A Sound Information-Flow Analysis for Cassandra

Technical Report TUD-CS-2014-0064

March 2014

Steffen Lortz,
Heiko Mantel,
Artem Starostin,
Alexandra Weber

TECHNISCHE
UNIVERSITÄT
DARMSTADT

MAIS
Modeling and Analysis
of Information Systems

A Sound Information-Flow Analysis for Cassandra

Steffen Lortz, Heiko Mantel, Artem Starostin, and Alexandra Weber

Modeling and Analysis of Information Systems (MAIS)
Computer Science Department, TU Darmstadt, Germany
`lastname@mais.informatik.tu-darmstadt.de`

Technical Report TUD-CS-2014-0064
March 2014

Abstract Cassandra, a Security-Certifying App Store for Android, is a tool that allows its user to specify information-flow requirements and to analyze apps against these requirements before they are installed on the user's mobile device. The information-flow analysis is performed statically by means of a security-type system for Dalvik bytecode, the language in which Android apps are typically distributed. One of Cassandra's distinguishing features is the soundness result for its analysis, which ensures that only apps adhering to the given security requirements pass the analysis. This report presents the formal foundations of the type-based information-flow analysis of Cassandra and the corresponding soundness result. To the best of our knowledge, this is the first sound information-flow analysis for Dalvik bytecode.

Contents

1 Introduction

Modern Android smartphones operate on all kinds of their user's private information including, e.g., contacts, calendars, GPS location, and browsing history. Whereas the Android operating system provides some protection mechanisms for private data [Prob], these mechanisms do not provide control and transparency on how apps use the private data. In fact, studies [TK10, EOMC11] show that many apps actually abuse private data by sending it silently over the Internet, e.g., to advertising companies.

These leaks of private data are possible because the built-in protection mechanisms of the Android operating system, such as the permission system, allow to restrict access to private data, but cannot control the propagation of such data after access has been granted. One possible solution to deal with this shortcoming is to augment the available protection mechanisms with *information-flow control* [DD77], which allows to control where private data is propagated.

Following this idea, we developed *Cassandra*, a Security-Certifying App Store for Android. Cassandra is a tool that allows its user to check apps against his personal information-flow requirements before installing them on the mobile device. Cassandra consists of a server, providing apps and the security analysis service, and a client-app, supporting the selection of apps and control of the security analysis. The functionality of Cassandra resembles that of existing app stores, e.g., F-Droid [Lim], augmented by means to specify security requirements and to analyze apps against these requirements.

The analysis of apps with Cassandra is performed statically by means of a novel security-type system in the style of [VIS96]. Cassandra's security-type system operates on Dalvik bytecode, the format in which Android apps are typically distributed. It detects data leaks as well as implicit information leaks that occur through control-flow dependencies on secrets.

The trust in Cassandra's information-flow analysis is substantiated by a proof that all typable programs satisfy a noninterference-like security property [GM82] with respect to formal semantics of Dalvik bytecode. This result ensures that the security-type system underlying the analysis of Cassandra detects all information leaks in an app with respect to a given information-flow requirement.

The purpose of this report is to give details on Cassandra's security-type system and its formal foundations. More specifically, we present

- the syntax and operational semantics for an abstract version of the Dalvik bytecode language (Section 2),
- the formalization of a timing- and termination-insensitive noninterference-like security property for programs in this abstract language (Section 3),
- the definition of the security-type system that enforces the security property (Section 4), and
- the soundness proof for this security type system, ensuring that all typable apps satisfy the given information-flow security requirements (Section 5).

To the best of our knowledge, the presented security-type system is the first information-flow analysis for Dalvik bytecode that has been proven sound.

1.1 Notational Conventions

We use the notation $f : X \rightharpoonup Y$ for partial functions, i.e., functions where some $x \in X$ are mapped to values in Y and for some $x \in X$ the value of f is undefined. We denote that the value for x is undefined by $f(x) = \perp$. The domain of the partial function f is denoted $dom(f)$. For the range of a function f, we write $rng(f)$.

We denote the composition of two functions $g : Y \to Z$ and $f : X \to Y$ by $g \circ f$ where $(g \circ f)(x) = g(f(x))$ for all $x \in X$.

For all functions $f : X \to Y$, elements $x_1, \ldots, x_n \in X$ and $y_1, \ldots, y_n \in Y$, we denote by $f[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ the updated function that maps each $x_i \in \{x_1, \ldots, x_n\}$ to the corresponding y_i and each $x \in X \setminus \{x_1, \ldots, x_n\}$ to $f(x)$.

We denote the inverse of an injective function $f : X \to Y$ by f^{-1}, i.e., for all $x \in X$ and all $y \in Y$, $f^{-1}(y) = x$ if and only if $f(x) = y$.

We denote the type for a list of elements of type T by T^*. Given a list $L \in T^*$, we write $\mathsf{length}(L)$ for the length of L, and $L[i]$ for the i-th element of the list L starting from $i = 0$. The empty list is denoted by $[]$ and the list $L \in T^*$ with the entries $t_1, \ldots, t_n \in T$ is written $[t_1, \ldots, t_n]$.

We use $\mathbb{N}_0$ to denote the set of natural numbers including 0. For any set X, $\mathcal{P}(X)$ denotes the powerset of X, i.e., the set of all possible subsets of X. For any two sets X, Y, we denote by $X \subseteq_{\mathsf{fin}} Y$ that X is a finite subset of Y.

2 The ADL Programming Language

This section introduces ADL (*Abstract Dalvik Language*), an abstract version of the Dalvik bytecode language [Proc]. ADL was designed with the idea in mind to focus on the information-flow aspects of the original Dalvik bytecode language and to abstract from less relevant operational details. The instruction set of ADL is based on [Man11].

2.1 Syntax of ADL

The syntactical domains of ADL include the following underspecified sets:

$\mathcal{CID}$: set of class names
$\mathcal{FID}$: set of field names
$\mathcal{MID}$: set of method names
$\mathcal{S}$: set of string symbols
$\mathcal{N}$: set of numerical symbols (e.g., integers and floating point numbers)
$\mathcal{L}_c$: set of constant memory locations
$\mathcal{F}$: set of fields

The sets $\mathcal{CID}$, $\mathcal{FID}$, and $\mathcal{MID}$ are assumed to be mutually disjoint. Memory locations from the set $\mathcal{L}_c$ are used to refer to global constants. Elements from the set $\mathcal{F}$ allow to uniquely identify fields, since different field names could refer to the same fields due to inheritance.

Register names of the form v_i for some number i are used to refer to registers in the syntax of ADL.

Definition 1 (Register names). *The set of register names $\mathcal{X}$ is defined by $\mathcal{X} = \{v_i \mid i \in \mathbb{N}_0\}$.*

The sets of ADL operators are divided into sets $\mathcal{UNOP}$ of unary operators, $\mathcal{BINOP}$ of binary operators, and $\mathcal{RELOP}$ of relational operators.

Definition 2 (Operator symbols). *Let $\mathcal{CONV}$ be a set of symbols of type casting operators, then the sets of operator symbols are defined by*

$$\begin{aligned}\mathcal{UNOP} &= \{-, \neg\} \cup \mathcal{CONV},\\ \mathcal{BINOP} &= \{+, -, *, /, \%, \wedge, \vee, \oplus, <<, >>, >>>\}, \text{ and}\\ \mathcal{RELOP} &= \{=, \neq, <, >, \leq, \geq\}.\end{aligned}$$

The underspecified set $\mathcal{CONV}$ contains symbols for type casting operations that convert between 32 bit and 64 bit values (e.g., integer to long), and from one representation to another representation with the same length (e.g., integer to float). $\mathcal{UNOP}$ is the set of symbols for unary operations, e.g., the negation of a value. The set $\mathcal{BINOP}$ contains symbols for binary operations, such as addition and subtraction. The set $\mathcal{RELOP}$ is the set of symbols of operations that compare values, e.g., for equality.

Definition 3 (ADL instructions). *The set $\mathcal{INSTR}$ of ADL instructions contains the following instructions:*

Arithmetic Instructions
`move` v_a, v_b
`move-wide` v_a, v_b
`const` v_a, n
`const-wide` v_a, n
`cmp` v_a, v_b, v_c
`cmp-wide` v_a, v_b, v_c
`unop` v_a, v_b, uop
`unop-wide` v_a, v_b, uop
`unop-wideS` v_a, v_b, uop
`unop-wideT` v_a, v_b, uop
`binop` v_a, v_b, v_c, bop
`binop-wide` v_a, v_b, v_c, bop
`binop-2addr` v_a, v_b, bop
`binop-2addr-wide` v_a, v_b, bop
`binop-lit` v_a, v_b, n, bop

Control Flow Instructions
`nop`
`goto` n
`if-test` v_a, v_b, n, rop
`if-testz` v_a, n, rop

Object-Related Instructions
`instance-of` v_a, v_b, cl
`new-instance` v_a, cl
`const-string` v_a, s
`const-class` v_a, cl
`iget` v_a, v_b, fid
`iget-wide` v_a, v_b, fid
`iput` v_a, v_b, fid
`iput-wide` v_a, v_b, fid
`sget` v_a, fid
`sget-wide` v_a, fid
`sput` v_a, fid
`sput-wide` v_a, fid

Array-Related Instructions
`array-length` v_a, v_b
`new-array` v_a, v_b

`filled-new-array` $v_a, \ldots, v_e, n$
`filled-new-array-range` v_a, n
`fill-array-data` $v_a, u_0, \ldots u_n$
`aget` v_a, v_b, v_c
`aget-wide` v_a, v_b, v_c
`aput` v_a, v_b, v_c
`aput-wide` v_a, v_b, v_c

Method-Related Instructions
`invoke-virtual` $v_a, \ldots, v_e, n, mid$
`invoke-super` $v_a, \ldots, v_e, n, mid$
`invoke-direct` $v_a, \ldots, v_e, n, mid$
`invoke-interface` $v_a, \ldots, v_e, n, mid$
`invoke-static` $v_a, \ldots, v_e, n, mid$
`invoke-virtual-range` v_a, n, mid
`invoke-super-range` v_a, n, mid
`invoke-direct-range` v_a, n, mid
`invoke-interface-range` v_a, n, mid
`invoke-static-range` v_a, n, mid
`move-result` v_a
`move-result-wide` v_a
`return-void`
`return` v_a
`return-wide` v_a

where $n \in \mathcal{N}$ denotes a constant numerical value, $s \in \mathcal{S}$ denotes a constant string, $mid \in \mathcal{MID}$ denotes a method name, $fid \in \mathcal{FID}$ denotes a field name, $cl \in \mathcal{CID}$ denotes a class name, $v_a, v_b, v_c, v_d, v_e \in \mathcal{X}$ denote registers, $u_i \in (\mathcal{N} \cup \mathcal{L}_c)$ for all $i \in \mathbb{N}_0$ denote constant values, $rop \in \mathcal{RELOP}$ denotes a relational operator, $uop \in \mathcal{UNOP}$ denotes a unary operator, and $bop \in \mathcal{BINOP}$ denotes a binary operator.

The intuitive meaning of the ADL instructions will be introduced in Section 2.2. The set of instructions abstracts from actual Dalvik bytecode instructions in aspects that have no impact on the the information flow analysis (e.g., the the actual number of method parameters, or the type of arrays in array specific instructions). Details on which concrete Dalvik bytecode instructions are covered by the instructions of Definition 3 can be found in Appendix B.

Each instruction represents one atomic computation step. To represent complex computations, methods are defined as non-empty lists of instructions.

Definition 4 (ADL methods). *The set $\mathcal{M}$ of all ADL methods is defined by $\mathcal{M} = \mathcal{INSTR}^* \setminus []$.*

We assume a total function $\mathsf{params} : \mathcal{MID} \to \mathbb{N}_0$ that specifies the number of parameters of a method with a given name.

In Dalvik bytecode programs, methods and fields are declared by classes. All classes are organized in a class hierarchy, which defines the accessibility of methods and fields. This means that classes can extend other classes (i.e., super classes) to inherit and overwrite methods and inherit fields from their super class. In ADL programs, the declaration of fields and methods as well as the class hierarchy that supports the inheritance of fields and methods is modeled implicitly by five partial functions:

- $\mathsf{lookup\text{-}field} : \mathcal{FID} \to \mathcal{F}$ returns the field corresponding to the given field name.
- $\mathsf{lookup\text{-}direct} : \mathcal{MID} \times \mathcal{CID} \rightharpoonup \mathcal{M}$ returns the method with the given name declared by the class with the given name. If the class does not declare a method with the given name, $\mathsf{lookup\text{-}direct}$ is undefined.

– $\mathsf{lookup\text{-}super} : \mathcal{MID} \times \mathcal{CID} \rightharpoonup \mathcal{M}$ returns the method with the given name declared or inherited by the super class of the class with the given name. If there is no such method declared or inherited by the super class, lookup-super is undefined.
– $\mathsf{lookup\text{-}virtual} : \mathcal{MID} \times \mathcal{CID} \rightharpoonup \mathcal{M}$ returns the method with the given name declared or inherited by the class with the given name. If there is no such method declared or inherited by the class with the given name, lookup-virtual is undefined.
– $\mathsf{lookup\text{-}static} : \mathcal{MID} \rightharpoonup \mathcal{M}$ returns the static method with the given name. If there is no such method lookup-static is undefined.

Definition 5 (ADL programs). *An ADL program P is a tuple*

$$P = (\mathcal{CID}_P, \mathcal{FID}_P, \mathcal{MID}_P, \mathcal{M}_P, \mathcal{F}_P, \textsf{\textit{lookup-field}}_P, \textsf{\textit{lookup-static}}_P, \textsf{\textit{lookup-direct}}_P, \textsf{\textit{lookup-super}}_P, \textsf{\textit{lookup-virtual}}_P),\ \textit{where}$$

$$\mathcal{CID}_P \subseteq_{\mathsf{fin}} \mathcal{CID},\ \mathcal{FID}_P \subseteq_{\mathsf{fin}} \mathcal{FID},\ \mathcal{MID}_P \subseteq_{\mathsf{fin}} \mathcal{MID},\ \mathcal{M}_P \subseteq_{\mathsf{fin}} \mathcal{M},\ \mathcal{F}_P \subseteq_{\mathsf{fin}} \mathcal{F},$$

$$\begin{aligned}
\textsf{\textit{lookup-field}}_P &: \mathcal{FID}_P \rightarrow \mathcal{F}_P,\\
\textsf{\textit{lookup-static}}_P &: \mathcal{MID}_P \rightharpoonup \mathcal{M}_P,\\
\textsf{\textit{lookup-direct}}_P &: \mathcal{MID}_P \times \mathcal{CID}_P \rightharpoonup \mathcal{M}_P,\\
\textsf{\textit{lookup-super}}_P &: \mathcal{MID}_P \times \mathcal{CID}_P \rightharpoonup \mathcal{M}_P,\ \textit{and}\\
\textsf{\textit{lookup-virtual}}_P &: \mathcal{MID}_P \times \mathcal{CID}_P \rightharpoonup \mathcal{M}_P.
\end{aligned}$$

An ADL program consists of finite sets of class names, field names, method names, methods, fields, and functions for looking up fields and methods. The sets of names correspond to the classes, fields, and methods that the program declares. The set of methods contains the method definitions required by the program. The five partial functions define the declaration and inheritance of fields and methods with respect to the classes of the program.

Remark 1. In the remainder of this report, we assume an arbitrary but fixed program P with corresponding sets and functions as of Definition 5.

2.2 Semantics of ADL

We define the operational semantics of ADL in the spirit of Barthe, Pichardie, and Rezk [BPR07, BPR08], who developed a formal semantics for the bytecode language of the Java Virtual Machine. Our proposed semantics for Dalvik bytecode is based on the official documentation of the *Android Open Source Project* [Proc].

