathbf{high}(\rho, rt)$.*

- *If there exists a state $\langle i', \rho'\rangle \in \text{State}_I$ and a registers type $rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t. $\langle i, \rho\rangle \rightsquigarrow \langle i', \rho'\rangle$ and $i \vdash rt \Rightarrow rt'$, then $\langle i', \rho'\rangle \sim_{\vec{k_a}, rt', rt_0} s_0$.*

- *If there exists a value $v \in V$ s.t. $\langle i, \rho\rangle \rightsquigarrow v$ and $i \vdash rt \Rightarrow$ then $k_r \not\leq k_{\text{obs}}$*

Proof: By contradiction. Assume that all the precedent are true, but the conclusion is false. That means, $\langle i', \rho'\rangle$ is distinguishable from s_0, which means that $\rho' \not\sim \rho_0$. We will do this by case for possible instructions :

- $\mathbf{move}(r, r_s)$. Again we differentiate the case between whether the target register is in $locR$.
  - $r \in locR \backslash \{r_0\}$. This instruction will cause the problem if the target register is low ($\vec{k_a}(r) \leq k_{\text{obs}}$) and at least one of the source is high ($sec(r_a) \not\leq k_{\text{obs}}$ and/or $sec(r_b) \not\leq k_{\text{obs}}$). According to the transfer rules it is not possible as we already have the constraint that $sec(r_a) \sqcup sec(r_b) \leq \vec{k_a}(r)$, which means that the security level of the target register must be high if the security level of at least one of the source register is high, therefore a contradiction.
  - $r \notin locR$. In this case, it's trivial because the security level of r will at least be as high as $se(i)$ thus we will have $\mathbf{high}(\rho', rt')$ and combining with the assumption $\mathbf{high}(\rho_0, rt_0)$, we have $\rho' \sim \rho_0$

- $\mathbf{binop}(r, r_a, r_b)$. Follows from the **move** instruction
  - $r \in locR$. This instruction will cause the problem if the target register is low ($\vec{k_a}(r) \leq k_{\text{obs}}$) and the source is high $sec(r_s) \not\leq k_{\text{obs}}$. According to the transfer rules it is not possible as we already have the constraint that $sec(r_s) \leq \vec{k_a}(r)$, which means that the security level of the target register must be high if the security level of the source register is high, therefore a contradiction.
  - $r \notin locR$. In this case, it's trivial because the security level of r will at least be as high as $se(i)$ thus we will have $\mathbf{high}(\rho', rt')$ and combining with the assumption $\mathbf{high}(\rho_0, rt_0)$, we have $\rho' \sim \rho_0$

- $\mathbf{const}(r, v)$. Follows from assumption that $se(i)$ is high.

- $\mathbf{goto}(j)$. Nothing to prove here, as the instruction only modify the program counter.

- $\mathbf{ifeq}(r, j)$. Nothing to prove here, as the instruction only modify the program counter and we assumed that $\mathbf{high}(\rho, rt)$ which will give us $\mathbf{high}(\rho', rt')$ and combined with $\mathbf{high}(\rho_0, rt_0)$ will give us the indistinguishability.

- $\mathbf{return}(r_s)$. In this instance, we only need to make sure that the returned value is high (i.e. $k_r \not\leq k_{\text{obs}}$) if the assumption holds. This is taken care of by the transfer rule, which states that $se(i) \leq k_r$ which means that since $se(i)$ is high (by assumption), the returned value must high as well ($k_r \not\leq k_{\text{obs}}$) to be typable ($i \vdash rt \Rightarrow$).

■

Lemma F.3 (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 $rt_1, rt_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ 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 $rt'_1, rt'_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t. $i_1 \neq i_2$, $s_1 \rightsquigarrow \langle i_1, \rho'_1\rangle$, $s_2 \rightsquigarrow \langle i_2, \rho'_2\rangle$, $i \vdash rt_1 \Rightarrow rt'_1$, $i \vdash rt_2 \Rightarrow rt'_2$ then $\mathbf{high}(\rho'_1, rt'_1)$, $\mathbf{high}(\rho'_2, rt'_2)$ and $\forall j \in \mathbf{region}(i), se(j) \not\leq k_{\text{obs}}$.*

Proof: This is already by definition of the branching instruction (**ifeq** and **ifneq**). The instruction will lift the

registers types rt'_1 and rt'_2 at least as high as the registers r ($\mathbf{lift_{sec}}$, where $sec = sec_1(r) = sec_2(r)$), and $se(i)$ with this level as well. 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. After the lifting of registers typing for rt'_1 and rt'_2, we will have $\mathbf{high}(\rho'_1, rt'_1)$ and $\mathbf{high}(\rho'_2, rt'_2)$. Since se is high for scope of the region, we have $\forall j \in \mathbf{region}(i), se(j) \not\leq k_{\text{obs}}$. ∎

Lemma F.4 (High Step)**.** *Let $\langle i, \rho\rangle, \langle i', \rho'\rangle \in \text{State}_I$ be two DEX$_\mathcal{I}$ states and let two registers types $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ such that $\langle i, \rho\rangle \rightsquigarrow \langle i', \rho'\rangle$, $i \vdash rt \Rightarrow rt'$, $se(i) \not\leq k_{\text{obs}}$ and $\mathbf{high}(\rho, rt)$ then $\mathbf{high}(\rho', rt')$.*

Proof: By contradiction. Assume the precedent, but we have rt' is not high i.e. $\exists r \notin locR.rt'(r) \not\leq k_{\text{obs}}$. This can only be the case if the instruction moves a value to one of the register, and that value is low. But, since all the possible instructions ($\mathbf{const}$, $\mathbf{move}$, $\mathbf{bop}$) have $se(i)$ as lub and we already assumed that $se(i)$ is high ($se(i) \not\leq k_{\text{obs}}$), therefore a contradiction. ∎

Lemma F.5 (high registers type sub-typing)**.** *if rt is a high registers type and $rt \sqsubseteq rt'$ then rt' is high*

Lemma F.6 (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.7 (indistinguishablility single monotony)**.** *if $s \sim_{k_{\text{obs}}, S, T} t, S \sqsubseteq S'$ and S is high then $s \sim_{k_{\text{obs}}, S', T} t$*

Having established the appropriated lemmas, we sketch the proof of type system soundness. This proof follows closely to the one in the original paper. In the induction step (base case is that if it's typable, then no direct flows), we have two executions $s_0 \rightsquigarrow ... \rightsquigarrow s_n$ and $s'_0 \rightsquigarrow ... \rightsquigarrow s'_m$ s.t. $\text{pc}(s_0) = \text{pc}(s'_0)$ and $s_0 \sim_{RT_{\text{pc}(s_0)}, RT_{\text{pc}(s'_0)}} s'_0$ and we want to establish that states s_n and s'_m are indistinguishable: $s'_n \sim_{RT_{\text{pc}(s_n)}, RT_{\text{pc}(s'_m)}} s'_m$ or both registers types $RT_{\text{pc}(s_n)}$ and $RT_{\text{pc}(s'_m)}$ are high.