The operational semantics of ADL programs is defined in terms of a transition relation on execution states. Each state specifies the position of the next instruction to be executed in the program and the current state of the memory, where the memory consists of the state of the registers and a heap. Execution states and their underlying domains are defined as follows.

Definition 6 (Semantical domains). *The semantical domains of ADL programs are defined by*

$$
\begin{aligned}
\mathcal{L} &= \mathcal{L}_c \cup \mathcal{L}_v && \textit{locations}\\
\mathcal{V} &= \mathcal{N} \cup \mathcal{L} \cup \{\mathsf{void}\} && \textit{values}\\
\mathcal{O} &= \mathcal{CID} \times (\mathcal{F} \rightharpoonup \mathcal{V}) && \textit{objects}\\
\mathcal{A} &= \mathbb{N}_0 \times (\mathbb{N}_0 \rightharpoonup \mathcal{V}) && \textit{arrays}\\
\mathcal{X}_{res} &= \{result_{lower}, result_{upper}\} && \textit{reserved registers}\\
\mathcal{R} &= (\mathcal{X} \cup \mathcal{X}_{res}) \to \mathcal{V} && \textit{register states}\\
\mathcal{H} &= \mathcal{L} \rightharpoonup (\mathcal{O} \cup \mathcal{A}) && \textit{heaps}\\
\mathcal{C} &= \mathcal{H} \times \mathbb{N}_0 \times \mathcal{R} && \textit{intermediate states}\\
\mathcal{C}_{\mathsf{final}} &= \mathcal{V} \times \mathcal{H} && \textit{final states}
\end{aligned}
$$

where $\mathcal{L}_v$ with $\mathcal{L}_v \cap \mathcal{L}_c = \emptyset$ is the set of variable locations, void *is a special value such that* void $\notin (\mathcal{N} \cup \mathcal{L})$, *and $result_{lower}, result_{upper}$ are special registers such that $result_{lower}$, $result_{upper} \notin \mathcal{X}$.*

The set $\mathcal{L}$ of locations consists of the locations $\mathcal{L}_c$ of constants and class objects that can be statically referred to from the bytecode and of the variable locations $\mathcal{L}_v$ at which dynamically created objects and arrays are stored. The set $\mathcal{V}$ contains *values* that can be stored in registers, fields, and arrays. These are, most importantly, numerical values and locations. The special value void only occurs as the result of methods with the return type `void`, i.e., methods that do not return a value, and in uninitialized registers. An *object* from the set $\mathcal{O}$ consists of a class name that specifies its type and a mapping of its fields to values. An *array* from the set $\mathcal{A}$ has a length and entries which are represented by a partial function that maps each index of the array to a value. A natural number is an index of a given array if it is smaller than the length of the array. This definition of arrays requires that $\mathcal{N}$ contains at least the natural numbers $\mathbb{N}_0$ as self-evaluating numerical symbols.

The *states of the registers* are represented as a mapping of register names to values. Any function $r \in \mathcal{R}$ returns, for any register name v_a, $a \in \mathbb{N}_0$, the value $r(v_a)$ that is stored in the given register. The special registers $result_{lower}$ and $result_{upper}$ are reserved to store the return values of method calls. The result of methods with 32 bit return values is obtained through register $result_{lower}$. The return values of 64 bits are divided into both registers: the lower 32 bits of the result value are stored in $result_{lower}$ and the upper 32 bits in $result_{upper}$.

A *heap* stores objects, arrays, as well as special class objects in which values of static fields are stored. For a given location $l \in \mathcal{L}$, a heap $h \in \mathcal{H}$ returns the instance $h(l)$ stored at that location. If nothing is stored at the given location, $h(l)$ is undefined.

Intermediate states $\langle h, pp, r\rangle \in \mathcal{C}$ consist of the program point $pp \in \mathbb{N}_0$ of the next instruction to be executed and the current memory state, represented as a heap $h \in \mathcal{H}$ and a register state $r \in \mathcal{R}$. The program point corresponds to the index of the instruction in the executed method. When a method terminates, it yields a *final state* $\langle u, h\rangle \in \mathcal{C}_{\mathsf{final}}$. A final state is a tuple of the returned value of the method $u \in \mathcal{V}$, and the heap $h \in \mathcal{H}$ at the point in time when the method terminated.

To increase the readability of frequently used operations on objects and arrays, we introduce corresponding selector functions and abbreviations.

Definition 7 (Selectors for objects). *For all objects $o \in \mathcal{O}$, mappings of fields to values $F : \mathcal{F} \rightharpoonup \mathcal{V}$, and class names $c \in \mathcal{CID}$ such that $o = (c, F)$, the function $\cdot.\mathsf{class} : \mathcal{O} \to \mathcal{CID}$ is defined by $o.\mathsf{class} = c$ and the function $\cdot.\mathsf{fields} : \mathcal{O} \to (\mathcal{F} \rightharpoonup \mathcal{V})$ is defined by $o.\mathsf{fields} = F$.*

The functions $\cdot.\mathsf{class}$ and $\cdot.\mathsf{fields}$ are selectors for the class of objects and the state of their fields respectively. To access the value of the field f of the object o, we write $o.f$ as a shorthand for $o.\mathsf{fields}(f)$. Moreover, we write $o[f_1 \mapsto u_1, \ldots, f_n \mapsto u_n]$ to denote the object $(o.\mathsf{class}, o.\mathsf{fields}[f_1 \mapsto u_1, \ldots, f_n \mapsto u_n])$, i.e., the object that is equal to o except that the fields $f_1, \ldots, f_n$ map to the values $u_1, \ldots, u_n$.

Definition 8 (Selectors for arrays). *For all arrays $a \in \mathcal{A}$, lengths $n \in \mathbb{N}_0$, and maps of indices to values $m : \mathbb{N}_0 \rightharpoonup \mathcal{V}$ such that $a = (n, m)$, the function $\cdot.\mathsf{length} : \mathcal{A} \to \mathbb{N}_0$ is defined by $a.\mathsf{length} = n$ and the function $\cdot.\mathsf{entries} : \mathcal{A} \to (\mathbb{N}_0 \rightharpoonup \mathcal{V})$ is defined by $a.\mathsf{entries} = m$.*

The functions $\cdot.\mathsf{length}$ and $\cdot.\mathsf{entries}$ are selectors for the length of arrays and their entries, respectively. To access the entry at index i of some array a, we write $a[i]$ as shorthand for $a.\mathsf{entries}(i)$. Moreover, we write $a[i_1 \mapsto u_1, \ldots, i_n \mapsto u_n]$ for referring to the array $(a.\mathsf{length}, a.\mathsf{entries}[i_1 \mapsto u_1, \ldots, i_n \mapsto u_n])$, i.e., the array in which the values at the positions $i_1, \ldots, i_n$ are set to $u_1, \ldots, u_n$.

Semantics of methods. The effect of the execution of a method $m \in \mathcal{M}$ of some program P is defined based on the relation $\Downarrow^{(\cdot)}_{P,m} \subseteq \mathcal{C} \times (\mathbb{N}_0 \times \mathcal{C}_{\mathsf{final}})$ on execution states. The relation is parametric in the method m and in the program P that defines the method.

The formal definition of $\Downarrow^{(\cdot)}_{P,m}$ and the definition of the execution relation for instructions $\rightsquigarrow^{(\cdot)}_{P,m} \subseteq \mathcal{C} \times (\mathbb{N}_0 \times (\mathcal{C} \cup \mathcal{C}_{\mathsf{final}}))$ are mutually recursive. For the definition of the relation $\Downarrow^{(\cdot)}_{P,m}$ we briefly provide the intuition of the relation $\rightsquigarrow^{(\cdot)}_{P,m}$ which is formalized later. Intuitively, the judgment $\langle h, pp, r\rangle \rightsquigarrow^{(n)}_{P,m} \langle h', pp', r'\rangle$ denotes that the instruction at program point pp of the method m defined by program P executed in the initial memory given by h and r yields the possibly changed heap h', register state r', and determines the instruction at program point pp' as the next instruction to be executed. The judgment $\langle h, pp, r\rangle \rightsquigarrow^{(n)}_{P,m} \langle u, h'\rangle$ denotes that the instruction at program point pp of the method m defined by program P executed in the initial memory given by h and r terminates the execution of the method and returns the value u and the possibly changed heap h'. In addition to the resulting states, $\Downarrow^{(\cdot)}_{P,m}$ and $\rightsquigarrow^{(\cdot)}_{P,m}$ both relate a natural number to the input state. It represents the number of method calls executed during the computation of the resulting state. This number is only relevant for the proof of soundness of the type system given in Chapter 5. It does not affect the computation of the resulting state for any instruction or method.

Definition 9 (Transition relation for methods). *Let $m \in \mathcal{M}_P$ be any method defined by program P. The execution relation $\Downarrow^{(\cdot)}_{P,m} \subseteq \mathcal{C} \times (\mathbb{N}_0 \times \mathcal{C}_{\mathsf{final}})$ for method m of program P is defined by the following rules:*

$$\frac{\langle h, pp, r\rangle \rightsquigarrow^{(n)}_{P,m} \langle u, h'\rangle}{\langle h, pp, r\rangle \Downarrow^{(n)}_{P,m} \langle u, h'\rangle}$$

$$\frac{\langle h, pp, r\rangle \rightsquigarrow^{(n)}_{P,m} \langle h', pp', r'\rangle \quad \langle h', pp', r'\rangle \Downarrow^{(n')}_{P,m} \langle u, h''\rangle}{\langle h, pp, r\rangle \Downarrow^{(n+n')}_{P,m} \langle u, h''\rangle}$$

where $h, h', h'' \in \mathcal{H}$, $pp, pp' \in \mathbb{N}_0$, $r, r' \in \mathcal{R}$, $u \in \mathcal{V}$, and $n, n' \in \mathbb{N}_0$.

The judgment $\langle h, pp, r\rangle \Downarrow^{(n)}_{P,m} \langle u, h'\rangle$ represents the terminating execution of a method. If some initial state $\langle h, pp, r\rangle \in \mathcal{C}$ and some final state $\langle u, h'\rangle \in \mathcal{C}_{\mathsf{final}}$ are related by $\Downarrow^{(n)}_{P,m}$, the method m of program P executed in the initial register state r and heap h recursively calls n methods and terminates in the possibly changed heap h' while returning the result u.

Definition 10 (Semantics of methods). *Let $m \in \mathcal{M}_P$ be a method defined by program P. The method m executed in any initial heap $h \in \mathcal{H}$ and register state $r \in \mathcal{R}$ terminates yielding the return value $u \in \mathcal{V}$ and the final heap $h' \in \mathcal{H}$ after $n \in \mathbb{N}_0$ method calls occurred, if and only if $\langle h, 0, r\rangle \Downarrow^{(n)}_{P,m} \langle u, h'\rangle$.*

The constant number 0 in the initial configuration indicates that the execution of methods always starts from the first instruction.

Semantics of programs. Android applications usually have more than one entry point method with which they start their execution [Proa], e.g., `Activity.-onCreate()` or `Activity.onResume()`. We denote the set of method names of entry points of the program P by the subset EP_P of $\mathcal{MID}_P$. Hence, the possible semantics of an ADL program P under any suitable initial state $\langle h, 0, r\rangle \in \mathcal{C}$ corresponds to the semantics of any such method $m \in \mathcal{M}_P$.

A "suitable" initial heap $h \in \mathcal{H}$ contains at least the objects of string constants and classes that are used in the program, as well as a special class object for each class that holds the values of its static fields. The function $\mathsf{nameToReference} : (\mathcal{CID} \cup \mathcal{S} \cup \mathcal{FID}) \rightharpoonup \mathcal{L}_c$ maps the syntactical representation of these objects (class names, strings, and names of static fields) to the locations of the respective objects on the heap. Since these locations can be referred to from the bytecode, they are constant over all program executions. We moreover assume that the initial register state $r \in \mathcal{R}$ maps the registers for the parameters of the method to suitable initial values, e.g., to valid locations of objects of the proper type on the heap, and that all other registers map to void. These assumptions are adequate as, in practice, the Dalvik virtual machine takes care of the proper initialization of class objects, static fields, and type safety.

Semantics of instructions. Values in Dalvik bytecode can be 32 or 64 bit wide whereas all registers are 32 bit wide. Values that are 64 bit wide are therefore stored in two consecutive registers [Proc]. This is also reflected in a set of ADL instructions specifically for the handling of 64 bit values, i.e., instructions with the suffix `-wide`. The definition of the semantics of these ADL instructions often depends on the conversion of 64 bit values to 32 bit values and vice versa. To this end, we assume the following functions:

- $\mathsf{lower} : \mathcal{V} \to \mathcal{V}$ takes a value and returns the value of the first 32 bits.
- $\mathsf{upper} : \mathcal{V} \to \mathcal{V}$ takes a value and returns the value of the last 32 bits.
- $\cdot \bullet \cdot : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ takes two 32 bit values and returns a 64 bit value which is the concatenation of the inputs.

Note that for 32 bit values, lower and upper yield the same result. We moreover assume the following auxiliary functions and relations:

$$\begin{aligned}
\mathsf{assignmentCompatible} &\subseteq \mathcal{CID} \times \mathcal{CID} \\
\mathsf{nextFreeLocation} &: \mathcal{H} \rightharpoonup \mathcal{L}_v \\
\mathsf{defaultObject} &: \mathcal{CID} \to \mathcal{O} \\
\mathsf{defaultArray} &: \mathbb{N}_0 \to \mathcal{A} \\
\mathsf{defaultRegisters} &: \mathcal{V}^* \to \mathcal{R}
\end{aligned}$$

Two class names $c_1, c_2 \in \mathcal{CID}$ are related by $\mathsf{assignmentCompatible}$, written $\mathsf{assignmentCompatible}(c_1, c_2)$, if and only if c_1 is a subclass of or equal to c_2. Intuitively, this means that it is safe to assign an object of class c_1 to a register or field of the type c_2. The partial function $\mathsf{nextFreeLocation}$ returns a free variable location on the heap, unless the heap is full. It is used in the semantics of instructions that create objects or arrays to determine where to put them on the heap. The creation of new objects and arrays is modeled by the functions $\mathsf{defaultObject}$ and $\mathsf{defaultArray}$. The function $\mathsf{defaultObject}$ returns an object of the class with the given name, and the function $\mathsf{defaultArray}$ returns an array with the given length. Both, the object's and the array's fields are initialized with a zero-value corresponding to the type of the field, e.g., 0 for fields of the type integer, 0.0 for fields containing floating point numbers, the special location `null` for fields storing references to objects and arrays, and so on. The function $\mathsf{defaultRegisters}$ allocates a list of registers to store parameter values for method invocation.

Definition 11 (Register allocation). *Let $r \in \mathcal{R}$, and $x \in \mathcal{V}^*$. The function* $\mathsf{defaultRegisters} : \mathcal{V}^* \to \mathcal{R}$ *for register allocation is defined such that*

$\mathsf{defaultRegisters}(x) = r$ *if and only if for all* $i \in \mathbb{N}_0$ *it holds that*

$$r(v_i) = \begin{cases} x[i] & \text{if } i < \mathsf{length}(x) \\ \mathsf{void} & \text{otherwise.} \end{cases}$$

The effect that the execution of instructions has on a state is defined by the execution relation $\rightsquigarrow^{(\cdot)}_{P,m} \subseteq \mathcal{C} \times (\mathbb{N}_0 \times \mathcal{C} \cup \mathcal{C}_{\mathsf{final}})$. The relation is parametric in a program P and the current method m defined by P. The execution relation is defined by rules of the form

$$\text{rName}\frac{premise_1 \quad \dots \quad premise_n}{\langle h, pp, r\rangle \rightsquigarrow^{(n)}_{P,m} \langle h', pp', r'\rangle} \qquad \text{rName}\frac{premise_1 \quad \dots \quad premise_n}{\langle h, pp, r\rangle \rightsquigarrow^{(n)}_{P,m} \langle u, h'\rangle}$$

for each instruction. To keep the rules clutter-free, we assume the method definitions to be well-formed with respect to the Dalvik bytecode verifier [Prod]. In particular, we assume type-correct programs, and that all numerical values that are given as parameters, such as n in `goto` n, lie within the range that is sensible for the parameter.