We assume the property holds for any strictly shorter execution paths (induction hypothesis) and suppose $n > 0$ and $m > 0$. We note $i_0 = \text{pc}(s_0) = \text{pc}(s'_0)$. Using Lemma F.1 and typability hypothesis, we have $s_1 \sim_{rt, rt'} s'_1$ for some registers types rt and rt' such that $i_0 \vdash RT_{i_0} \Rightarrow rt$, $rt \sqsubseteq RT_{\text{pc}(s_1)}, i_0 \vdash RT_{i'_0} \Rightarrow rt'$, $rt' \sqsubseteq RT_{\text{pc}(s'_1)}$.

- If $\text{pc}(s_1) = \text{pc}(s'_1)$ we can apply the Lemma F.6 to establish that $s_1 \sim_{RT_{\text{pc}(s_1)}, RT_{\text{pc}(s'_1)}} s'_1$ and conclude by induction hypothesis (now we already have shorter execution path $s_1 \rightsquigarrow ... \rightsquigarrow s_n$ and $s'_1 \rightsquigarrow ... \rightsquigarrow s'_m$).

- If $\text{pc}(s_1) \neq \text{pc}(s'_1)$ (it is from a branching point, and it is high, as the low will fall into previous case) we know by the Lemma F.3 that se is high in region $\mathbf{region}(i_0)$ and rt and rt' are high. Lemma F.5 implies that both $RT_{\text{pc}(s_1)}$ and $RT_{\text{pc}(s'_1)}$ are high. By $SOAP1$ we know that $\text{pc}(s_1) \in \mathbf{region}(i_0)$ or $\text{pc}(s_1) = \mathbf{jun}(i_0)$, then going for induction on the path $s_1 \rightsquigarrow ... \rightsquigarrow s_n$ (using SOAP2) we know that either there exists $k, 1 \leq k \leq n$ s.t. $k = \mathbf{jun}(i_0)$ and using the help of Lemma F.2 to get $s_k \sim_{RT_{\text{pc}(s_k)}, RT_{i_0}} s'_0$ (high path reaches junction point) or $\text{pc}(s_n) \in \mathbf{region}(i_0)$ and $RT_{\text{pc}(s_n)}$ is high (the high path stays in the region). Both relies on Lemma F.4 and Lemma F.7. A similar property holds for the other execution path. We can group two cases here :

- $\mathbf{jun}(i_0)$ is defined and there exists k, k' s.t. $1 \leq k \leq n$, $1 \leq k' \leq m$ s.t. $k = k' = \mathbf{jun}(i_0)$ and $s_k \sim_{RT_{\text{pc}(s_k)}, RT_{i_0}} s'_0$, $s_0 \sim_{RT_{i_0}, RT_{\text{pc}(s'_{k'})}} s'_{k'}$. Since $s_0 \sim_{RT_{i_0}, RT_{i_0}} s'_0$ we have by transitivity and symmetry of $\sim$, $s_k \sim_{RT_{\text{pc}(s_k)}, \text{pc}(s'_{k'})} s'_{k'}$, with $\text{pc}(s_k) = \text{pc}(s'_{k'})$ and we can conclude by induction hypothesis.
- $\mathbf{jun}(i_0)$ is undefined and both $RT_{\text{pc}(s_n)}$ and $RT_{\text{pc}(s'_m)}$ are high

To reduce clutter, we only formally prove non-interference of the last submachine where all the features are included. The proof will resemble the proof sketched in this submachine and can be appropriated for another submachines as well.

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 $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$, 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 β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \text{State}_O$ be two DEX$_\mathcal{O}$ states at the same program point i and let two registers types $rt_1, rt_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ 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 $rt'_1, rt'_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ such that $s_1 \rightsquigarrow s'_1, s_2 \rightsquigarrow s'_2$, $i \vdash rt_1 \Rightarrow rt'_1$, and $i \vdash rt'_2 \Rightarrow rt'_2$, then there exists $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $s'_2 \sim_{\vec{k_a}, rt'_1, rt'_2, \beta'} s'_2$ and $\beta \sqsubseteq \beta'$.*
- *Let $v_1, v_2 \in \mathcal{V}$ such that $s_1 \rightsquigarrow v_1$, $s_2 \rightsquigarrow v_2$, $i \vdash rt_1 \Rightarrow$, and $i \vdash rt'_2 \Rightarrow$, then $k_r \leq k_{\text{obs}}$ implies $v_1 \sim_\beta v_2$.*

Proof: 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 ρ'_1, h'_1 are parts of s'_1 and ρ'_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 $\mathbf{iput}, \mathbf{newarray}$, and $\mathbf{aput}$.
  - $\mathbf{iput}(r_s, r_o, f)$ 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 $\mathbf{iput}$ 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 $sec = sec_1(r_s) = sec_2(r_s)$. A plain contradiction.
  - $\mathbf{aput}(r_s, r_a, r_i)$ 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 $sec = sec_1(r_s) = sec_2(r_s)$. A plain contradiction.
    - $\mathbf{newarray}(r_a, r_l, t)$ can only cause the difference by creating array of different lengths ($\rho_1(r_l) \neq \rho_2(r_l)$) with $sec_1(r_s) \not\leq k_{\text{obs}}$ and $sec_2(r_s) \not\leq k_{\text{obs}}$. But if that's the case, then that means this new array does not have to be included in the mapping β' 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 ρ_1' and ρ_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 ($h_1 \sim h_2$). There are two possible cases depending on which register is used as the target register:
    - $r \in locR \backslash \{r_0\}$. The only possible scenario where it will cause a problem is when the source value have different value (thus have high security level), and the target register is low. Assume that it is the case. But the constraint of the three instructions always mention that the security value of the target register have to be at least as high as the source, therefore it can not be the case that $\vec{k_a}(r) \leq k_{\text{obs}}$ while the security level of the source value is high. A contradiction.
    - $r \notin locR$ or $r = r_0$. In this case, 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 (Step Consistent)**.** *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, let $\langle i, \rho, h\rangle, \langle i_0, \rho_0, h_0\rangle \in \text{State}_O$ be two DEX$_\mathcal{O}$ states and two registers types $rt, rt_0 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ such that $\langle i, \rho, h\rangle \sim_{\vec{k_a}, rt, rt_0, \beta} \langle i_0, \rho_0, h_0\rangle$, $se(i) \not\leq k_{\text{obs}}$ and $\mathbf{high}(\rho, rt)$.*

- *If a state $\langle i', \rho', h'\rangle \in \text{State}_O$ and a registers type $rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t. $\langle i, \rho, h\rangle \rightsquigarrow \langle i', \rho', h'\rangle$ and $i \vdash rt \Rightarrow rt'$, then $\langle i', \rho', h'\rangle \sim_{\vec{k_a}, rt', rt_0} \langle i_0, \rho_0, h_0\rangle$.*
- *If there exists a value $v \in V$ and $h' \in \mathbf{Heap}$ s.t. $\langle i, \rho, h\rangle \rightsquigarrow v, h'$ and $i \vdash rt \Rightarrow$ then $h' \sim_\beta h_0$ and $k_r \not\leq k_{\text{obs}}$*

Proof: By contradiction. Assume that all the precedent are true, but the conclusion is false. That means, $\langle i', \rho', h'\rangle$ is distinguishable from $\langle i_0, \rho_0, h_0\rangle$, which means either $\rho' \not\sim \rho_0$ or $h' \not\sim h_0$.

- assume $h \not\sim h_0$. This can be the case only if the instruction at i are either $\mathbf{new}, \mathbf{iput}, \mathbf{newarray}$, or $\mathbf{aput}$.
    - **newarray** and **newinstance** are proved directly by Lemma E.6.
    - **iput** is proved directly by lemma E.3 because we have the constraint $se(i) \leq \mathbf{ft}(f)$ in the transfer rule, so it implies that $\mathbf{ft}(f) \not\leq k_{\text{obs}}$.
    - $\mathbf{aput}(r_s, r_a, r_i)$ is proved directly by lemma E.4 because we have the constraint $se(i) \leq sec(\rho(r_a))$ in the transfer rule, so it implies that $sec(\rho(r_a)) \not\leq k_{\text{obs}}$.
- assume $\rho_{s'} \not\sim \rho_t$. This is only the case for possible instructions : **iget**, **aget**, and **arraylength**. There are two cases based on the target register to put the value r.
    - $r \in locR \backslash \{r_0\}$. We already have by the assumption that se is high, and for each of these instructions the value put will be we have the constraint that $se(i) \leq \vec{k_a}(r_t)$. Therefore, to be typable these instructions will have the target register to be high, thus maintaining indistinguishability
    - $r \notin locR$. For all the instructions we know that the value put here will be lub-ed with $se(i)$ which is high, therefore maintaining the indistinguishability.

■

Lemma G.3 (High Branching)**.** *Let β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, $s_1, s_2 \in \text{State}_O$ be two DEX$_\mathcal{O}$ states at the same program point i and let two registers types $rt_1, rt_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ 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 $rt_1', rt_2' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t. $i_1 \neq i_2, s_1 \rightsquigarrow \langle i_1, \rho_1', h_1'\rangle$, $s_2 \rightsquigarrow \langle i_2, \rho_2', h_2'\rangle$. If $i \vdash rt_1 \Rightarrow rt_1', i \vdash rt_2 \Rightarrow rt_2'$ then $\mathbf{high}(\rho_1', rt_1')$, $\mathbf{high}(\rho_2', rt_2')$ and $\forall j \in \mathbf{region}(i), se(j) \not\leq k_{\text{obs}}$*

Lemma G.4 (High Step)**.** *Let $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle \in \text{State}_O$ be two DEX$_\mathcal{O}$ states and let two registers types $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ such that $\langle i, \rho, h\rangle \rightsquigarrow \langle i', \rho', h'\rangle$, $i \vdash rt \Rightarrow rt', se(i) \not\leq k_{\text{obs}}$ and $\mathbf{high}(\rho, rt)$ then $\mathbf{high}(\rho', rt')$.*

Proof: By contradiction. Assume the precedent, but we have rt' is not high i.e. $\exists r \notin locR. rt'(r) \not\leq k_{\text{obs}}$. This can only be the case if the instruction moves a value to one of the register, and that value is low. But, for all the possible instructions it is an impossible case, therefore a contradiction. For both **iget** and **aget** we already have $se(i)$ as lub and we already assumed that $se(i)$ is high ($se(i) \not\leq k_{\text{obs}}$). ■

APPENDIX H
PROOF THAT TYPABLE 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* $\langle i,\rho,h\rangle, \langle i',\rho',h'\rangle \in \mathrm{State}_C$ *be two states s.t.* $\langle i,\rho,h\rangle\rangle \rightsquigarrow_m \langle i',\rho',h'\rangle$. *Let two registers types* $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ *s.t.* $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash^{\mathrm{Norm}} rt \Rightarrow rt'$ *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* $\langle i,\rho,h\rangle \in \mathrm{State}_C$ *be a state,* $h' \in \mathbf{Heap}$*, and* $v \in V$ *s.t.* $\langle i,\rho,h\rangle \rightsquigarrow v, h'$. *Let* $rt \in (\mathcal{R} \rightharpoonup \mathcal{S})$ *s.t.* $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash^{\mathrm{Norm}} rt \Rightarrow$*, 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* $\langle i,\rho,h\rangle, \langle i',\rho',h'\rangle \in \mathrm{State}_C$ *be two states s.t.* $\langle i,\rho,h\rangle \rightsquigarrow_m \langle i',\rho',h'\rangle$. *Let two registers types* $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ *s.t.* $\mathbf{region}$*,* se*,* $\vec{k_a} \overset{k_h}{\to} k_r$*,* $i \vdash^{\mathrm{Norm}} rt \Rightarrow rt'$ *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

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

where $m \rightsquigarrow m'$ 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 $m_0 \rightsquigarrow m_1$ is dropped from the call chain, we know that **invoke** on m_1 will fulfill *side-effect-safety*. Since all possible instructions 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 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* Γ *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,* $s_1, s_2 \in \mathrm{State}_C$ *two DEX*$_\mathcal{C}$ *states at the same program point* i *and two registers types* $rt_1, rt_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ *s.t.* $s_1 \sim_{k_{\mathrm{obs}}, rt_1, rt_2, \beta} s_2$. *If there exist two states* $s_1'.s_2' \in \mathrm{State}_C$ *and two registers types* $rt_1', rt_2' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ *s.t.*

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

and

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

then there exists $\beta' \in L \rightharpoonup L$ *s.t.* $s_1' \sim_{k_{\mathrm{obs}}, rt_1', rt_2', \beta'} s_2'$ *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 $\rho_1' \not\sim \rho_2'$ or $h_1' \not\sim h_2'$, where ρ_1', h_1' are parts of s_1' and ρ_2', h_2' are parts of s_2'.

- Assume $h_1' \not\sim h_2'$. $\mathbf{invoke}(m, n, \vec{p})$ 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_{obs} (let's say this register x). By the transfer rules of **invoke**, this will imply that the $\vec{k_a'}[x] \not\leq k_{\mathrm{obs}}$ (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 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 $\rho_{s'} \not\sim \rho_{t'}$. **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 DEX$_\mathcal{I}$), except that the source is now the pseudo-register ret.