In the following, we present the semantics of selected instructions that are representative for groups of similar instructions. The semantics of the remaining instructions from Definition 3 are given in Appendix C.

Arithmetic instructions. Arithmetic instructions compute values from given parameters, e.g., constants or values stored in registers. All have in common that they do neither access the heap nor invoke any methods. The instruction to be executed after an arithmetic instruction is always the next instruction in the method, i.e., the instruction at the current program point plus one. In the following, the function $\underline{uop}$ denotes the semantics of the operation uop on the domain $\mathcal{N}$. The functions $\underline{bop}$, and $\underline{rop}$ denote the semantics of the operation bop on the domain $\mathcal{N}$, respectively rop on the domain $\mathcal{V}$, and are used in infix notation.

The instruction `move` copies the value from register v_b to register v_a as it is reflected in the judgment derivable with the rule (rMove). The instruction `const` stores a constant numerical value in a register. The instruction `unop` stores the result of applying the unary operation uop to the parameter v_b in the given destination register v_a. The instruction `binop` applies the binary operation $\underline{bop}$ to the values from v_b and v_c and stores the result to v_a.

Control flow instructions. Control flow instructions define the control flow in the executed method. They allow to continue execution at any instruction in the method. They have no effect on the heap or registers and they do not invoke any methods.

The instruction `nop` executes without changing the program state besides selecting the next instruction in the method for execution. The instruction `goto` sets the next instruction to be executed to the instruction with the offset n from the current program point. The semantics of `if-test` is defined by two rules. The rule (rIfTestTrue) covers the case where the comparison of the values in

$$\text{rMove}\;\frac{m[pp] = \texttt{move}\ v_a, v_b}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto r(v_b)]\rangle}$$

$$\text{rConst}\;\frac{m[pp] = \texttt{const}\ v_a, n}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto n]\rangle}$$

$$\text{rUnop}\;\frac{m[pp] = \texttt{unop}\ v_a, v_b, uop \qquad u = \underline{uop}(r(v_b))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto u]\rangle}$$

$$\text{rBinop}\;\frac{m[pp] = \texttt{binop}\ v_a, v_b, v_c, bop \qquad x = r(v_b)\ \underline{bop}\ r(v_c)}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto x]\rangle}$$

Figure 1. Semantics of arithmetic instructions

$$\text{rNop}\;\frac{m[pp] = \texttt{nop}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r\rangle}$$

$$\text{rGoto}\;\frac{m[pp] = \texttt{goto}\ n}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+n, r\rangle}$$

$$\text{rIfTestTrue}\;\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad r(v_a)\ \underline{rop}\ r(v_b)}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+n, r\rangle}$$

$$\text{rIfTestFalse}\;\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad \neg(r(v_a)\ \underline{rop}\ r(v_b))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r\rangle}$$

Figure 2. Semantics of control flow instructions

v_a and v_b by the operator $\underline{rop}$ yields true. In this case, the instruction to be executed is set to the instruction with the offset n from the current program point. The rule (rIfTestFalse) covers the case where the comparison of v_a and v_b yields false. It selects the next instruction of the method for execution.

Object-related instructions. Object-related instructions read from or write to objects on the heap. All instructions select their immediate successor in the method to be executed next. None of the instructions invokes any methods.

The semantics of the instruction `instance-of` consists of two rules, (rInstanceOfTrue) and (rInstanceOfFalse). If the class of the object referenced by v_b is a subtype of cl, the value 1 is stored in v_a. Otherwise, 0 is stored in v_a. Hence, the result is 1 if and only if the object referenced by register v_b is an instance of the class cl. The instruction `new-instance` creates a new object of the given class and stores a reference to it in v_a. The location to create the object at is determined by the function nextFreeLocation whereas the new object is determined using defaultObject. The instruction `const-string` stores the reference to the constant string object with the given text s in v_a. The initial heap we assume already contains all constant string objects of a program and the reference to the specific string's location is determined through the function nameToReference. The instruction `const-class` works like `const-string` but taking a class name as its second parameter. The instruction `iget` copies the value from the field with the name *fid* of the object referenced by v_b to v_a, provided that the object exists at the specified location and has the respective field. The function $\mathsf{lookup\text{-}field}_P$ is used to obtain the field for the given field name. On the same lines as `iget`, the instruction `iput` stores the value of v_a in the field *fid* of the object referenced by v_b. The instruction `sget` stores the value from the static field denoted by *fid* in v_a, given that the respective special object declares the field. The location of the special objects that hold the values of static fields are obtained using nameToReference. The instruction `sput` stores the value from v_a in the static field *fid*.

Array-related instructions. Array-related instructions create, access, and manipulate arrays on the heap. As for the object-specific instructions, all array-specific instructions select their immediate successor in the method to be executed next and none of the instructions invokes any methods.

The instruction `array-length` stores the length of the array referenced by a location in v_b to v_a. The instruction `new-array` creates a new array of the length stored in v_b, stores the array at a new location on the heap, and stores the location of the new array in v_a. The new location is determined using the function nextFreeLocation and the new array is determined using defaultArray. The length of the array is required to be greater than or equal to zero. The instruction `filled-new-array-range` creates a new array in the same way as `new-array`. In addition, `filled-new-array-range` stores the values of n consecutive registers starting from the parameter register v_k in the array. The location of the new array on the heap is stored in the result register $result_{lower}$. The instruction `aget` stores the value at the index given in v_c of the array referenced by the location

$$\text{rInstanceOfTrue}\ \frac{m[pp] = \texttt{instance-of}\ \ v_a, v_b, cl \qquad r(v_b) \in dom(h) \\ \mathsf{assignmentCompatible}(h(r(v_b)).\mathsf{class}, cl)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto 1]\rangle}$$

$$\text{rInstanceOfFalse}\ \frac{m[pp] = \texttt{instance-of}\ \ v_a, v_b, cl \qquad r(v_b) \in dom(h) \\ \neg(\mathsf{assignmentCompatible}(h(r(v_b)).\mathsf{class}, cl))}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto 0]\rangle}$$

$$\text{rNewInstance}\ \frac{m[pp] = \texttt{new-instance}\ \ v_a, cl \qquad h \in dom(\mathsf{nextFreeLocation}) \\ l = \mathsf{nextFreeLocation}(h)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[l \mapsto \mathsf{defaultObject}(cl)], pp+1, r[v_a \mapsto l]\rangle}$$

$$\text{rConstString}\ \frac{m[pp] = \texttt{const-string}\ \ v_a, s \qquad s \in dom(\mathsf{nameToReference})}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{nameToReference}(s)]\rangle}$$

$$\text{rConstClass}\ \frac{m[pp] = \texttt{const-class}\ \ v_a, cl \qquad cl \in dom(\mathsf{nameToReference})}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{nameToReference}(cl)]\rangle}$$

$$\text{rIget}\ \frac{m[pp] = \texttt{iget}\ \ v_a, v_b, fid \qquad fid \in dom(\mathsf{lookup\text{-}field}_P) \\ r(v_b) \in dom(h) \qquad o = h(r(v_b)) \\ f = \mathsf{lookup\text{-}field}_P(fid) \qquad f \in dom(o.\mathsf{fields})}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto o.f]\rangle}$$

$$\text{rIput}\ \frac{m[pp] = \texttt{iput}\ \ v_a, v_b, fid \qquad fid \in dom(\mathsf{lookup\text{-}field}_P) \\ r(v_b) \in dom(h) \qquad o = h(r(v_b)) \\ f = \mathsf{lookup\text{-}field}_P(fid) \qquad f \in dom(o.\mathsf{fields})}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[r(v_b) \mapsto o[f \mapsto r(v_a)]], pp+1, r\rangle}$$

$$\text{rSget}\ \frac{m[pp] = \texttt{sget}\ \ v_a, fid \qquad fid \in dom(\mathsf{nameToReference}) \\ l = \mathsf{nameToReference}(fid) \qquad fid \in dom(\mathsf{lookup\text{-}field}_P) \\ f = \mathsf{lookup\text{-}field}_P(fid) \qquad f \in dom(h(l).\mathsf{fields}) \qquad u = h(l).f}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto u]\rangle}$$

$$\text{rSput}\ \frac{m[pp] = \texttt{sput}\ \ v_a, fid \qquad fid \in dom(\mathsf{nameToReference}) \\ l = \mathsf{nameToReference}(fid) \qquad fid \in dom(\mathsf{lookup\text{-}field}_P) \\ f = \mathsf{lookup\text{-}field}_P(fid) \qquad o = h(l) \qquad f \in dom(o.\mathsf{fields})}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[l \mapsto o[f \mapsto r(v_a)]], pp+1, r\rangle}$$

Figure 3. Semantics of object-related instructions

$$\text{rArrayLength}\ \frac{m[pp] = \texttt{array-length}\ v_a, v_b \qquad r(v_b) \in dom(h)}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto h(r(v_b)).\mathsf{length}]\rangle}$$

$$\text{rNewArray}\ \frac{\begin{array}{cc} m[pp] = \texttt{new-array}\ v_a, v_b & h \in dom(\mathsf{nextFreeLocation}) \\ l = \mathsf{nextFreeLocation}(h) & 0 \leq r(v_b) \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[l \mapsto \mathsf{defaultArray}(r(v_b))], pp+1, r[v_a \mapsto l]\rangle}$$

$$\text{rFilledNewArrayR}\ \frac{\begin{array}{c} m[pp] = \texttt{filled-new-array-range}\ v_k, n \\ h \in dom(\mathsf{nextFreeLocation}) \\ l = \mathsf{nextFreeLocation}(h) \qquad x = \mathsf{defaultArray}(n) \\ ar = x[0 \mapsto r(v_k), \ldots, n-1 \mapsto r(v_{k+n-1})] \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[l \mapsto ar], pp+1, r[result_{lower} \mapsto l]\rangle}$$

$$\text{rAget}\ \frac{\begin{array}{ccc} m[pp] = \texttt{aget}\ v_a, v_b, v_c & r(v_b) \in dom(h) & ar = h(r(v_b)) \\ u = ar[r(v_c)] & 0 \leq r(v_c) < ar.\mathsf{length} & \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto u]\rangle}$$

$$\text{rAput}\ \frac{\begin{array}{ccc} m[pp] = \texttt{aput}\ v_a, v_b, v_c & r(v_b) \in dom(h) & ar = h(r(v_b)) \\ x = ar[r(v_c) \mapsto r(v_a)] & 0 \leq r(v_c) < ar.\mathsf{length} & \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[r(v_b) \mapsto x], pp+1, r\rangle}$$

Figure 4. Semantics of array-related instructions

in v_b to register v_a, given that the value of v_b points to an array on the heap and the index given in v_c is within the domain of this array. Correspondingly, the instruction `aput` stores the value from v_a at the index given in v_c of the array referenced by the location in v_b.

Method-related instructions. Method-related instructions comprise instructions for method invocation, return instructions, and instructions that copy return values of method calls to registers.

$$\text{rIVR}\frac{\begin{array}{c} m[pp] = \texttt{invoke-virtual-range}\ v_k, n, mid \qquad r(v_k) \in dom(h) \\ (mid, h(r(v_k)).\mathsf{class}) \in dom(\mathsf{lookup\text{-}virtual}_P) \\ m' = \mathsf{lookup\text{-}virtual}_P(mid, h(r(v_k)).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_k), \ldots r(v_{k+n-1})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h'\rangle \end{array}}{\langle h, pp, r\rangle \leadsto^{(n'+1)}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{rIStR}\frac{\begin{array}{c} m[pp] = \texttt{invoke-static-range}\ v_k, n, mid \\ mid \in dom(\mathsf{lookup\text{-}static}_P) \qquad m' = \mathsf{lookup\text{-}static}_P(mid) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_k), \ldots r(v_{k+n-1})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h'\rangle \end{array}}{\langle h, pp, r\rangle \leadsto^{(n'+1)}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{rMoveR}\frac{m[pp] = \texttt{move-result}\ v_a}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto r(result_{lower})]\rangle}$$

$$\text{rReturnVoid}\frac{m[pp] = \texttt{return-void}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle \mathsf{void}, h\rangle}$$

$$\text{rReturn}\frac{m[pp] = \texttt{return}\ v_a}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle r(v_a), h\rangle}$$

Figure 5. Semantics of method-related instructions

The instruction `invoke-virtual-range` determines the method with the name *mid*, declared or inherited by the class of the object referenced by the location in v_k, and executes it. The method definition is determined using $\mathsf{lookup\text{-}virtual}_P$. The method is executed from its first statement, i.e., program point 0, with the current heap and a list of fresh registers that are initialized with the n parameters of the method, from parameter v_k to v_{k+n-1}. The result value of the method invocation is stored to the special registers $result_{lower}$ and $result_{upper}$. Afterwards, the execution continues with the next instruction in the current method and with the possibly changed memory resulting from the method invocation. The number n' of method calls in the called method m' is added to the one call that occurs when m' is called. Hence, the transition is annotated with $n'+1$. The semantics of the instruction `invoke-static-range` is almost the same as that of `invoke-virtual-range` but it uses $\mathsf{lookup\text{-}static}_P$ instead of $\mathsf{lookup\text{-}virtual}_P$ to obtain the method to execute, identified by the given method name only. The instruction `move-result` copies the value from register $result_{lower}$ to v_a. This instruction and `move-result-wide` (see Definition 15 in the appendix) are the only instructions that can read the special result registers and make their values available for other computations. The instruction `return-void` terminates the execution of the current method with a transition to a final state. In case of `return-void`, the return value is always the constant void. The semantics of the instruction `return` only differs from the semantics of `return-void` in the value of the final state, which is read from the register that is given as a parameter.

Instructions for 64 bit values. Many of the instructions discussed in this section are also available with arguments of 64 bits width. They are mostly similar to the 32 bit variants but they split 64 bit arguments to store them in two successive 32 bit registers and combine 32 bit values from two successive registers to 64 bit values before using them. For their formal semantics, see Appendix C. Special instructions without corresponding 32 bit variants are `unop-wideS` and `unop-wideT`.

$$\text{rUnopWideS}\frac{m[pp] = \texttt{unop-wideS}\ v_a, v_b, uop \qquad u = \underline{uop}(r(v_b) \bullet r(v_{b+1}))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto u]\rangle}$$

$$\text{rUnopWideT}\frac{m[pp] = \texttt{unop-wideT}\ v_a, v_b, uop \qquad u = \underline{uop}(r(v_b))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{lower}(u), v_{a+1} \mapsto \mathsf{upper}(u)]\rangle}$$

Figure 6. Semantics of conversion instructions for 64 bit values

The instruction `unop-wideS` applies unary operators that convert a 64 bit value read from register v_b and v_{b+1} to a 32 bit value and stores the result to register v_a. `unop-wideT` applies unary operators that convert a 32 bit value read from register v_b to a 64-bit value, which is then stored to the registers v_a and v_{a+1}. Both instructions do not access the heap and execution continues with their immediate successor in the method.