∎

Lemma H.6 (Step Consistent Lemma)**.** *Let P be a program and a table of signature Γ s.t. all its method m' are* side-effect-safe *w.r.t. the heap effect level of all the policies in* $\mathbf{Policies}_{\mathbf{\Gamma}}(m')$*. Let m a method in P, β a partial function $\beta \in L \rightharpoonup L, \langle i, \rho, h\rangle, s_0 \in \mathrm{State}_C$ two DEX$_\mathcal{C}$ states, and two registers types $rt, rt_0 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t. $\langle i, \rho, h\rangle \sim_{k_{\mathrm{obs}}, rt, rt_0, \beta} s_0, se(i) \not\leq k_{\mathrm{obs}}$ and $\mathbf{high}(\rho, rt)$. If there exists a state $\langle i', \rho', h'\rangle \in \mathrm{State}_C$ and a registers types $rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

$$\langle i, \rho, h\rangle \rightsquigarrow_m \langle i', \rho', h'\rangle \qquad \Gamma, \mathrm{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash rt \Rightarrow rt'$$

then $\langle i', \rho', h'\rangle \sim_{k_{\mathrm{obs}}, rt', rt_0, \beta} s_0$.

Proof: By contradiction. Assume that all the precedent are true, but the conclusion is false. That means, $\langle i', \rho', h'\rangle$ is distinguishable from $\langle i_0, \rho_0, h_0\rangle$, which means either $\rho' \not\sim \rho_0$ or $h' \not\sim h_0$. Assume $h' \not\sim h_0$. **invoke** is proved directly by lemma E.5 therefore it can not be the case. Assume $\rho' \not\sim \rho_0$. **invoke** only modifies the pseudo-register ret in which case the transfer rule states that the value put there is lub-ed with $se(i) \not\leq k_{\mathrm{obs}}$, therefore we know that register ret is indistinguishable. As for **moveresult**, we follow the arguments of **move**, except that the source is now the pseudo-register ret. ∎

Lemma H.7 (High Step Lemma)**.** *Let m a method, let two DEX$_\mathcal{C}$ states $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle$ and two registers types $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

$$\langle i, \rho, h\rangle \rightsquigarrow_m \langle i', \rho', h'\rangle \quad se(i) \not\leq k_{\mathrm{obs}} \qquad \Gamma, \mathrm{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash rt \Rightarrow rt' \quad \textit{and} \quad \mathbf{high}(\rho, rt)$$

then $\mathbf{high}(\rho', rt')$.

Proof: By contradiction. Assume the precedent, but we have rt' is not high i.e. $\exists r \in R. rt'(r) \not\leq k_{\mathrm{obs}}$. For **invoke**, this may cause a problem if the result put into ret is low. But we know from the assumption that $se(i)$ is high and the transfer rule states that the value put into ret is lub-ed with $se(i)$, therefore the value will be high as well. Same reasoning goes for **moveresult**, where the value will be lub-ed with $se(i)$ which is high. ∎

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

Like the one in DEX$_\mathcal{C}$, we also need to firstly prove the *side-effect-safety* of a program if it's typable. Fortunately, this proof extends almost directly from the one in DEX$_\mathcal{C}$. The only difference is that there is a possibility 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 $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle \in \mathrm{State}_G$ be two states s.t. $\langle i, \rho, h\rangle \rightsquigarrow_m \langle i', \rho', h'\rangle$. Let two registers types $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t. $\mathbf{region}, se, \vec{k_a} \overset{k_h}{\to} k_r, i \vdash rt \Rightarrow rt'$ and $P[i] \neq \mathbf{invoke}$, then $h \preceq_{k_h} h'$*

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

Proof: This can only be one of two cases, either it is **return** instruction or 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 $\mathbf{Policies}_{\mathbf{\Gamma}}(m)$. Let $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle \in \mathrm{State}_G$ be two states s.t. $\langle i, \rho, h\rangle \rightsquigarrow_m \langle i', \rho', h'\rangle$. Let two registers types $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ 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 $\mathbf{Policies}_{\mathbf{\Gamma}}(m)$. Then all method m are* side-effect-safe *w.r.t. the heap effect level of all the policies in $\mathbf{Policies}_{\mathbf{\Gamma}}(m)$.*

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

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

there exists $RT \in \mathcal{PP} \to (\mathcal{R} \rightharpoonup \mathcal{S})$ and $rt_1, \ldots rt_n \in (\mathcal{R} \rightharpoonup \mathcal{S})$ 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 $\mathbf{highResult}_{\vec{\mathbf{k_r}}}(r, h)$ is defined as :*

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

Lemma I.5 (High Step)**.** *Let $\langle i, \rho, h\rangle, \langle i', \rho', h'\rangle \in \mathrm{State}_G$ be two DEX$_\mathcal{G}$ states and two registers types $rt, rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

$$\langle i, \rho, h\rangle \rightsquigarrow_{m, \tau} \langle i', \rho', h'\rangle \quad se(i) \not\leq k_{\mathrm{obs}} \qquad i \vdash rt \Rightarrow rt' \quad \mathbf{high}(\rho, rt)$$

then

$$\mathbf{high}(\rho', rt')$$

Proof: Since we already assumed that $se(i) \not\leq k_{\text{obs}}$, this means that for any value put into the registers in the stack space will also be high, thus maintaining the $\mathbf{high}(\rho', rt')$. This behaviour is due to the transfer rules which lub the security with $se(i)$ which is high. The only thing to note is that $\mathbf{lift}_k$ operation is point wise least upper bound with k, therefore if we already have $\mathbf{high}(\rho, rt)$, it imply $\mathbf{high}(\rho, \mathbf{lift}_k(rt))$ as well. ■

Lemma I.6 (Step Consistent)**.** *Let β a partial function $\beta \in L \rightharpoonup L$, let $\langle i, \rho, h\rangle$, $\langle i_0, \rho_0, h_0\rangle \in \text{State}_G$ be two DEX$_\mathcal{G}$ states and two registers types $rt, rt_0 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

$$\langle i, \rho, h\rangle \sim_{rt, rt_0, \beta} \langle i_0, \rho_0, h_0\rangle \qquad se(i) \not\leq k_{\text{obs}} \qquad \mathbf{high}(\rho, rt)$$

1) *If there exists a state $\langle i', \rho', h'\rangle \in \text{State}_G$, a tag $\tau \in \{\text{Norm}\} + C$ and a registers type $rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

$$\langle i, \rho, h\rangle \rightsquigarrow_{m,\tau} \langle i', \rho', h'\rangle \qquad i \vdash^{\tau} rt \Rightarrow rt'$$

then

$$\langle i', \rho', h'\rangle \sim_{rt', rt_0, \beta} \langle i_0, \rho_0, h_0\rangle$$

2) *If there exists a result $v \in V + L$, a heap h', and a tag $\tau \in \{\text{Norm}\} + C$ s.t.*

$$\langle i, \rho, h\rangle \rightsquigarrow_{m,\tau} v, h' \qquad i \vdash^{\tau} rt \Rightarrow$$

then

$$\mathbf{highResult}_{\vec{\mathbf{k_r}}}(v, h') \text{ and } h' \sim_\beta h_0$$

Proof: Here, we only concern the instruction that may throw an exception. There are two main cases here :

- throw a caught exception : In this case, we have we have three main groupings : caught null exception, **invoke** with a caught exception, and **throw** with a caught exception. In each of these case, we have the transfer rule that the new mapping to ex will be lub-ed with $se(i)$ which is high, therefore resulting in $\mathbf{high}(\rho', rt')$. Thus concluding the first case.
- throw an uncaught exception : Similar to the previous case, in this case the constraint that for any exception e, a security level will be lub-ed with $se(i)$ and compared against $\vec{k_r}[e]$. Thus if the instruction is typable and $se(i) \not\leq k_{\text{obs}}$, it is necessary that $\mathbf{highResult}_{\vec{\mathbf{k_r}}}(v, h')$.