3 Security Property

In this section, we introduce the capabilities of our attacker and define a non-interference-like security property that specifies which programs can be executed without leaking information to the attacker observing the execution.

We assume an attacker who knows the code of the program that is executed. He can observe the content of a subset of all storage locations (e.g., registers, fields, the content of arrays) of the program. For the content of these storage locations, the attacker can observe the type of the content, e.g., whether it is a number or location, the class of an object, and the length of an array. The attacker *cannot* observe non-termination and timing behavior of a program execution.

To specify which storage locations in the program are observable to the attacker, we classify them with respect to two security domains, *low* and *high*. Storage locations classified as *low* are public and may be accessed by anybody. The domain *high* is used to classify private storage locations which are not directly observable by the attacker. To prevent that the attacker learns anything about private information from observing storage locations classified as *low*, no information must flow from storage locations classified as *low* to storage locations classified as *high*. This requirement is formalized by the following flow policy.

Definition 12 (Flow policy). *The flow policy is defined as the lattice* $(\mathcal{SL}, \sqsubseteq)$*, where* $\mathcal{SL} = \{low, high\}$ *and* $\sqsubseteq\, = \{(low, high), (low, low), (high, high)\}$*.*

The *flow relation* $\sqsubseteq$ specifies which flows of information are permitted. For any two security domains $s_1, s_2 \in \mathcal{SL}$, information may flow from a storage location classified as s_1 to a storage location with the security domain s_2 if and only if $s_1 \sqsubseteq s_2$. We write $s_1 \sqcup s_2$ to denote the least upper bound of the two domains.

How the storage locations of a particular program are classified into the two security domains is specified with domain assignments.

Definition 13 (Domain assignments).

- *The security domains of the parameters and return values of methods are defined by a set*

  $$\mathsf{mda} \subseteq_{\mathsf{fin}} \{(mid, [p_0, \ldots, p_{n-1}], ret) \in \mathcal{MID}_P \times \mathcal{SL}^* \times \mathcal{SL} \mid n = \mathsf{params}(mid)\}.$$

- *The security domains of fields are defined by a function* $\mathsf{fda} : \mathcal{FID}_P \to \mathcal{SL}$*.*
- *The security domain of the content of all arrays in the program is defined as a constant* $\mathsf{ada} \in \mathcal{SL}$*.*

We refer to elements of the set mda as *method signatures*. A method signature $(mid, [p_0, \ldots, p_{n-1}], ret)$ denotes that a call of the method mid with n parameters that are classified as p_0 to p_{n-1} yields a return value classified as ret. If $\mathsf{fda}(fid) = s$, the content stored in a field with the name fid is classified as s. All contents of all arrays in the program are classified as ada.

Definition 14 (Security policy). *A security policy for a program P consists of the flow policy $(\mathcal{SL}, \sqsubseteq)$ and domain assignments for methods* $\mathsf{mda} \subseteq_{\mathsf{fin}} \mathcal{MID}_P \times \mathcal{SL}^* \times \mathcal{SL}$*, fields* $\mathsf{fda} : \mathcal{FID}_P \to \mathcal{SL}$*, and arrays* $\mathsf{ada} \in \mathcal{SL}$*.*

Remark 2. For the remainder of this report, we assume an arbitrary but fixed *security policy* for program P with domain assignments mda, fda, and ada.

The domain assignments for registers are not directly specified by the security policy but are inferred from the method signatures. Registers are assigned security domains based on their names, independent of a concrete program.

Definition 15 (Domain assignments for registers). *The security domains of registers are defined by functions from the set $\mathcal{RDA}$, where $\mathcal{RDA} = (\mathcal{X} \cup \mathcal{X}_{res} \to \mathcal{SL})$. For any two $\mathsf{rda}_1, \mathsf{rda}_2 \in \mathcal{RDA}$, $\mathsf{rda}_1 \sqsubseteq \mathsf{rda}_2$ holds if and only if $\mathsf{rda}_1(x) \sqsubseteq \mathsf{rda}_2(x)$ holds for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$. For all $\mathsf{rda}_1, \mathsf{rda}_2 \in \mathcal{RDA}$, $\mathsf{rda}_1 \sqcup \mathsf{rda}_2$ is defined by $(\mathsf{rda}_1 \sqcup \mathsf{rda}_2)(x) = \mathsf{rda}_1(x) \sqcup \mathsf{rda}_2(x)$ for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$.*

Note that the relation $\sqsubseteq$ and the function $\sqcup$ on domain assignments for registers are pointwise extensions of $\sqsubseteq$ and $\sqcup$ on security domains.

The notion of information flow in a program from a private security storage location to a public storage location is formalized with the security condition *TIN-ADL* (*Termination-Insensitive Noninterference for the Abstract Dalvik Language*). Intuitively, it requires that if any entry point of the program is executed in any two initial states that are indistinguishable to the attacker, then the two final states of the execution are also indistinguishable to the attacker. Hence, if a program satisfies *TIN-ADL*, the observable part of the output of any entry point execution does not depend on the private input of the entry point. Indistinguishability and the security condition *TIN-ADL* are formalized in the following.

3.1 Indistinguishability

Different program executions may yield a different allocation of objects and arrays on the final heaps. However, any two such heaps may still be indistinguishable to the attacker, as long as any observable array or object on one heap has a corresponding indistinguishable array or object on the other heap. Following Banerjee and Naumann [BN05], the locations of any two corresponding observable arrays and objects are related by a partial injective function on locations $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$. This allows to define indistinguishability modulo the placement of objects and arrays on the heap by making the relations parametric in β. In addition, we require β to map constant locations to themselves: As the attacker is assumed to know the program, he also knows the content of the heap at the constant locations in $\mathcal{L}_c$, i.e., special class objects that store the values of static fields, classes, string constants, and so on.

Definition 16 (Partial injective functions on locations). *The set $\mathcal{B}$ of partial injective functions on locations is defined by*

$$\mathcal{B} = \{\beta : \mathcal{L} \rightharpoonup \mathcal{L} \mid \beta \textit{ is injective} \wedge \forall l \in \mathcal{L}_c.\ \beta(l) = l\}.$$

Two values are indistinguishable for the attacker if they are both void, if they are the same numerical value, or if they are both locations and the first location corresponds to the second one with respect to β.

Definition 17 (Indistinguishability of values). *Let $v_1, v_2 \in \mathcal{V}$ be arbitrary values, and $\beta \in \mathcal{B}$ be a partial injective function on locations. The values v_1 and v_2 are indistinguishable, written $v_1 \sim_\beta v_2$, if and only if*

- *$v_1 = v_2 = \mathsf{void}$, or*
- *there exists a number $n \in \mathcal{N}$ such that $v_1 = v_2 = n$, or*
- *$v_1, v_2 \in \mathcal{L}$, $v_1 \in dom(\beta)$, and $\beta(v_1) = v_2$.*

The notion of indistinguishability for concatenated values is a straightforward extension of value indistinguishability.

Definition 18 (Indistinguishability of concatenated values). *Let $x_1, y_1, x_2, y_2 \in \mathcal{V}$ be arbitrary values and let $\beta \in \mathcal{B}$ be a partial injective function on locations. The concatenated values $x_1 \bullet y_1$ and $x_2 \bullet y_2$ are indistinguishable, written $x_1 \bullet y_1 \sim_\beta x_2 \bullet y_2$ if and only if $x_1 \sim_\beta x_2$ and $y_1 \sim_\beta y_2$. Two concatenated values are equal, i.e., $x_1 \bullet y_1 = x_2 \bullet y_2$ if and only if $x_1 = x_2$ and $y_1 = y_2$.*

Two register states are indistinguishable if all registers classified as *low* hold indistinguishable values.

Definition 19 (Indistinguishability of register states). *Let $r, r' \in \mathcal{R}$ be two register states, $\mathsf{rda} \in \mathcal{RDA}$ be a register domain assignment, and $\beta \in \mathcal{B}$ be a partial injective function on locations. The register states r and r' are indistinguishable with respect to rda, written $r \sim_{\beta,\mathsf{rda}} r'$, if and only if for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$ with $\mathsf{rda}(x) = low$ it holds that $r(x) \sim_\beta r'(x)$.*

Two objects are indistinguishable for the attacker if they are instances of the same class, and all fields that could be referenced by a field name of a *low* security domain hold indistinguishable values in both objects.

Definition 20 (Indistinguishability of objects). *Let $o_1, o_2 \in \mathcal{O}$ be two objects, and let $\beta \in \mathcal{B}$ be a partial injective function on locations. The objects o_1 and o_2 are indistinguishable, written $o_1 \sim_\beta o_2$, if and only if*

1. *$o_1.\mathsf{class} = o_2.\mathsf{class}$ and*
2. *for all fields $f \in dom(o_1.\mathsf{fields})$ and field names $fid \in \mathcal{FID}_P$ such that $f = \textsf{lookup-field}_P(fid)$, it holds that if $\mathsf{fda}(fid) = low$, then $o_1.f \sim_\beta o_2.f$.*

Two arrays are indistinguishable for the attacker if they have the same length and, in case $\mathsf{ada} = low$, all entries are indistinguishable.

Definition 21 (Indistinguishability of arrays). *Let $a_1, a_2 \in \mathcal{A}$ be two arrays and let $\beta \in \mathcal{B}$ be a partial injective function on locations. The arrays a_1 and a_2 are indistinguishable, written $a_1 \sim_\beta a_2$ if and only if $a_1.\mathsf{length} = a_2.\mathsf{length}$ and if $\mathsf{ada} = low$, then for all indices $i \in \mathbb{N}_0$ such that $0 \leq i < a_1.\mathsf{length}$ it holds that $a_1[i] \sim_\beta a_2[i]$.*

Two heaps are indistinguishable if for all locations on the first heap that are potentially observable by the attacker, there exists a corresponding location on the second heap such that the object or array at both locations are indistinguishable.

Since the attacker can distinguish between objects and arrays, the partial function β must not map locations of objects to locations of arrays or vice versa. To distinguish between locations of arrays and objects, we introduce the notations $dom_{\mathcal{O}}(h)$ for the locations that the function h maps to an object and $dom_{\mathcal{A}}(h)$ for the locations that the function h maps to an array.

Definition 22 (Indistinguishability of heaps). *Let h_1 and h_2 be two heaps, and let $\beta \in \mathcal{B}$ be a partial injective function on locations. The heaps h_1 and h_2 are indistinguishable, written $h_1 \sim_\beta h_2$, if and only if*

1. $dom(\beta) \subseteq dom(h_1)$,
2. $rng(\beta) \subseteq dom(h_2)$, *and*
3. *for all locations $l \in dom(\beta)$, either*
   (a) $l \in dom_{\mathcal{O}}(h_1)$, $\beta(l) \in dom_{\mathcal{O}}(h_2)$, *and* $h_1(l) \sim_\beta h_2(\beta(l))$, *or*
   (b) $l \in dom_{\mathcal{A}}(h_1)$, $\beta(l) \in dom_{\mathcal{A}}(h_2)$ *and* $h_1(l) \sim_\beta h_2(\beta(l))$.

3.2 Security

The indistinguishability relations capture the capabilities of the attacker to observe differences of any two register states, heaps, and values. Based on these relations, the security property *TIN-ADL* formalizes that the attacker cannot learn more information about the private input of a program that satisfies *TIN-ADL* by executing the program and observing the results of the execution.

Definition 23 (*TIN-ADL* for methods). *Let $m \in \mathcal{M}_P$ be a method of program P, $mid \in \mathcal{MID}_P$ be a method name, and $p_0, \ldots p_n, ret \in \mathcal{SL}$ for some $n \in \mathbb{N}_0$ be security domains such that $(mid, [p_0, \ldots, p_n], ret) \in \mathsf{mda}$.*

The method m satisfies TIN-ADL with respect to $(mid, [p_0, \ldots, p_n], ret)$ if and only if there exists a register domain assignment $\mathsf{rda} \in \mathcal{RDA}$ with $p_i \sqsubseteq \mathsf{rda}(v_i)$ for all $i \in \mathbb{N}_0$, $i \leq n$ and for all partial injective functions $\beta \in \mathcal{B}$, register states $r_1, r_2 \in \mathcal{R}$, heaps $h_1, h_2, h'_1, h'_2 \in \mathcal{H}$, return values $u_1, u_2 \in \mathcal{V}$, and natural numbers $n_1, n_2 \in \mathbb{N}_0$ such that

$$\begin{aligned}
&r_1 \sim_{\beta,\mathsf{rda}} r_2,\\
&h_1 \sim_\beta h_2,\\
&\langle h_1, 0, r_1 \rangle \Downarrow^{(n_1)}_{P,m} \langle u_1, h'_1 \rangle, \text{ and}\\
&\langle h_2, 0, r_2 \rangle \Downarrow^{(n_2)}_{P,m} \langle u_2, h'_2 \rangle,
\end{aligned}$$

there exists a partial injective function on locations $\beta' \in \mathcal{B}$, such that $\beta \subseteq \beta'$, $h'_1 \sim_{\beta'} h'_2$ and, if $ret = low$, $u_1 \sim_{\beta'} u_2$.

A method m satisfies *TIN-ADL* with respect to a method signature if and only if for any two terminating executions of m from initial configurations with indistinguishable registers and indistinguishable heaps, the final configurations have indistinguishable heaps and, if the method has a public return value, the return values are indistinguishable. Intuitively, the property ensures that the attacker who cannot tell apart the initial states also cannot distinguish the final states after the execution of the method. Hence, the public outputs of a method that satisfies *TIN-ADL* do not depend on its private inputs. Note that the domain assignment for the registers, rda, classifies the parameter registers of the method at least as private as declared in the method signature. The classification of all non-parameter registers does not matter, as these registers are initialized with void and, thus, intuitively contain public information.

This notion of security is extended to ADL programs by ensuring that each method that could be called to execute the program, i.e., each entry point, satisfies *TIN-ADL* for all signatures of the method. Moreover, to assess the security of a program the security classification of the entry points must be complete, i.e., each entry point must have at least one method signature.

Definition 24 (*TIN-ADL* for programs). *Program P satisfies TIN-ADL if and only if*

1. *for all method names of entry points $mid \in \mathsf{EP}_P$ there exists $p_0, \ldots p_n, ret \in \mathcal{SL}$ for some $n \in \mathbb{N}_0$ such that $(mid, [p_0, \ldots p_n], ret) \in \mathsf{mda}$, and*
2. *for all method names of entry points $mid \in \mathsf{EP}_P$, methods $m \in \mathcal{M}_P$, classes $c \in \mathcal{CID}_P$, and security domains $p_0, \ldots p_n, ret \in \mathcal{SL}$ such that $(mid, [p_0, \ldots p_n], ret) \in \mathsf{mda}$, if*
   - $m = \textsf{lookup-static}(mid)$,
   - $m = \textsf{lookup-direct}(mid, c)$,
   - $m = \textsf{lookup-super}(mid, c)$, *or*
   - $m = \textsf{lookup-virtual}(mid, c)$

   holds, then m must satisfy TIN-ADL with respect to $(mid, [p_0, \ldots p_n], ret)$.

Intuitively, P satisfies *TIN-ADL* if and only if any method that may be called on any object to execute the program does not reveal private information to the attacker.

4 Security Type System

In this section, we present a security-type system that facilitates the certification of *TIN-ADL* for ADL programs.

The type system not only captures direct data flows through instructions with assignments but also indirect information flows through control flow instructions with branching conditions that depend on private information. To this end, we utilize the concept of control dependence regions to determine control-flow dependencies between the instructions of a method, following the approach of Barthe, Pichardie, and Rezk [BPR07] who applied it to Java bytecode.