■

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

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

Recall that in the proofs for type system soundness, there is this one part which relies on the step consistent, but not in isolation. We need the step consistent because we need the characteristic of execution in high region. Now that we distinguish such region also based on tag, we now need more lemmas to establish this characteristics.

Lemma I.7 (Iterated Step Consistent)**.** *Let β a partial function $\beta \in L \rightharpoonup L$, $\langle i_0, \rho_0, h_0\rangle, \langle i, \rho, h\rangle \in \text{State}_G$ two states of DEX$_\mathcal{G}$ and a registers type $rt \in (\mathcal{R} \rightharpoonup \mathcal{S})$. s.t.*

$$\langle i_0, \rho_0, h_0\rangle \sim_{RT_{i_0}, rt, \beta} \langle i, \rho, h\rangle \qquad \mathbf{high}(\rho_0, RT_{i_0})$$

Let $i \in PP$ and $\tau \in \{\text{Norm}\} + C$ s.t.

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

Suppose we have a derivation

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

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

1) $\mathbf{jun}(i, \tau)$ *is defined and there exists j with $0 < j \leq k$ s.t.*

$$i_j = \mathbf{jun}(i, \tau), \langle i_j, \rho_j, h_j\rangle \sim_{RT_{i_j}, rt, \beta} \langle i, \rho, h\rangle$$

and

$$\mathbf{high}(\rho_j, RT_{i_j})$$

2) $\mathbf{jun}(i, \tau)$ *is undefined, $k \in \mathbf{region}(i, \tau)$, $h' \sim_\beta h$ and* $\mathbf{highResult}_{\vec{\mathbf{k_r}}}(v, h')$

Proof: By induction on k.

- $k = 0$ we can use SOAP3 to conclude that $\mathbf{jun}(i, \tau)$ is undefined ($i_0 \in \mathbf{region}(i, \tau)$ and i_0 is a return point) and apply case 2 of Lemma I.6 to conclude the rest of case 2
- suppose the statement is true for a given k and try to prove it for $k + 1$. From the assumption that $i_0 \in \mathbf{region}(i, \tau)$ we have that $se(i_0) \not\leq k_{\text{obs}}$. Combining with the assumption that $\mathbf{high}(\rho_0, RT_{i_0})$ and from the typability definition ($i_0 \vdash RT_{i_0} \Rightarrow rt'_1$ hold for some $rt'_1 \sqsubseteq RT_{i_1}$), so we have

  $$\langle i_1, \rho_1, h_1\rangle \sim_{rt'_1, rt, \beta} \langle i, \rho, h\rangle \quad \text{and} \quad \mathbf{high}(\rho_1, rt'_1)$$

  by case 1 of Lemma I.5 and Lemma I.6. By Lemma F.5 and Lemma F.7, we have

  $$\langle i_1, \rho_1, h_1\rangle \sim_{RT_{i_1}, rt, \beta} \langle i, \rho, h\rangle \quad \text{and} \quad \mathbf{high}(\rho_1, RT_{i_1})$$

  Then, using SOAP2, we have $i_1 \in \mathbf{region}(i, \tau)$ or $\mathbf{jun}(i, \tau)$. In the case where $i_1 \in \mathbf{region}(i, \tau)$, it is sufficient to invoke induction hypothesis on derivation

  $$\langle i_1, \rho_1, h_1\rangle \rightsquigarrow_{m,\tau_1} ...\langle i_{k+1}, \rho_{k+1}, h_{k+1}\rangle \rightsquigarrow_{m,\tau_{k+1}} (v, h')$$

  to conclude. In the case where $i_1 = \mathbf{jun}(i, \tau)$, we can conclude we are in case 1 by taking $j = 1$

because we know that $\langle i_1, \rho_1, h_1\rangle \sim_{RT_{i_1}, rt, \beta} \langle i, \rho, h\rangle$ and $\mathbf{high}(\rho_1, RT_{i_1})$ hold.

■

Since now we have different regions based on tags, there is a possibility that an execution starts from a junction point in some region, but still contained in the other region (thus the step consistent still applies). This lemma is used for the proof for execution starting from a high branching.

Lemma I.8 (Iterated Step Consistent on Junction Point)**.** *Let β a partial function $\beta \in L \rightharpoonup L$ and $\langle i_0, \rho_0, h_0\rangle, \langle i'_0, \rho'_0, h'_0\rangle \in \mathrm{State}_G$ two DEX$_\mathcal{G}$ states s.t.*

$$\langle i_0, \rho_0, h_0\rangle \sim_{RT_{i_0}, RT_{i'_0}, \beta} \langle i'_0, \rho'_0, h'_0\rangle$$
$$\mathbf{high}(\rho_0, RT_{i_0}) \text{ and } \mathbf{high}(\rho'_0, RT_{i'_0})$$

Let $i \in PP$ and $\tau, \tau' \in \{\mathrm{Norm}\} + C$ s.t.

$$i_0 \in \mathbf{region}(i, \tau) \qquad \text{se is high in region } \mathbf{region}(i, \tau)$$
$$i'_0 \in \mathbf{jun}(i, \tau') \qquad \text{se is high in region } \mathbf{region}(i, \tau')$$

Suppose we have a derivation

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

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

$$\langle i'_0, \rho'_0, h'_0\rangle \rightsquigarrow_{m,\tau_0} \ldots \langle i'_k, \rho'_k, h'_k\rangle \rightsquigarrow_{m,\tau'_k} (r', h')$$

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

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

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

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

Proof: We first invoke Lemma I.7 on the derivation

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

and the region $\mathbf{region}(i, \tau)$ and so we will have two cases :

1) There exists j, with $0 < j \le k$ s.t. $i_j = \mathbf{jun}(i, \tau)$ and $\langle i_j, \rho_j, h_j\rangle \sim_{RT_{i_j}, RT_{i'_0}, \beta} \langle i'_0, \rho'_0, h'_0\rangle$. If $\mathbf{jun}(i, \tau) = \mathbf{jun}(i, \tau')$ we are in case 1 with $j' = 0$. Otherwise, we know that the program satisfies SOAP4: $\mathbf{jun}(i, \tau) \in \mathbf{region}(i, \tau')$ or $\mathbf{jun}(i, \tau') \in \mathbf{region}(i, \tau)$.
   - If $i_j = \mathbf{jun}(i, \tau) \in \mathbf{region}(i, \tau')$, we invoke Lemma I.7 on the derivation $\langle i_j, \rho_j, h_j\rangle \rightsquigarrow_{m,\tau_j} \ldots (r, h)$ and the region $\mathbf{region}(i, \tau')$ to establish that there exists q with $j < q \le k$ s.t. $i_q = \mathbf{jun}(i, \tau') = i'_0$ and $\langle i_q, \rho_q, h_q\rangle \sim_{RT_{i_q}, RT_{i'_0}, \beta} \langle i'_0, \rho'_0, h'_0\rangle$ and we can conclude we are in case 1 with $j = q$ and $j' = 0$. Since we know that the program satisfies SOAP3, the case where $k \in \mathbf{region}(i, \tau')$ is not possible because then a junction point is defined for a return point.
   - If $i'_0 = \mathbf{jun}(i, \tau') \in \mathbf{region}(i, \tau)$, we invoke Lemma I.7 on the derivation $\langle i'_0, \rho'_0, h'_0\rangle \rightsquigarrow_{m,\tau'_0} \ldots (r', h')$ and the region $\mathbf{region}(i, \tau)$ to establish that there exists j' with $0 < j' \le k'$ s.t. $i_{j'} = \mathbf{jun}(i, \tau) = i_j$ and $\langle i_{j'}, \rho_{j'}, h_{j'}\rangle \sim_{RT_{i_{j'}}, RT_{i_0}, \beta} \langle i_0, \rho_0, h_0\rangle$ and we can conclude we are in case 1. Since we know that the program satisfies SOAP3, the case where $k' \in \mathbf{region}(i, \tau)$ is not possible because then a junction point is defined for a return point.
2) $\mathbf{jun}(i, \tau)$ is undefined, $k \in \mathbf{region}(i, \tau)$, $\mathbf{highResult}_{\tilde{\mathbf{k_r}}}(r, h)$ and $h \sim_\beta h'_0$. k is a return point in region $\mathbf{region}(i, \tau)$ so, thanks to SOAP5, we know that $i'_0 = \mathbf{jun}(i, \tau') \in \mathbf{region}(i, \tau)$. We can hence invoke Lemma I.7 lemma on the derivation $\langle i'_0, \rho'_0, h'_0\rangle \rightsquigarrow_{m,\tau'_0} \ldots (r', h')$ and the region $\mathbf{region}(i, \tau)$. Since we know that $\mathbf{jun}(i, \tau)$ is undefined, it will be the case that $k' \in \mathbf{region}(i, \tau)$, $\mathbf{highResult}_{\tilde{\mathbf{k_r}}}(r', h')$, $h' \sim_\beta h_0$ and we can conclude we are in case 2.

■

Lemma I.9 (Iterated Step Consistent After Branching)**.** *Let β a partial function $\beta \in L \rightharpoonup L$ and $\langle i_0, \rho_0, h_0\rangle, \langle i'_0, \rho'_0, h'_0\rangle \in \mathrm{State}_G$ two DEX$_\mathcal{G}$ states such that*

$$\langle i_0, \rho_0, h_0\rangle \sim_{RT_{i_0}, RT_{i'_0}, \beta} \langle i'_0, \rho'_0, h'_0\rangle$$
$$\mathbf{high}(\rho_0, rt_0) \qquad \mathbf{high}(\rho'_0, rt'_0)$$

Let $i \in PP$ and $\tau, \tau' \in \{\mathrm{Norm}\} + C$ s.t.

$$i \mapsto^\tau i_0 \text{ and } i \mapsto^{\tau'} i'_0$$

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

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

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

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

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

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

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

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

Proof: If $i_0 = i'_0$ then case 1 trivially holds. If $i_0 \neq i'_0$ then, using SOAP1 two times (for each i and i'), we distinguish four cases:

1) $i_0 \in \mathbf{region}(i, \tau)$ and $i'_0 \in \mathbf{region}(i, \tau')$. We first invoke Lemma I.7 on the derivation $\langle i_0, \rho_0, h_0\rangle \rightsquigarrow_{m,\tau_0} \ldots (r, h)$ and the region $\mathbf{region}(i, \tau)$ to obtain two possibilities:
   (a) $\exists j$, with $0 < j \le k$ s.t. $i_j = \mathbf{jun}(i, \tau)$ and $\langle i_j, \rho_j, h_j\rangle \sim_{RT_{i_j}, RT_{i'_0}, \beta} \langle i'_0, \rho'_0, h'_0\rangle$
   (b) $k \in \mathbf{region}(i, \tau)$, $\mathbf{highResult}_{\tilde{\mathbf{k_r}}}(r, h)$ and $h \sim_\beta h'_0$.

In the case (a), we know that it falls to case 1 by invoking Lemma I.8. In the case (b), we invoke Lemma I.7 on the derivation $\langle i'_0, \rho'_0, h'_0\rangle \rightsquigarrow_{m,\tau'_0} ...(r', h')$ and the region $\mathbf{region}(i, \tau')$. We again obtain two cases. If there exists j', with $0 < j' \leq k'$ s.t. $i_{j'} = \mathbf{jun}(i, \tau')$ and $\langle i_{j'}, \rho_{j'}, h_{j'}\rangle \sim_{RTi_{j'}, RTi_0, \beta} \langle i_0, \rho_0, h_0\rangle$, we can conclude we are in case 1 thanks to Lemma I.8. In the second case $k' \in \mathbf{region}(i, \tau'), \mathbf{highResult}_{\tilde{\mathbf{k_r}}}(r', h')$ and $h' \sim_\beta h_0$. We hence have

$$\begin{cases} \mathbf{highResult}_{\tilde{\mathbf{k_r}}}(r, h) \\ \mathbf{highResult}_{\tilde{\mathbf{k_r}}}(r', h') \\ h' \sim_\beta h_0 \\ h \sim_\beta h'_0 \\ h_0 \sim_\beta h'_0 \qquad \textit{assumption} \end{cases}$$

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

2) $i_0 = \mathbf{jun}(i, \tau)$ and $i'_0 \in \mathbf{region}(i, \tau')$. We can conclude by Lemma I.8.

3) $i'_0 = \mathbf{jun}(i, \tau')$ and $i_0 \in \mathbf{region}(i, \tau)$. We can conclude by Lemma I.8.

4) $i_0 = \mathbf{jun}(i, \tau)$ and $i'_0 = \mathbf{jun}(i, \tau')$. If $\mathbf{jun}(i, \tau) = \mathbf{jun}(i, \tau')$ we are in case 1. If not, SOAP4 applies : $i_0 = \mathbf{jun}(i, \tau) \in \mathbf{region}(i, \tau')$ or $i'_0 = \mathbf{jun}(i, \tau') \in \mathbf{region}(i, \tau)$. In both cases, we can conclude by Lemma I.8.

■

Corollary of Lemma I.6.