Definition 25 (Successor relation). *Let $m \in \mathcal{M}$ be an arbitrary method. The successor relation $\rightarrow_m \subseteq \mathbb{N}_0 \times \mathbb{N}_0$ of method m is defined such that for all program points $i, j \in \mathbb{N}_0$, it holds that $i \rightarrow_m j$ if and only if program point j of method m is possibly executed directly after the execution of program point i with respect to the semantics of the instruction at program point i.*

Intuitively, the relation $\rightarrow_m$ specifies the control flow graph of method m. Based on the control flow graph, the control flow dependency between instructions of a method can be approximated.

Definition 26 (Control dependence region). *Let $m \in \mathcal{M}$ be an arbitrary method. The functions $region_m : \mathbb{N}_0 \rightarrow \mathcal{P}(\mathbb{N}_0)$ and $jun_m : \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$ are a safe over approximation of the method's control dependence regions if they satisfy the three safe over approximation properties (SOAPs):*

SOAP1 *For all program points $i, j, k \in \mathbb{N}_0$ such that $i \rightarrow_m j$, $i \rightarrow_m k$, and $j \neq k$, $k \in region_m(i)$ or $k = jun_m(i)$.*
SOAP2 *For all program points $i, j, k \in \mathbb{N}_0$, if $j \in region_m(i)$ and $j \rightarrow_m k$, then either $k \in region_m(i)$ or $k = jun_m(i)$.*
SOAP3 *For all program points $i, j \in \mathbb{N}_0$, if $j \in region_m(i)$ and there exists no $k \in \mathbb{N}_0$ such that $j \rightarrow_m k$, then $jun_m(i)$ is undefined.*

The control dependence region of a program point *pp* with a branching instruction, $region_m(pp)$, contains at least those program points that are executed depending on what the branching condition evaluates to. The junction point corresponding to *pp*, $jun_m(pp)$, specifies an instruction that is again executed independently of the evaluation of the branching condition. If the method returns in a control dependence region, this region does not have any junction points.

Remark 3. In the remainder of this report, we assume for all methods $m \in \mathcal{M}$ functions $region_m : \mathbb{N}_0 \rightarrow \mathcal{P}(\mathbb{N}_0)$ and $jun_m : \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$ such that $region_m$ and jun_m are a safe over approximation of m's control dependence regions.

To prevent information leaks due to control flow dependencies, the security type system ensures that no assignments to public storage locations are made at any program point in the control dependence region of a program point with a branching instruction that has a condition depending on private information. For this purpose, the security environment *se* records for each program point the upper bound of the security domains of all information that determines whether the respective program point is executed or not.

Definition 27 (Security environment.). *A security environment is a function $se : \mathbb{N}_0 \rightarrow \mathcal{SL}$.*

4.1 Security Typing Rules

The judgment $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}'$ is parametric in the method m, the control dependence region $region_m$, the set of method signatures mda, the field domain assignment fda, the security domain of all array contents ada, the domain $ret \in \mathcal{SL}$ of the return value of m, and a security environment se. The judgment denotes that after the execution of program point pp in the context of $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se$, the security domain assignment of the registers must be rda' if it was rda before. We abbreviate long judgments by $m, \dots \vdash pp : \mathsf{rda} \to \mathsf{rda}'$. The typing rules of the form

$$\text{tName}\frac{premise_1 \quad \dots \quad premise_n}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}'}$$

are introduced in the following.

Security typing rules for arithmetic instructions. The security typing rules for arithmetic instructions set the security domain of the target register to at least the highest security domain of any argument register of the computation, and at least to the security domain of the environment in which the instruction is executed. The former ensures that no direct information flows from the argument registers to the target register occur. The latter rules out indirect information flows through control-flow dependencies on private information: If a register is written in the control dependence region of a branching instruction that depends on private information (i.e., $se(pp) = high$), then treating the register as private ensures that the attacker cannot learn from its content which path in the control flow graph of the method was executed and, thus, what the value of the private branching condition was. Constant values, as in (tConst), have a *low* security domain, as the attacker is assumed to know the program.

$$\text{tMove}\frac{m[pp] = \texttt{move}\ \ v_a, v_b}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto \mathsf{rda}(v_b) \sqcup se(pp)]}$$

$$\text{tConst}\frac{m[pp] = \texttt{const}\ \ v_a, n}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto se(pp)]}$$

$$\text{tUnop}\frac{m[pp] = \texttt{unop}\ \ v_a, v_b, uop}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto \mathsf{rda}(v_b) \sqcup se(pp)]}$$

$$\text{tBinop}\frac{m[pp] = \texttt{binop}\ \ v_a, v_b, v_c, bop \qquad t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

Figure 7. Security typing rules for arithmetic instructions

Security typing rules for control flow instructions. The instructions `nop` and `goto` cannot leak information since they are statically known to the attacker and their execution does not depend on additional information from the memory. The only control flow instructions that may leak information are branchings on private information. To ensure that they do not cause indirect leaks through which branch they execute, the security environment of all program points in the control dependence region of a branching instruction must have at least the highest security domain of all source registers. If a program point is in a high security environment, then its instruction is forbidden to make assignments (e.g., to fields, arrays, method parameters) that may eventually become visible to the attacker.

$$\text{tNop}\ \frac{m[pp] = \texttt{nop}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}}$$

$$\text{tGoto}\ \frac{m[pp] = \texttt{goto}\ n}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}}$$

$$\text{tIfTest}\ \frac{\begin{array}{c} m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \\ \forall j \in region_m(pp).\mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_b) \sqsubseteq se(j) \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}}$$

Figure 8. Security typing rules for control flow instructions

Security typing rules for object-related instructions. The rules (tNewInstance), (tConstString), and (tConstClass) resemble the rule for loading constant numbers. Similarly, the value to be stored in the target register is statically known and, thus, only information about the control flow could be leaked, which is prevented by raising the domain of the target register to the domain of the security environment. The typing rule (tIget) sets the domain of the target register to the least upper bound of the security environment, the source field, and the register holding the reference to the source object. The domain of the register that holds the object reference is incorporated into the security domain of the target register, because reading the value from a field also reveals the instance behind the reference, which may have been set depending on private information. Incorporating the security domain of the field and the security domain of the security environment prevents the usual direct and indirect information flows, respectively. The security typing rule (tIput) ensures that the security domain of the target field is at least as high as the highest domain of the register holding the object reference, the source register, and the security environment. As for (tIget), incorporating the domain of the register with the object reference prevents indirect leaks into the field through aliasing. Incorporating the domains of the source register and the security environment prevent direct and indirect information flows, respectively. The rules (tSget) and (tSput) are analogous to the rules for instances but without the security domain of the register holding the object reference.

$$\text{tInstOf}\ \frac{m[pp] = \texttt{instance-of}\ \ v_a, v_b, cl}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto \mathsf{rda}(v_b) \sqcup se(pp)]}$$

$$\text{tNewInstance}\ \frac{m[pp] = \texttt{new-instance}\ \ v_a, cl}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto se(pp)]}$$

$$\text{tConstString}\ \frac{m[pp] = \texttt{const-string}\ \ v_a, s}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto se(pp)]}$$

$$\text{tConstClass}\ \frac{m[pp] = \texttt{const-class}\ \ v_a, cl}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto se(pp)]}$$

$$\text{tIget}\ \frac{m[pp] = \texttt{iget}\ \ v_a, v_b, fid \qquad \mathsf{fda}(fid) = st \qquad t = \mathsf{rda}(v_b) \sqcup st \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

$$\text{tIput}\ \frac{\begin{array}{c} m[pp] = \texttt{iput}\ \ v_a, v_b, fid \qquad \mathsf{fda}(fid) = st \\ \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_b) \sqcup se(pp) \sqsubseteq st \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

$$\text{tSget}\ \frac{m[pp] = \texttt{sget}\ \ v_a, fid \qquad \mathsf{fda}(fid) = st}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto st \sqcup se(pp)]}$$

$$\text{tSput}\ \frac{m[pp] = \texttt{sput}\ \ v_a, fid \qquad \mathsf{fda}(fid) = st \qquad \mathsf{rda}(v_a) \sqcup se(pp) \sqsubseteq st}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

Figure 9. Security typing rules for object-related instructions

Security typing rules for array-related instructions. The security typing rules (tAget) and (tAput) are very similar to the rules (tIget) and (tIput) for fields of objects. However, array operations do not have a constant field name as parameter but address the fields of arrays by dynamic index values stored in registers. Hence, (tAget) and (tAput) in addition ensure the privacy of the index in register v_c. Rule (tNewA) sets the security domain of a register storing a newly created array to the least upper bound of the domain of the security environment and the length of the array. This is because the initial length, stored in register v_b, may be private. For (tFNAR), the initial length of the array is statically known, but since the array's content is initialized with values from registers, the rule ensures that the domain of the content ada is at least as high as the least upper bound of the security domains of the argument registers.

$$\text{tALength}\frac{m[pp] = \texttt{array-length}\ v_a, v_b \qquad t = \mathsf{rda}(v_b) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}[v_a \mapsto t]}$$

$$\text{tNewA}\frac{m[pp] = \texttt{new-array}\ v_a, v_b}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}[v_a \mapsto \mathsf{rda}(v_b) \sqcup se(pp)]}$$

$$\text{tFNAR}\frac{m[pp] = \texttt{filled-new-array-range}\ v_k, n \qquad \bigsqcup_{i=k}^{k+n-1} \mathsf{rda}(v_i) \sqsubseteq \mathsf{ada}}{m, \cdots \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}[result_{lower} \mapsto se(pp), result_{upper} \mapsto se(pp)]}$$

$$\text{tAget}\frac{m[pp] = \texttt{aget}\ v_a, v_b, v_c \qquad t = se(pp) \sqcup \mathsf{ada} \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}[v_a \mapsto t]}$$

$$\text{tAput}\frac{m[pp] = \texttt{aput}\ v_a, v_b, v_c \qquad \mathsf{rda}(v_a) \sqcup se(pp) \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqsubseteq \mathsf{ada}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}}$$

Figure 10. Security typing rules for array-related instructions

Security typing rules for method-related instructions. Rules for method invocation ensure that the called method supports a signature with the respective security domains of the parameter registers. If there exists such a signature, the security domain of the result registers is set to the declared return type of the signature. To prevent indirect leaks due to observable effects of the method calls on the heap, methods may only be called in a low security environment and, in case of instance methods, on objects that are stored in a public register. The rule (tReturn) ensures that the declared security domain of the return value of the current method is at least the least upper bound of the security domain of the environment and the domain of the register that contains the return value.

Security typing rules for conversion instructions for 64 bit values. In case a 64 bit value is reduced to a 32 bit value (tUnopWS), the security domain of the target register is set to the least upper bound of the domains of the two registers containing the 64 bit value and the domain of the security environment. In case of converting a 32 bit value to a 64 bit value, the two target registers are set to the same domain, the least upper bound of the domains of the source register and the security environment.

The typing rules for 64 bit instructions and the remaining instructions from Definition 3 are listed in Appendix D.

$$\text{tIR}\,\frac{\begin{array}{c} m[pp] = \texttt{invoke-virtual-range}\ v_k, n, mid \\ (mid, [\mathsf{rda}(v_k), \ldots, \mathsf{rda}(v_{k+n-1})], st) \in \mathsf{mda} \\ se(pp) = low \qquad \mathsf{rda}(v_k) = low \end{array}}{m, \cdots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto st, result_{upper} \mapsto st]}$$

$$\text{tIRS}\,\frac{\begin{array}{c} m[pp] = \texttt{invoke-static-range}\ v_k, n, mid \\ (mid, [\mathsf{rda}(v_k), \ldots, \mathsf{rda}(v_{k+n-1})], st) \in \mathsf{mda} \qquad se(pp) = low \end{array}}{m, \cdots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto st, result_{upper} \mapsto st]}$$

$$\text{tMoveRes}\,\frac{m[pp] = \texttt{move-result}\ v_a \qquad t = se(pp) \sqcup \mathsf{rda}(result_{lower})}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

$$\text{tReturnVoid}\,\frac{m[pp] = \texttt{return-void}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

$$\text{tReturn}\,\frac{m[pp] = \texttt{return}\ v_a \qquad se(pp) \sqcup \mathsf{rda}(v_a) \sqsubseteq ret}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

Figure 11. Security typing rules for method-related instructions

$$\text{tUnopWS}\,\frac{m[pp] = \texttt{unop-wideS}\ v_a, v_b, uop \qquad t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_{b+1}) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

$$\text{tUnopWT}\,\frac{m[pp] = \texttt{unop-wideT}\ v_a, v_b, uop \qquad t = \mathsf{rda}(v_b) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

Figure 12. Security typing rules for conversion instructions for 64 bit values

4.2 Typable Methods and Programs

A method is typable if there exists a suitable declaration of the security environment and a register domain assignment for each program point, such that for each potential step in program execution a judgment can be derived in the security type system for the respective program point.

Definition 28 (Typable method). *Let $m \in \mathcal{M}_P$ be an arbitrary method of program P with $\mathsf{length}(m) = k+1$ for some $k \in \mathbb{N}_0$. Moreover, let $mid \in \mathcal{MID}_P$, and $p_0, \ldots p_n, ret \in \mathcal{SL}$ for some $n \in \mathbb{N}_0$ such that $(mid, [p_0, \ldots, p_n], ret) \in \mathsf{mda}$.*

The method m is typable with respect to the signature $(mid, [p_0, \ldots, p_n], ret)$ if and only if there exist a security environment $se : \mathbb{N}_0 \to \mathcal{SL}$ and register domain assignments $\mathsf{rda}_0, \ldots, \mathsf{rda}_k \in \mathcal{RDA}$ such that

1. *for all $i \in \mathbb{N}_0$ with $i \leq n$ it holds that $p_i \sqsubseteq \mathsf{rda}_0(v_i)$,*
2. *for all $i, j \in \mathbb{N}_0$, if $i \to_m j$ then there exists a register domain assignment $\mathsf{rda}'_j \in \mathcal{RDA}$ such that $\mathsf{rda}'_j \sqsubseteq \mathsf{rda}_j$ and the judgment*

$$m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}'_j$$

   is derivable, and

3. *for all $i \in \mathbb{N}_0$, if there exists no $j \in \mathbb{N}_0$ such that $i \to_m j$, then the judgment*

$$m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}_i$$

   is derivable.

The first condition ensures that the method treats the parameters given in the initial register state at least as confidential as they have been declared in the method signature. The second condition requires that a typing rule is applicable for each possible transition between program points i and j in the method such that the register domain assignment resulting from the derivable judgment rda'_j is not more restrictive than the fixed register domain assignment rda_j. The third condition requires that a typing rule can be applied for each return instruction in the method.

To allow for the use of methods that are not part of the analyzed program itself, e.g., methods of the Android framework, $\mathsf{framework} \subseteq \mathcal{M}$ specifies a set of trusted methods. The set $\mathsf{framework}$ contains only those methods that can be safely assumed to satisfy *TIN-ADL* with respect to all applicable signatures in mda, e.g., after careful manual inspection.

An ADL program is typable with respect to a security policy if each of its methods is typable with respect to all method signatures that could possibly apply to the respective method. Each entry point of the program must have at least one corresponding method signature (see Definition 24) to make sure that all input and output of the program is classified into one of the security domains, regardless which entry point is called to start the program.

Definition 29 (Typable program). *The program P is typable if and only if*

1. *for all method names of entry points $mid \in \mathsf{EP}_P$ there exists $p_0, \ldots p_n, ret \in \mathcal{SL}$ for some $n \in \mathbb{N}_0$ such that $(mid, [p_0, \ldots p_n], ret) \in \mathsf{mda}$,*
2. *for all field names $fid_1, fid_2 \in \mathcal{FID}_P$, it holds that if $\textit{lookup-field}_P(fid_1) = \textit{lookup-field}_P(fid_2)$, then $\mathsf{fda}(fid_1) = \mathsf{fda}(fid_2)$, and*
3. *for all method names $mid \in \mathcal{MID}_P$, methods $m \in \mathcal{M}_P$, class names $c \in \mathcal{CID}_P$, and security domains $p_0, \ldots p_n, ret \in \mathcal{SL}$ with $(mid, [p_0, \ldots p_n], ret) \in \mathsf{mda}$, if*
   - $m = \textit{lookup-static}(mid)$,
   - $m = \textit{lookup-direct}(mid, c)$,
   - $m = \textit{lookup-super}(mid, c)$, *or*
   - $m = \textit{lookup-virtual}(mid, c)$,

   then $m \in \mathsf{framework}$ or m is typable with respect to $(mid, [p_0, \ldots p_n], ret)$.

The first condition requires that each entry point of the program has at least one declared method signature. The second condition ensures that all field identifiers that could refer to the same field must have the same security domain. The third condition requires that all methods which the name of a signature could refer to must be typable with respect to that signature or be a framework method.

5 Soundness

In this section, we establish the formal guarantee that if a program is typable in the security type system from Section 4, then it also satisfies the security condition *TIN-ADL* from Section 3.

Theorem 1 (Soundness of the type system). *If program P is typable, then P satisfies TIN-ADL.*

The proof of this theorem is inspired by [BPR08]. It depends on lemmas with the following intuition about the execution of typable programs:

Locally respect (Lemmas 1, 2, 3) If the same program point is executed in a low security environment with indistinguishable heaps and register states, then the resulting heap and register states are also indistinguishable. These lemmas ensure that no private data is copied to public storage locations.

Step consistent (Lemma 4, 5) Heap and register state before and after the execution of a program point in a high security environment are indistinguishable. This implies that, in a high security environment, there are no observable information flows to the heap, to the register state, or to return values. These lemmas ensure, together with high branching, that no information is leaked implicitly through control flow dependencies on branching conditions with secrets.

High branching (Lemma 6) All program points in control-flow dependence of a branching based on private information are in a high security environment. This lemma ensures that all control flow dependencies on branching conditions with secrets are taken into account.

Indistinguishable after high branch (Lemma 7) Executing sequences of program points that are all in a high security environment starting with indistinguishable heaps and register states do not affect the indistinguishability of the heaps and register states at any point in execution. This lemma ensures that executions in a high security environment, i.e., depending on secrets, have no observable effect.

Security of typable sequences (Lemma 8) For arbitrary two execution sequences from the same program point with indistinguishable initial register states and heaps, each state with a program point in a low security environment of one execution sequence has a matching state in the second execution sequence with indistinguishable heaps and register states. This lemma ensures that the same potentially observable steps are executed in two independent runs of a method starting from indistinguishable inputs.

Security of typable methods (Lemma 9) If a method of a typable program is typable with respect to a given method signature, then it satisfies *TIN-ADL* with respect to the same signature.

Moreover, we utilize in some proofs that all indistinguishability relations are equivalence relations, which is shown in Appendix A.

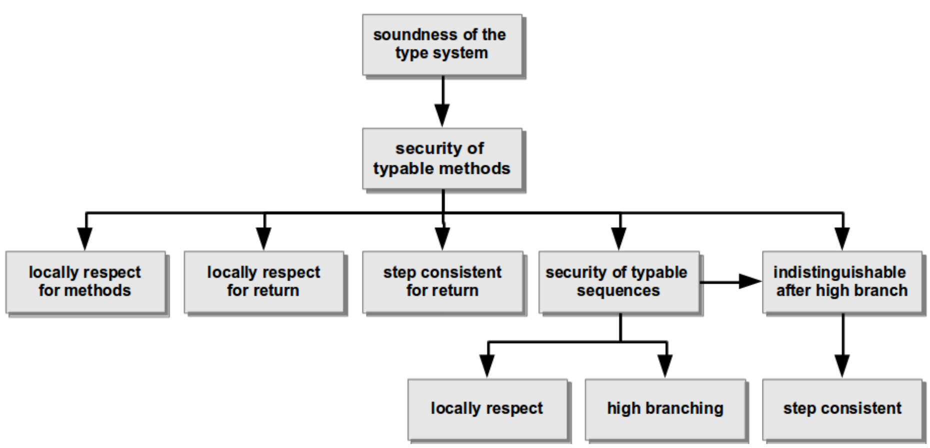

Figure 13. Proof Structure

Figure 13 shows an overview of the dependencies between the lemmas in the proof of soundness. The arrowhead denotes on which lemma the lemma from which the arrow originates depends. We show the basic lemmas, i.e., those without dependencies, with respect to instructions that are representatives for groups of similar instructions. All remaining instructions can be shown analogously.

Lemma 1 (Locally respect). *For all methods $m \in \mathcal{M}_P$ of a typable program P, register states $r_1, r_2, r_1', r_2' \in \mathcal{R}$, heaps $h_1, h_2, h_1', h_2' \in \mathcal{H}$, program points $pp_1, pp_2, pp_2' \in \mathbb{N}_0$, partial injective functions on locations $\beta \in \mathcal{B}$, register domain assignments $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$, security environments $se : \mathbb{N}_0 \to \mathcal{SL}$, and security domains $ret \in \mathcal{SL}$, if*

1. $se(pp_1) = low$,
2. $h_1 \sim_\beta h_1'$,
3. $r_1 \sim_{\beta,\mathsf{rda}} r_1'$,
4. $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$,
5. $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(0)}_{P,m} \langle h_2, pp_2, r_2 \rangle$, *and*
6. $\langle h_1', pp_1, r_1' \rangle \rightsquigarrow^{(0)}_{P,m} \langle h_2', pp_2', r_2' \rangle$,

then there exists some $\beta' \in \mathcal{B}$ with $\beta \subseteq \beta'$ such that $h_2 \sim_{\beta'} h_2'$ and $r_2 \sim_{\beta',\mathsf{rda}'} r_2'$.

Proof. Let $m \in \mathcal{M}_P$ be a method of a typable program P, $r_1, r_2, r_1', r_2' \in \mathcal{R}$ be register states, $h_1, h_2, h_1', h_2' \in \mathcal{H}$ be heaps, $pp_1, pp_2, pp_2' \in \mathbb{N}_0$ be program points, $\beta \in \mathcal{B}$ be partial injective functions on locations, $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$ be register domain assignments, $se : \mathbb{N}_0 \to \mathcal{SL}$ be security environments, and $ret \in \mathcal{SL}$ be security domains such that $se(pp_1) = low$, $h_1 \sim_\beta h_1'$, $r_1 \sim_{\beta,\mathsf{rda}} r_1'$, $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$, $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(0)}_{P,m} \langle h_2, pp_2, r_2 \rangle$, and $\langle h_1', pp_1, r_1' \rangle \rightsquigarrow^{(0)}_{P,m} \langle h_2', pp_2', r_2' \rangle$.

To show that there exists some $\beta' \in \mathcal{B}$ with $\beta \subseteq \beta'$ such that $h_2 \sim_{\beta'} h_2'$ and $r_2 \sim_{\beta',\mathsf{rda}'} r_2'$, we distinguish cases over the different instructions. For convenience, we repeat the respective semantic rules and typing rules at the beginning of each case.

Case 1 (`binop` v_a, v_b, v_c, bop).

$$\text{rBinop}\frac{m[pp] = \texttt{binop}\ \ v_a, v_b, v_c, bop \qquad x = r(v_b)\ \underline{bop}\ \ r(v_c)}{\langle h, pp, r \rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto x] \rangle}$$

$$\text{tBinop}\frac{m[pp] = \texttt{binop}\ \ v_a, v_b, v_c, bop \qquad t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

As the heap does not change, $\beta' = \beta$ and $h_1 \sim_\beta h_1'$ implies $h_2 \sim_{\beta'} h_2'$.

We need to show that $r_2 \sim_{\beta,\mathsf{rda}'} r_2'$. The only register that is updated in the register state and the register domain assignment is v_a. Hence, given that $r_1 \sim_{\beta,\mathsf{rda}} r_1'$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r_2(v_a) \sim_\beta r_2'(v_a)$ to have $r_2 \sim_{\beta,\mathsf{rda}'} r_2'$.

Assume $\mathsf{rda}'(v_a) = low$, then $\mathsf{rda}(v_b) = low$ and $\mathsf{rda}(v_c) = low$ because otherwise t in (tBinop) would be *high*. Since the operators represented by the symbols in $\mathcal{BINOP}$ all operate on numbers, v_b and v_c must store numbers. With

$r_1 \sim_{\beta,\mathsf{rda}} r_1'$, we have $r_1(v_b) \sim_\beta r_1'(v_b)$ and $r_1(v_c) \sim_\beta r_1'(v_c)$. Thus, $r_1(v_b) = r_1'(v_b)$ and $r_1(v_c) = r_1'(v_c)$ by the definition of indistinguishability of values. Since $\underline{bop}$ is a function, we have $r_1(v_b)\ \underline{bop}\ r_1(v_c) = r_1'(v_b)\ \underline{bop}\ r_1'(v_c)$ and therefore $r_2(v_a) \sim_\beta r_2'(v_a)$ holds.

Case 2 (`if-test` v_a, v_b, n, rop).

$$\text{rIfTestTrue}\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad r(v_a)\ \underline{rop}\ r(v_b)}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+n, r\rangle}$$

$$\text{rIfTestFalse}\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad \neg(r(v_a)\ \underline{rop}\ r(v_b))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r\rangle}$$

$$\text{tIfTest}\frac{\begin{array}{c} m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \\ \forall j \in region_m(pp).\mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_b) \sqsubseteq se(j)\end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}}$$

Both potentially applicable rules (rIfTestTrue) and (rIfTestFalse) do not modify the heap or register set. Hence $\beta' = \beta$, $h_1 = h_2$, $h_1' = h_2'$, $r_1 = r_2$, and $r_1' = r_2'$. Since the register security domains are also not modified in the applicable typing rule (tIfTest), i.e., $\mathsf{rda} = \mathsf{rda}'$, we know from $h_1 \sim_\beta h_1'$ that $h_2 \sim_{\beta'} h_2'$ and from $r_1 \sim_{\beta,\mathsf{rda}} r_1'$ that $r_2 \sim_{\beta',\mathsf{rda}'} r_2'$.

Case 3 (`new-instance` v_a, cl).

$$\text{rNewInstance}\frac{\begin{array}{c} m[pp] = \texttt{new-instance}\ v_a, cl \qquad h \in dom(\mathsf{nextFreeLocation}) \\ l = \mathsf{nextFreeLocation}(h)\end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[l \mapsto \mathsf{defaultObject}(cl)], pp+1, r[v_a \mapsto l]\rangle}$$

$$\text{tNewInstance}\frac{m[pp] = \texttt{new-instance}\ v_a, cl}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}[v_a \mapsto se(pp)]}$$

Let $l = \mathsf{nextFreeLocation}(h_1)$, $l' = \mathsf{nextFreeLocation}(h_1')$, and $\beta' = \beta[l \mapsto l']$. Since $\mathsf{nextFreeLocation}$ allocates fresh locations on the heap provided as argument, we know that $l \notin dom(h_1)$ and $l' \notin dom(h_1')$. Hence, knowing that β was a partial injective function, β' is still a partial injective function, and $\beta \subseteq \beta'$. Moreover, as $\mathsf{nextFreeLocation}$ only returns variable locations, $\forall l \in \mathcal{L}_c.\ \beta'(l) = l$ holds. Thus, $\beta' \in \mathcal{B}$.

We first show $h_2 \sim_{\beta'} h_2'$. From $h_1 \sim_\beta h_1'$, we know that $dom(\beta) \subseteq dom(h_1)$ and $rng(\beta) \subseteq dom(h_1')$, and by (rNewInstance), $dom(h_2) = dom(h_1) \cup \{l\}$ and $dom(h_2') = dom(h_1') \cup \{l'\}$. Moreover, from the definition of β', we know that $dom(\beta') = dom(\beta) \cup \{l\}$ and $rng(\beta') = rng(\beta) \cup \{l'\}$. Ultimately, we can conclude $dom(\beta') \subseteq dom(h_2)$ and $rng(\beta') \subseteq dom(h_2')$.

From $h_2(l) = h_2'(l') = \mathsf{defaultObject}(cl)$ follows $h_2(l) \sim_{\beta'} h_2'(l')$, and since $h_2(l), h_2'(l')$ store objects at the new locations l, l', we have $l \in dom_{\mathcal{O}}(h_2)$ and $l' \in dom_{\mathcal{O}}(h_2')$. Hence, with $h_1 \sim_\beta h_1'$ we can conclude that for all $l \in dom(\beta')$, either

$l \in dom_{\mathcal{A}}(h_2)$, $\beta'(l) \in dom_{\mathcal{A}}(h'_2)$, and $h_2(l) \sim_{\beta'} h'_2(\beta'(l))$, or $l \in dom_{\mathcal{O}}(h_2)$, $\beta'(l) \in dom_{\mathcal{O}}(h'_2)$, and $h_2(l) \sim_{\beta'} h'_2(\beta'(l))$. Hence, we have $h_2 \sim_{\beta'} h'_2$.

We still need to show that $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$. According to the rules (rNewInstance) and (tNewInstance), the only register that is updated in the register state and the register domain assignment is v_a, i.e., $r_2 = r_1[v_a \mapsto l]$ and $r'_2 = r'_1[v_a \mapsto l']$. Given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$ and β' equals β except for the additional point (l, l'), we only need to show that $r_2(v_a) \sim_{\beta'} r'_2(v_a)$. Since, $\beta'(l) = l'$ we have $l \sim_{\beta'} l'$ and, thus, $r_2(v_a) \sim_{\beta'} r'_2(v_a)$.

Case 4 (`const-string` v_a, s).

$$\text{rConstString}\frac{m[pp] = \ \texttt{const-string}\ \ v_a, s \qquad s \in dom(\mathsf{nameToReference})}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{nameToReference}(s)]\rangle}$$

$$\text{tConstString}\frac{m[pp] = \ \texttt{const-string}\ \ v_a, s}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto se(pp)]}$$

As the heap does not change due to the semantics of `const-string`, $\beta' = \beta$ and $h_1 \sim_\beta h'_1$ implies $h_2 \sim_{\beta'} h'_2$.

We still need to show that $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$ where $\mathsf{rda}' = \mathsf{rda}[v_a \mapsto se(pp_1)]$ by (tConstString), i.e., $\mathsf{rda}' = \mathsf{rda}[v_a \mapsto low]$. The only register that is updated in the register state is v_a. Given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$ and $\beta' = \beta$, we only need to show that $r_2(v_a) \sim_{\beta'} r'_2(v_a)$. Since $r_2(v_a) = \mathsf{nameToReference}(s) = r'_2(v_a)$ by rule (rConstString), this reduces to showing $l \sim_{\beta'} l$. From $\beta' = \beta$ and $\beta(l) = l$ for all $l \in \mathcal{L}_c$, we have $l \sim_{\beta'} l$.