Lemma I.10 (Final Step Consistent)**.** *Let β a partial function $\beta \in L \rightharpoonup L$, $\langle i, \rho, h\rangle \in \mathrm{State}_G$ a DEX$_G$ state, $h_0 \in \mathrm{Heap}$ a heap and a registers type $rt \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

$$h \sim_\beta h_0 \qquad se(i) \notin k_{\mathrm{obs}} \qquad \mathbf{high}(\rho, rt)$$

1) *If there exists a state $\langle i', \rho', h'\rangle \in \mathrm{State}_G$, a tag $\tau \in \{\mathrm{Norm}\} + C$ and a registers type $rt' \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

$$\langle i, \rho, h\rangle \rightsquigarrow_{m,\tau} \langle i', \rho', h'\rangle \qquad i \vdash^\tau rt \Rightarrow rt'$$

then

$$\mathbf{high}(\rho', rt') \textit{ and } h' \sim_\beta h_0$$

2) *If there exists a result $r \in V + L$, a heap $h' \in \mathrm{Heap}$ and a tag $\tau \in \{\mathrm{Norm}\} + C$ s.t.*

$$\langle i, \rho, h\rangle \rightsquigarrow_{m,\tau} (r, h') \qquad i \vdash^\tau rt \Rightarrow$$

then

$$\mathbf{highResult}_{\tilde{\mathbf{k_r}}}(r, h') \textit{ and } h' \sim_\beta h_0$$

Lemma I.11 (Iterated Final Step Consistent After Branching)**.** *Let β a partial function $\beta \in L \rightharpoonup L$, $\langle i_0, \rho_0, h_0\rangle \in \mathrm{State}_G$ a DEX$_\mathcal{G}$ states and $h'_0 \in \mathrm{Heap}$ s.t.*

$$h_0 \sim_\beta h'_0 \qquad \mathbf{high}(\rho_0, RT_{i_0})$$

Let $i \in PP$ and $\tau, \tau' \in \{\mathrm{Norm}\} + C$ s.t.

$$i_0 \in \mathbf{region}(i, \tau) \textit{ and } i \mapsto^{\tau'}$$

Suppose that se is high in region $\mathbf{region_m}(i, \tau)$. Suppose we have a derivation

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

and suppose this derivation is typable w.r.t. RT. Then

$$\mathbf{highResult}_{\tilde{\mathbf{k_r}}}(r, h) \textit{ and } h \sim_\beta h'_0$$

Proof: By induction on k.

- $k = 0$ we can directly apply case 2 of Lemma I.10 to conclude.
- we suppose the statement is true for a given k and we prove it now for $k + 1$. First note that $se(i_0) \notin k_{\mathrm{obs}}$, $\mathbf{high}(\rho_0, RT_{i_0})$ and $i_0 \vdash RT_{i_0} \Rightarrow rt_1$ hold for some rt_1 s.t. $rt_1 \sqsubseteq RT_{i_1}$, so we have

  $$h_1 \sim_\beta h'_0 \text{ and } \mathbf{high}(\rho_1, rt_1)$$

  by case 1 of Lemma I.10. By subtyping lemma we have also:

  $$\mathbf{high}(\rho_1, RT_{i_1})$$

  Then, using SOAP2, we have $i_1 \in \mathbf{region}(i, \tau)$ or $i_1 = \mathbf{jun}(i, \tau)$. In the first case, it is sufficient to invoke induction hypothesis on derivation

  $$\langle i_1, \rho_1, h_1\rangle \rightsquigarrow_{m,\tau_1} ...\langle i_{k+1}, \rho_{k+1}, h_{k+1}\rangle \rightsquigarrow_{m,\tau_{k+1}} (r, h)$$

  to conclude. The second case is an impossible case because we know that the program satisfies SOAP3 therefore $\mathbf{jun}(i, \tau)$ is undefined.

■

Lemma I.12 (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, $s_1, s_2 \in \mathrm{State}_G$ and two registers types $rt_1, rt_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

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

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

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} \langle i'_1, \rho'_1, h'_1\rangle & s_2 \rightsquigarrow_{m,\tau_2} \langle i'_2, \rho'_2, h'_2\rangle \\ i \vdash^{\tau_1} rt_1 \Rightarrow rt'_1 & i \vdash^{\tau_2} rt_2 \Rightarrow rt'_2 \end{array}$$

then

$$\begin{array}{lll} \mathbf{high}(\rho'_1, rt'_1) & \text{and} & \textit{se is high in } \mathbf{region}(i, \tau_1) \\ \mathbf{high}(\rho'_2, rt'_2) & \text{and} & \textit{se is high in } \mathbf{region}(i, \tau_2) \end{array}$$

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

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} \langle i'_1, \rho'_1, h'_1\rangle & s_2 \rightsquigarrow_{m,\tau_2} (r_2, h'_2) \\ i \vdash^{\tau_1} rt_1 \Rightarrow rt'_1 & i \vdash^{\tau_2} rt_2 \Rightarrow \end{array}$$

then

$$\mathbf{high}(\rho'_1, rt'_1) \quad \text{and} \quad \textit{se is high in } \mathbf{region}(i, \tau_1)$$

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 $\vec{k'_a} \overset{k'_h}{\rightarrow} \vec{k'_r}$. 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. Now for the registers typing, we know that for normal executing one the registers typing will be lifted to k_e (which we know is high) thus yielding $\mathbf{high}(\rho'_1, rt'_1)$. For the exception throwing one, if the exception is caught, then we know we will be in the first case. From the transfer rule, we know that the only register which determine $\mathbf{high}(\rho'_2, rt'_2)$ is ex, which will be set to $sec(\vec{p}[0]) \sqcup \vec{k'_r}[e']$, thus yielding $\mathbf{high}(\rho'_2, rt'_2)$. 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 $\vec{k'_a} \overset{k'_h}{\rightarrow} \vec{k'_r}$. 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 $\mathbf{high}(\rho'_1, rt'_1)$. If e_2 is caught, using the same argument we will have $\mathbf{high}(\rho'_2, rt'_2)$ 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 $\mathbf{region}(i_1, \text{np})$. Note we will also $\mathbf{high}(\rho'_1, rt'_1)$ since the only register to determine it is ex, which will have the value lub-ed with the security level of the register pointing to object / array reference. Now, regarding the part which is not null, we also can deduce that it is $\mathbf{high}(\rho'_2, rt'_2)$ because of the transfer rules (the registers types are lifted to at least as high as object / array reference, which is $\not\leq k_{\text{obs}}$). And then also from the transfer rules we have the condition that se will be high in that region, which implies that se will be high in $\mathbf{region}(i_2, \text{Norm})$

- **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.13 (Locally Respect)**.** *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, $s_1, s_2 \in \text{State}_G$ two DEX$_\mathcal{G}$ states at the same program point i and two registers types $rt_1, rt_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t. $s_1 \sim_{rt_1,rt_2,\beta} s_2$.*

1) *If there exists two states $s'_1, s'_2 \in \text{State}_G$ and two registers types $rt'_1, rt'_2 \in (\mathcal{R} \rightharpoonup \mathcal{S})$ s.t.*

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} s'_1 & s_2 \rightsquigarrow_{m,\tau_2} s'_2 \\ i \vdash^{\tau_1} rt_1 \Rightarrow rt'_1 & i \vdash^{\tau_2} rt_2 \Rightarrow rt'_2 \end{array}$$

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

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

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

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} \langle i'_1, \rho'_1, h'_1 \rangle & s_2 \rightsquigarrow_{m,\tau_2} (r_2, h'_2) \\ i \vdash^{\tau_1} rt_1 \Rightarrow rt'_1 & i \vdash^{\tau_2} rt_2 \Rightarrow \end{array}$$

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

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

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

$$\begin{array}{cc} s_1 \rightsquigarrow_{m,\tau_1} (r_1, h'_1) & s_2 \rightsquigarrow_{m,\tau_2} (r_2, h'_2) \\ i \vdash^{\tau_1} rt_1 \Rightarrow & i \vdash^{\tau_2} rt_2 \Rightarrow \end{array}$$

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

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

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.