Case 5 (`iget` v_a, v_b, fid).

$$\text{rIget}\frac{\begin{array}{cc} m[pp] = \ \texttt{iget}\ \ v_a, v_b, \mathit{fid} & \mathit{fid} \in dom(\mathsf{lookup\text{-}field}_P) \\ r(v_b) \in dom(h) & o = h(r(v_b)) \\ f = \mathsf{lookup\text{-}field}_P(\mathit{fid}) & f \in dom(o.\mathsf{fields}) \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto o.f]\rangle}$$

$$\text{tIget}\frac{m[pp] = \ \texttt{iget}\ \ v_a, v_b, fid \qquad \mathsf{fda}(fid) = st \qquad t = \mathsf{rda}(v_b) \sqcup st \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

As the heaps are not changed by the semantics of `iget`, we have $\beta' = \beta$ and $h_1 \sim_\beta h'_1$ implies $h_2 \sim_{\beta'} h'_2$.

It remains to show that $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$. The only register that is updated in the register state and the register domain assignment is v_a. Hence, given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r_2(v_a) \sim_\beta r'_2(v_a)$ to have $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$.

Assume $\mathsf{rda}'(v_a) = low$, then $\mathsf{rda}(v_b) = low$ and $\mathsf{fda}(\mathit{fid}) = st = low$ by the premise of the typing rule (tIget). Then we have $r_1(v_b) \sim_\beta r'_1(v_b)$ because $r_1 \sim_{\beta,\mathsf{rda}} r'_1$. As $r_1(v_b), r'_1(v_b) \in \mathcal{L}$, this implies that $\beta(r_1(v_b)) = r'_1(v_b)$. As $h_1 \sim_\beta h'_1$, we know by the definition of indistinguishability of heaps that

$h_1(r_1(v_b)) \sim_\beta h'_1(\beta(r_1(v_b)))$. With objects $o, o' \in \mathcal{O}$ such that $o = h_1(r_1(v_b))$ and $o' = h'_1(r'_1(v_b))$, we have that $o \sim_\beta o'$. By the definition of object indistinguishability, $o \sim_\beta o'$, $\mathsf{fda}(\mathit{fid}) = \mathit{low}$, and $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ follows that $o.f \sim_\beta o'.f$. Hence, $r_2(v_a) \sim_\beta r'_2(v_a)$.

*Case 6 (**iput** v_a, v_b, fid).*

$$\text{rIput}\frac{\begin{array}{cc} m[pp] = \texttt{iput}\ v_a, v_b, \mathit{fid} & \mathit{fid} \in \mathit{dom}(\mathsf{lookup\text{-}field}_P) \\ r(v_b) \in \mathit{dom}(h) & o = h(r(v_b)) \\ f = \mathsf{lookup\text{-}field}_P(\mathit{fid}) & f \in \mathit{dom}(o.\mathsf{fields}) \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[r(v_b) \mapsto o[f \mapsto r(v_a)]], pp+1, r\rangle}$$

$$\text{tIput}\frac{\begin{array}{c} m[pp] = \texttt{iput}\ v_a, v_b, \mathit{fid} \quad \mathsf{fda}(\mathit{fid}) = st \\ \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_b) \sqcup se(pp) \sqsubseteq st \end{array}}{m, \mathit{region}_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, \mathit{ret}, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

As no new objects or arrays are created, $\beta' = \beta$. The register states and register domain assignments are not changed, so we have $\mathsf{rda} = \mathsf{rda}'$ and $r_1 \sim_{\beta,\mathsf{rda}} r'_1$ implies $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$.

It remains to show that $h_2 \sim_\beta h'_2$. In the following, let $o_1 = h_1(r_1(v_b))$, $o'_1 = h'_1(r'_1(v_b))$, $o_2 = o_1[f \mapsto r_1(v_a)]$, and $o'_2 = o'_1[f \mapsto r'_1(v_a)]$. Since the field name fid is a constant in the bytecode and $\mathsf{lookup\text{-}field}_P$ is a function, $f = \mathsf{lookup\text{-}field}_P(\mathit{fid})$ is the same for all executions of the program point. According to rule (rIput), the only change to the respective heaps h_1, h'_1 is that the object o_1 at location $l = r_1(v_b)$, respectively the object o'_1 at location $l' = r'_1(v_b)$, is updated in the field f to o_2, respectively o'_2. Moreover, by the definition of heap indistinguishability, two heaps can only be distinguished by the instances that are at locations related by β. Hence, to show that $h_2 \sim_\beta h'_2$ given $h_1 \sim_\beta h'_1$, it remains to show that $h_2(l) \sim_\beta h'_2(\beta(l))$ if $l \in \mathit{dom}(\beta)$, and $h_2(\beta^{-1}(l')) \sim_\beta h'_2(l')$ if $l' \in \mathit{rng}(\beta)$. We distinguish two cases:

$\mathsf{fda}(\mathit{fid}) = \mathit{high}$. Since $h_1 \sim_\beta h'_1$ and objects can only be distinguished by public fields, changes to the private field fid leave the resulting objects indistinguishable. Moreover, from the typability of the program also follows that there are no other field names $\mathit{fid}' \in \mathcal{FID}$ such that $\mathsf{lookup\text{-}field}(\mathit{fid}') = f$ and $\mathsf{fda}(\mathit{fid}') = \mathit{low}$. That means $h_2(l) \sim_\beta h'_2(\beta(l))$ if $l \in \mathit{dom}(\beta)$ and $h_2(\beta^{-1}(l')) \sim_\beta h'_2(l')$ if $l' \in \mathit{rng}(\beta)$ trivially follow from $\mathsf{fda}(\mathit{fid}) = \mathit{high}$, $h_1(l) \sim_\beta h'_1(\beta(l))$, $h_1(\beta^{-1}(l')) \sim_\beta h'_1(l')$, and the definition of indistinguishability of objects.

$\mathsf{fda}(\mathit{fid}) = \mathit{low}$. Then $\mathsf{rda}(v_a) = \mathit{low}$ and $\mathsf{rda}(v_b) = \mathit{low}$ by the premise of (tIput). With $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, it follows that $r_1(v_b) \sim_\beta r'_1(v_b)$ and, thus, $\beta(l) = l'$, respectively $\beta^{-1}(l') = l$. Hence, we have to show that $h_2(l) \sim_\beta h'_2(l')$ given that $h_1(l) \sim_\beta h'_1(l')$. This is equivalent to showing $o_2 \sim_\beta o'_2$ given that $o_1 \sim_\beta o'_1$. Since $o_2 = o_1[f \mapsto r_1(v_a)]$ and $o'_2 = o'_1[f \mapsto r'_1(v_a)]$, we have to show that $r_1(v_a) \sim_\beta r'_1(v_a)$, which is fulfilled because $r_1 \sim_{\beta,\mathsf{rda}} r'_1$ and $\mathsf{rda}(v_a) = \mathit{low}$ by assumption.

Case 7 (`sget` v_a, fid).

$$\text{rSget}\frac{\begin{array}{c} m[pp] = \texttt{sget}\ v_a, \mathit{fid} \qquad \mathit{fid} \in \mathit{dom}(\mathsf{nameToReference}) \\ l = \mathsf{nameToReference}(\mathit{fid}) \qquad \mathit{fid} \in \mathit{dom}(\mathsf{lookup\text{-}field}_P) \\ f = \mathsf{lookup\text{-}field}_P(\mathit{fid}) \quad f \in \mathit{dom}(h(l).\mathsf{fields}) \quad u = h(l).f \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto u]\rangle}$$

$$\text{tSget}\frac{m[pp] = \texttt{sget}\ v_a, fid \qquad \mathsf{fda}(fid) = st}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto st \sqcup se(pp)]}$$

As the heaps are not changed by the semantics of `sget`, we have $\beta' = \beta$ and $h_1 \sim_\beta h'_1$ implies $h_2 \sim_{\beta'} h'_2$.

It remains to show that $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$. The only register that is updated in the register state and the register domain assignment is v_a. Hence, given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we only need to show if $\mathsf{rda}'(v_a) = \mathit{low}$ that $r_2(v_a) \sim_\beta r'_2(v_a)$ to have $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$.

Assume $\mathsf{rda}'(v_a) = \mathit{low}$, then $\mathsf{fda}(\mathit{fid}) = st = \mathit{low}$ by the premise of the typing rule (tSget). Moreover, let $l = \mathsf{nameToReference}(\mathit{fid})$ and $f = \mathsf{lookup\text{-}field}_P(\mathit{fid})$. As $h_1 \sim_\beta h'_1$, we know by the definition of indistinguishability of heaps and the assumption that $\beta(l) = l$ for all $l \in \mathcal{L}_c$, that $h_1(l) \sim_\beta h'_1(l)$. By the definition of object indistinguishability, $h_1(l) \sim_\beta h'_1(l)$, $\mathsf{fda}(\mathit{fid}) = \mathit{low}$, and $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ follows that $h_1(l).f \sim_\beta h'_1(l).f$. Hence, $r_2(v_a) \sim_\beta r'_2(v_a)$.

Case 8 (`sput` v_a, fid).

$$\text{rSput}\frac{\begin{array}{c} m[pp] = \texttt{sput}\ v_a, \mathit{fid} \quad \mathit{fid} \in \mathit{dom}(\mathsf{nameToReference}) \\ l = \mathsf{nameToReference}(\mathit{fid}) \quad \mathit{fid} \in \mathit{dom}(\mathsf{lookup\text{-}field}_P) \\ f = \mathsf{lookup\text{-}field}_P(\mathit{fid}) \quad o = h(l) \quad f \in \mathit{dom}(o.\mathsf{fields}) \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[l \mapsto o[f \mapsto r(v_a)]], pp+1, r\rangle}$$

$$\text{tSput}\frac{m[pp] = \texttt{sput}\ v_a, fid \qquad \mathsf{fda}(fid) = st \qquad \mathsf{rda}(v_a) \sqcup se(pp) \sqsubseteq st}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

As no new objects or arrays are created, $\beta' = \beta$. The register states and register domain assignments are not changed, so we have $\mathsf{rda} = \mathsf{rda}'$ and $r_1 \sim_{\beta,\mathsf{rda}} r'_1$ implies $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$.

It remains to show that $h_2 \sim_\beta h'_2$. In the following, let $f = \mathsf{lookup\text{-}field}_P(\mathit{fid})$, $l = \mathsf{nameToReference}(\mathit{fid})$, $o_1 = h_1(l)$, $o'_1 = h'_1(l)$, $o_2 = o_1[f \mapsto r_1(v_a)]$, and $o'_2 = o'_1[f \mapsto r'_1(v_a)]$. According to rule (rSput), the only change to the respective heaps h_1, h'_1 is that the object o_1 at location l, respectively the object o'_1 at location l, is updated in the field f to o_2, respectively o'_2. Moreover, by assumption, $\beta(l) = l$. Hence, to show that $h_2 \sim_\beta h'_2$ given $h_1 \sim_\beta h'_1$, it remains to show that $h_2(l) \sim_\beta h'_2(l)$. We distinguish two cases:

$\mathsf{fda}(\mathit{fid}) = \mathit{high}$. Since $h_1 \sim_\beta h'_1$ and objects can only be distinguished by public fields, changes to the private field fid leave the resulting objects indistinguishable. Moreover, from the typability of the program also follows that there are no other field names $fid' \in \mathcal{FID}$ such that $\mathsf{lookup\text{-}field}(fid') = f$ and $\mathsf{fda}(fid') = low$. That means $h_2(l) \sim_\beta h'_2(l)$ trivially follows from $\mathsf{fda}(fid) = high$, $h_1(l) \sim_\beta h'_1(l)$, and the definition of indistinguishability of objects.

$\mathsf{fda}(fid) = low$. Then $\mathsf{rda}(v_a) = low$ by the premise of (tSput). Showing that $h_2(l) \sim_\beta h'_2(l)$ given that $h_1(l) \sim_\beta h'_1(l)$ is equivalent to showing $o_2 \sim_\beta o'_2$ given that $o_1 \sim_\beta o'_1$. Since $o_2 = o_1[f \mapsto r_1(v_a)]$ and $o'_2 = o'_1[f \mapsto r'_1(v_a)]$, we have to show that $r_1(v_a) \sim_\beta r'_1(v_a)$, which is fulfilled because $r_1 \sim_{\beta,\mathsf{rda}} r'_1$ and $\mathsf{rda}(v_a) = low$ by assumption.

Case 9 (`new-array` v_a, v_b).

$$\text{rNewArray}\,\frac{m[pp] = \texttt{new-array}\ v_a, v_b \qquad h \in dom(\mathsf{nextFreeLocation}) \qquad l = \mathsf{nextFreeLocation}(h) \qquad 0 \le r(v_b)}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[l \mapsto \mathsf{defaultArray}(r(v_b))], pp+1, r[v_a \mapsto l]\rangle}$$

$$\text{tNewA}\,\frac{m[pp] = \texttt{new-array}\ v_a, v_b}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto \mathsf{rda}(v_b) \sqcup se(pp)]}$$

We distinguish cases over $\mathsf{rda}(v_b)$.

$\mathsf{rda}(v_b) = high$. Let $l = \mathsf{nextFreeLocation}(h_1)$, $l' = \mathsf{nextFreeLocation}(h'_1)$, and $\beta' = \beta$.

We first show $h_2 \sim_\beta h'_2$, which is equivalent to $h_2 \sim_{\beta'} h'_2$ because $\beta' = \beta$. From $h_1 \sim_\beta h'_1$, we know that $dom(\beta) \subseteq dom(h_1)$ and $rng(\beta) \subseteq dom(h'_1)$. Moreover, by rule (rNewArray), we know that $dom(h_2) = dom(h_1) \cup \{l\}$ and $dom(h'_2) = dom(h'_1) \cup \{l'\}$. Ultimately, we can conclude $dom(\beta) \subseteq dom(h_2)$ and $rng(\beta) \subseteq dom(h'_2)$. It remains to show for all locations $l \in dom(\beta)$ either $l \in dom_{\mathcal{A}}(h_2)$, $\beta(l) \in dom_{\mathcal{A}}(h'_2)$, and $h_2(l) \sim_\beta h'_2(\beta(l))$ or $l \in dom_{\mathcal{O}}(h_2)$, $\beta(l) \in dom_{\mathcal{O}}(h'_2)$, and $h_2(l) \sim_\beta h'_2(\beta(l))$.

As of rule (rNewArray) h_2 and h'_2 differ from h_1 and h'_1 only in location l and l', respectively. As $\mathsf{nextFreeLocation}$ allocates fresh locations on the heap provided as argument, we know that $l \notin dom(h_1)$ and $l' \notin dom(h'_1)$. Hence, for all locations $l \in dom(h_1)$ it holds that $h_1(l) = h_2(l)$ and for all locations $l' \in dom(h'_1)$ it holds that $h'_1(l') = h'_2(l')$. With $h_1 \sim_\beta h'_1$, we have for all locations $l \in dom(\beta)$ either $l \in dom_{\mathcal{A}}(h_2)$, $\beta(l) \in dom_{\mathcal{A}}(h'_2)$, and $h_2(l) \sim_\beta h'_2(\beta(l))$ or $l \in dom_{\mathcal{O}}(h_2)$, $\beta(l) \in dom_{\mathcal{O}}(h'_2)$, and $h_2(l) \sim_\beta h'_2(\beta(l))$.

We still need to show that $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$, which is equivalent to $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$ because $\beta' = \beta$. According to the rules (rNewArray) and (tNewA), the only register that is updated in the register state and the register domain assignment is v_a. Hence, given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r_2(v_a) \sim_\beta r'_2(v_a)$ to have $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$. From rule (tNewA) follows that $\mathsf{rda}'(v_a) = \mathsf{rda}(v_b) \sqcup se(pp_1)$. Since $\mathsf{rda}(v_b) = high$ by assumption, we have $\mathsf{rda}'(v_a) = high$. Thus, we have $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$.