- **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 we fall in to the first one, where we have two successor state and they are indistinguishable. This is possible only if the security level of at least one of the arguments (including the object) is high.
  - if the object is high, we know that by the transfer rules that the normal execution will have its registers types lifted to high, and the pseudo-register containing exception will have security level lub-ed with the object security

level (which is high), therefore they are indistinguishable (both are high).

- if at least one of the argument is high, but the object is low. We know that this method invoked m' is non-interferent, therefore will have its outputs indistinguishable. One of the clause for output indistinguishability for one normal and one exception (let's say e) is that $\vec{k'_r}[e] \not\leq k_{\text{obs}}$, which will mean the upper bound of these security levels k_e in the transfer rule will be at least this high as well. Since the registers types of the normal execution is lifted to this level and ex is lub-ed with $\vec{k'_r}[e]$, they are indistinguishable (both are high).

2) One normal and one has uncaught exception (the case where one throw caught exception and one throw uncaught exception is subsumed by this one, as we already proved that normal execution and caught exception are indistinguishable). 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 $\mathbf{highResult_{k_r}}(r_2, h'_2)$ (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 $\vec{k_r}[e] \not\leq k_{\text{obs}}$ which in turns implies that $\mathbf{highResult_{k_r}}(r_2, h'_2)$.
3) Both has caught exception (and different exception on top of that). Again we appeal to the output indistinguishability of the method m'. Since they are indistinguishable, therefore the two exceptions have to be high, which will implies that $\rho'_1 \sim_{rt'_1, rt'_2, \beta'} \rho'_2$ since rt'_1 and rt'_2 only contains ex which is high.
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 $\vec{k'_r}[e_1] \not\leq k_{\text{obs}}$ and $\vec{k'_r}[e_2] \not\leq k_{\text{obs}}$. 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.

- **iget**. There are four cases to consider here:
1) One is normal execution and one has caught null exception. In this case, as usual we can appeal to the transfer rules for this particular instruction. Since this case can only happens only if r_o (the register pointing to the object) have different value, that means $rt_1(r_o) \not\leq k_{\text{obs}}$ and $rt_2(r_o) \not\leq k_{\text{obs}}$. The transfer rule for normal execution says that the registers types will be lifted to at least this value as well, i.e. $\mathbf{high}(\mathbf{lift_{sec(r_o)}}(rt))$. And for the caught null exception, the registers types will be $\{\vec{k_a}, ex \mapsto sec(r_o) \sqcup se(i)\}$ (high). Therefore, since both of the registers types are high, they are indistinguishable. Now for the heap, we use lemma E.2 to establish that $h'_1 \sim_{\beta'} h'_2$, since one heap stays while the other heap only added with new location ($h'_1 \oplus \{l \mapsto \mathbf{default}_{\text{np}}\} \sim_{\beta'} h'_2$, and vice versa as this is symmetric case)
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 ($\mathbf{highResult_{k_r}}(r_2, h'_2)$). We know that r_o (the register containing the object) is high ($sec(r_o) \not\leq k_{\text{obs}}$), otherwise $s_1 \not\sim_{rt_1, rt_2 \beta} s_2$. The transfer rule for **iget** with uncaught exception states that $sec(r_o) \leq \vec{k_r}[\text{np}]$, which will give us $\vec{k_r}[\text{np}] \not\leq k_{\text{obs}}$, which will implies that $\mathbf{highResult_{k_r}}(r_2, h'_2)$.
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 ($l_1 = \mathbf{fresh}(h_1)$, $l_1 \mapsto \mathbf{default}_{\text{np}}$, $l_2 = \mathbf{fresh}(h_2)$, $l_2 \mapsto \mathbf{default}_{\text{np}}$), 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 β' 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 $\rho'_2 = \{ex \mapsto l_2\}$.
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 $(\langle l_1 \rangle, h'_1) \sim_{\beta'} (\langle l_2 \rangle, h'_2)$.

- **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 $\rho'_1 \sim_{\beta'} \rho'_2$. 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.
2) Two different exceptions, both are caught. We can appeal directly to the transfer rule. We know that the register containing the exception is high, and since the register will now only contains the ex register and it is high, therefore the registers are indistinguishable.
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.
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 $\mathbf{highResult_{k_r}}(r_2, h'_2)$, $h'_1 \sim_{\beta'} h'_2$ (throw instruction does not modify the heap), and $\beta \subseteq \beta'$.

■

After we defined all the lemmas we need, we proceed

to the soundness proof, i.e. typable DEX bytecode is non-interferent. Suppose we have β a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and $\langle i_0, \rho_0, h_0\rangle, \langle i_0', \rho_0', h_0'\rangle \in \text{State}_G$ two $\text{DEX}_\mathcal{G}$ 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.13. There are several possible cases for the induction step:

1) $k > 0$ and $k' = 0$: then we can use case 2 of Lemma I.13 to get existence of $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ s.t.

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

   Using case 2 of Lemma I.12 and Lemma F.5 we get

   $$\mathbf{high}(\rho_1, RT_{i_1}) \text{ and } se \text{ is high in } \mathbf{region}(i_0, \tau_1)$$

   where τ_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 caseis rendered impossible due to SOAP3. Applying Lemma I.11 we get

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

   Combining [1] and [2] we get

   $$h_1 \sim_{\beta'} h_1', \quad h_1 \sim_{\beta'} h', \mathbf{highResult}_{\mathbf{k_a}}(r', h')$$

   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.13 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.12 and Lemma F.5 we have

   $$\mathbf{high}(\rho_1, RT_{i_1}) \quad \text{and} \quad se \text{ is high in } \mathbf{region}(i_0, \tau)$$
   $$\mathbf{high}(\rho_1', RT_{i_1'}) \quad \text{and} \quad se \text{ is high in } \mathbf{region}(i_0', \tau')$$

   where τ, τ' are tags s.t. $i_0 \mapsto^{\tau} i_1$ and $i_0' \mapsto^{\tau'} i_1'$. Using case 1 of Lemma I.13 there exists $\beta', \beta \subseteq \beta'$ s.t. (with the help of Lemma F.7

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

   Invoking Lemma I.9 will give us two cases:

   - There exists j, j' with $1 \le j \le k$ and $1 \le j' \le k'$ 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).

■