$\mathsf{rda}(v_b) = low$. Let $l = \mathsf{nextFreeLocation}(h_1)$, $l' = \mathsf{nextFreeLocation}(h'_1)$, and $\beta' = \beta[l \mapsto l']$. Since $\mathsf{nextFreeLocation}$ allocates fresh locations on the heap provided as argument, we know that $l \notin dom(h_1)$ and $l' \notin dom(h'_1)$. Hence, knowing that β was a partial injective function, β' is still a partial injective function, and $\beta \subseteq \beta'$. Moreover, as $\mathsf{nextFreeLocation}$ only returns variable locations, $\forall l \in \mathcal{L}_c.\ \beta'(l) = l$ holds. Thus, $\beta' \in \mathcal{B}$.

We first show $h_2 \sim_{\beta'} h'_2$. From $h_1 \sim_\beta h'_1$, we know that $dom(\beta) \subseteq dom(h_1)$ and $rng(\beta) \subseteq dom(h'_1)$, and by (rNewArray), $dom(h_2) = dom(h_1) \cup \{l\}$ and $dom(h'_2) = dom(h'_1) \cup \{l'\}$. Moreover, from the definition of β', we know that $dom(\beta') = dom(\beta) \cup \{l\}$ and $rng(\beta') = rng(\beta) \cup \{l'\}$. Ultimately, we can conclude $dom(\beta') \subseteq dom(h_2)$ and $rng(\beta') \subseteq dom(h'_2)$. It remains to show that for all $l \in dom(\beta')$, either $l \in dom_{\mathcal{A}}(h_2)$, $\beta'(l) \in dom_{\mathcal{A}}(h'_2)$, and $h_2(l) \sim_{\beta'} h'_2(\beta'(l))$, or $l \in dom_{\mathcal{O}}(h_2)$, $\beta'(l) \in dom_{\mathcal{O}}(h'_2)$, and $h_2(l) \sim_{\beta'} h'_2(\beta'(l))$.

The heaps h_2 and h'_2 are the same as h_1 and h'_1 except for the locations l and l', respectively. Moreover, h_2 and h'_2 store arrays at the new locations l and l', i.e., $l \in dom_{\mathcal{A}}(h_2)$ and $l' \in dom_{\mathcal{A}}(h'_2)$. With $h_1 \sim_\beta h'_1$ we can conclude that for all $l \in dom(\beta')$, either $l \in dom_{\mathcal{A}}(h_2)$, $\beta'(l) \in dom_{\mathcal{A}}(h'_2)$, and $h_2(l) \sim_{\beta'} h'_2(\beta'(l))$, or $l \in dom_{\mathcal{O}}(h_2)$, $\beta'(l) \in dom_{\mathcal{O}}(h'_2)$, and $h_2(l) \sim_{\beta'} h'_2(\beta'(l))$ if we show that $h_2(l) \sim_{\beta'} h'_2(l')$ for the new locations l, l'. From the assumption $\mathsf{rda}(v_b) = low$ and $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we know that there exists an $n \in \mathbb{N}_0$ such that $r_1(v_b) = n = r'_1(v_b)$. From $h_2(l) = \mathsf{defaultArray}(r_1(v_b)) = \mathsf{defaultArray}(n)$ and $h'_2(l') = \mathsf{defaultArray}(r'_1(v_b)) = \mathsf{defaultArray}(n)$ follows $h_2(l) \sim_{\beta'} h'_2(l')$. Hence, we have $h_2 \sim_{\beta'} h'_2$.

We still need to show that $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$. According to the rules (rNewArray) and (tNewA), the only register that is updated in the register state and the register domain assignment is v_a, i.e., $r_2 = r_1[v_a \mapsto l]$ and $r'_2 = r'_1[v_a \mapsto l']$. Given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$ and β' equals β except for the additional point (l, l'), we only need to show that $r_2(v_a) \sim_{\beta'} r'_2(v_a)$. Since, $\beta'(l) = l'$ we have $l \sim_{\beta'} l'$ and, thus, $r_2(v_a) \sim_{\beta'} r'_2(v_a)$.

Case 10 (`filled-new-array-range` v_k, n).

$$\text{rFilledNewArrayR}\ \frac{\begin{array}{c} m[pp] = \texttt{filled-new-array-range}\ v_k, n \\ h \in dom(\mathsf{nextFreeLocation}) \\ l = \mathsf{nextFreeLocation}(h) \qquad x = \mathsf{defaultArray}(n) \\ ar = x[0 \mapsto r(v_k), \ldots, n-1 \mapsto r(v_{k+n-1})] \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[l \mapsto ar], pp+1, r[result_{lower} \mapsto l]\rangle}$$

$$\text{tFNAR}\ \frac{m[pp] = \texttt{filled-new-array-range}\ v_k, n \qquad \bigsqcup_{i=k}^{k+n-1} \mathsf{rda}(v_i) \sqsubseteq \mathsf{ada}}{m, \cdots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto se(pp), result_{upper} \mapsto se(pp)]}$$

Let $l = \mathsf{nextFreeLocation}(h_1)$, $l' = \mathsf{nextFreeLocation}(h'_1)$, and $\beta' = \beta[l \mapsto l']$. Since $\mathsf{nextFreeLocation}$ allocates fresh locations on the heap provided as argument, we know that $l \notin dom(h_1)$ and $l' \notin dom(h'_1)$. Hence, knowing that β was a partial injective function, β' is still a partial injective function, and $\beta \subseteq \beta'$. Moreover, as

$\mathsf{nextFreeLocation}$ only returns variable locations, $\forall l \in \mathcal{L}_c.\ \beta'(l) = l$ holds. Thus, $\beta' \in \mathcal{B}$.

From $h_1 \sim_\beta h'_1$, we know that $dom(\beta) \subseteq dom(h_1)$ and $rng(\beta) \subseteq dom(h'_1)$, and by (rFilledNewArrayR), $dom(h_2) = dom(h_1) \cup \{l\}$ and $dom(h'_2) = dom(h'_1) \cup \{l'\}$. Moreover, from the definition of β', we know that $dom(\beta') = dom(\beta) \cup \{l\}$ and $rng(\beta') = rng(\beta) \cup \{l'\}$. Ultimately, we can conclude $dom(\beta') \subseteq dom(h_2)$ and $rng(\beta') \subseteq dom(h'_2)$.

Since $h_2(l), h'_2(l')$ store arrays at the new locations l, l', we have $l \in dom_{\mathcal{A}}(h_2)$ and $l' \in dom_{\mathcal{A}}(h'_2)$. Hence, with $h_1 \sim_\beta h'_1$ we can conclude that for all $l \in dom(\beta')$, either $l \in dom_{\mathcal{A}}(h_2)$, $\beta'(l) \in dom_{\mathcal{A}}(h'_2)$, and $h_2(l) \sim_{\beta'} h'_2(\beta'(l))$, or $l \in dom_{\mathcal{O}}(h_2)$, $\beta'(l) \in dom_{\mathcal{O}}(h'_2)$, and $h_2(l) \sim_{\beta'} h'_2(\beta'(l))$ (i.e., $h_2 \sim_{\beta'} h'_2$) if we show $h_2(l) \sim_{\beta'} h'_2(l')$.

Let $x = \mathsf{defaultArray}(n)$. From (rFilledNewArrayR), we know that $h_2(l) = x[0 \mapsto r_1(v_k), \ldots, n-1 \mapsto r_1(v_{k+n-1})]$, and $h'_2(l') = x[0 \mapsto r'_1(v_k), \ldots, n-1 \mapsto r'_1(v_{k+n-1})]$. Hence, in any case, $h_2(l).\mathsf{length} = h'_2(l').\mathsf{length} = n$. It remains to show that if $\mathsf{ada} = low$, then $h_2(l)[i] \sim_{\beta'} h'_2(l)[i]$ for all indices $i \in \mathbb{N}_0$. If $\mathsf{ada} = low$, then $\mathsf{rda}(v_k) = \ldots = \mathsf{rda}(v_{k+n-1}) = low$. With $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we have that $r_1(v_i) \sim_\beta r'_1(v_i)$ for all indices $i \in \mathbb{N}_0$ such that $0 \leq i < h_2(l).\mathsf{length}$. Hence, we can conclude that $h_2(l) \sim_{\beta'} h'_2(l')$.

We still need to show that $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$ where $\mathsf{rda}'(result_{lower}) = low$ by (tFNAR). The only register that is updated in the register state and the register domain assignment is $result_{lower}$. Given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$ and β' equals β except for the point (l, l'), we only need to show that $r_2(result_{lower}) \sim_{\beta'} r'_2(result_{lower})$. Since, $\beta'(l) = l'$ we have $l \sim_{\beta'} l'$ and, thus, $r_2(result_{lower}) \sim_{\beta'} r'_2(result_{lower})$.

Case 11 (`aget` v_a, v_b, v_c*).*

$$\text{rAget}\frac{\begin{array}{ccc} m[pp] = \texttt{aget}\ v_a, v_b, v_c & r(v_b) \in dom(h) & ar = h(r(v_b)) \\ u = ar[r(v_c)] & & 0 \leq r(v_c) < ar.\mathsf{length} \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto u]\rangle}$$

$$\text{tAget}\frac{m[pp] = \texttt{aget}\ v_a, v_b, v_c \qquad t = se(pp) \sqcup \mathsf{ada} \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

As the heaps are not changed by the semantics of `aget`, we have $\beta' = \beta$ and $h_1 \sim_\beta h'_1$ implies $h_2 \sim_{\beta'} h'_2$.

It remains to show that $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$. The only register that is updated in the register state and the register domain assignment is v_a. Hence, given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r_2(v_a) \sim_\beta r'_2(v_a)$ to have $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$.

Assume $\mathsf{rda}'(v_a) = low$, then $\mathsf{rda}(v_b) = \mathsf{rda}(v_c) = low$ and $\mathsf{ada} = low$ by the premise of the typing rule (tAget). Then we have $r_1(v_b) \sim_\beta r'_1(v_b)$ and $r_1(v_c) \sim_\beta r'_1(v_c)$ because of $r_1 \sim_{\beta,\mathsf{rda}} r'_1$. As $r_1(v_b), r'_1(v_b) \in \mathcal{L}$, this implies that $\beta(r_1(v_b)) = r'_1(v_b)$. With $h_1 \sim_\beta h'_1$, we know by the definition of indistinguishability of heaps that $h_1(r_1(v_b)) \sim_\beta h'_1(\beta(r_1(v_b)))$. Hence, for the arrays $ar, ar' \in \mathcal{A}$ such that $ar = h_1(r_1(v_b))$ and $ar' = h'_1(r'_1(v_b))$, we have that $ar \sim_\beta ar'$. Moreover, by $r_1(v_c) \sim_\beta r_1'(v_c)$, we know that $r_1(v_c) = r_1'(v_c)$ since v_c must contain a number. By the definition of array indistinguishability, $ar \sim_\beta ar'$, $\mathsf{ada} = low$, and $r_1(v_c) = r_1'(v_c)$ follows that $ar[r_1(v_c)] \sim_\beta ar'[r_1'(v_c)]$. Hence, $r_2(v_a) \sim_\beta r_2'(v_a)$.

*Case 12 (**aput** v_a, v_b, v_c).*

$$\text{rAput}\frac{\begin{array}{ccc} m[pp] = \texttt{aput}\ v_a, v_b, v_c & r(v_b) \in dom(h) & ar = h(r(v_b)) \\ x = ar[r(v_c) \mapsto r(v_a)] & & 0 \leq r(v_c) < ar.\mathsf{length} \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[r(v_b) \mapsto x], pp+1, r\rangle}$$

$$\text{tAput}\frac{m[pp] = \texttt{aput}\ v_a, v_b, v_c \qquad \mathsf{rda}(v_a) \sqcup se(pp) \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqsubseteq \mathsf{ada}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}}$$

As no new objects or arrays are created, $\beta' = \beta$. The register states and register domain assignments are not changed, so we have $\mathsf{rda} = \mathsf{rda}'$ and $r_1 \sim_{\beta,\mathsf{rda}} r_1'$ implies $r_2 \sim_{\beta',\mathsf{rda}'} r_2'$.

It remains to show that $h_2 \sim_\beta h_2'$. In the following, let $ar_1 = h_1(r_1(v_b))$, $ar_1' = h_1'(r_1'(v_b))$, $ar_2 = ar_1[r_1(v_c) \mapsto r_1(v_a)]$, and $ar_2' = ar_1'[r_1'(v_c) \mapsto r_1'(v_a)]$. According to rule (rAput), the only change to the heap h_1 is that the array ar_1 at location $l = r_1(v_b)$ is updated at the index $r_1(v_c)$ to ar_2. Respectively, the heap h_1' is changed such that that the array ar_1' at location $l' = r_1'(v_b)$ is updated at the index $r_1'(v_c)$ to ar_2'. Moreover, by the definition of heap indistinguishability, two heaps can only be distinguished by the instances that are at locations related by β. Hence, to show that $h_2 \sim_\beta h_2'$ given $h_1 \sim_\beta h_1'$, it remains to show that $h_2(l) \sim_\beta h_2'(\beta(l))$ if $l \in dom(\beta)$, and $h_2(\beta^{-1}(l')) \sim_\beta h_2'(l')$ if $l' \in rng(\beta)$. We distinguish two cases:

$\mathsf{ada} = high$**.** Since $h_1 \sim_\beta h_1'$ and the content of arrays in general is not observable due to the assumption $\mathsf{ada} = high$, changes to the array content leave the resulting arrays indistinguishable, i.e., $h_2(l) \sim_\beta h_2'(\beta(l))$ if $l \in dom(\beta)$ and $h_2(\beta^{-1}(l')) \sim_\beta h_2'(l')$ if $l' \in rng(\beta)$ trivially follow from $\mathsf{ada} = high$, $h_1(l) \sim_\beta h_1'(\beta(l))$, $h_1(\beta^{-1}(l')) \sim_\beta h_1'(l')$, and the definition of indistinguishability of arrays.

$\mathsf{ada} = low$**.** Then $\mathsf{rda}(v_a) = \mathsf{rda}(v_b) = \mathsf{rda}(v_c) = low$ by the premise of (tAput). With $r_1 \sim_{\beta,\mathsf{rda}} r_1'$, it follows that $r_1(v_b) \sim_\beta r_1'(v_b)$ and, thus, $\beta(l) = l'$, respectively $\beta^{-1}(l') = l$. Hence, we have to show that $h_2(l) \sim_\beta h_2'(l')$ given that $h_1(l) \sim_\beta h_1'(l')$. This is equivalent to showing $ar_2 \sim_\beta ar_2'$ given that $ar_1 \sim_\beta ar_1'$. From $\mathsf{rda}(v_c) = low$ and $r_1 \sim_{\beta,\mathsf{rda}} r_1'$ we have that $r_1(v_c) \sim_\beta r_1'(v_c)$ and, since v_c must store an index number, $r_1(v_c) = r_1'(v_c)$. Hence there exists an $i \in \mathbb{N}_0$ such that $r_1(v_c) = r_1'(v_c) = i$, $ar_2 = ar_1[i \mapsto r_1(v_a)]$, and $ar_2' = ar_1'[i \mapsto r_1'(v_a)]$. To show $ar_2 \sim_\beta ar_2'$, it remains to show that $r_1(v_a) \sim_\beta r_1'(v_a)$, which is fulfilled because $r_1 \sim_{\beta,\mathsf{rda}} r_1'$ and $\mathsf{rda}(v_a) = low$ by assumption.

*Case 13 (**unop-wideS** v_a, v_b, uop).*

$$\text{rUnopWideS}\frac{m[pp] = \texttt{unop-wideS}\ v_a, v_b, uop \qquad u = \underline{uop}(r(v_b) \bullet r(v_{b+1}))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto u]\rangle}$$

$$\text{tUnopWS}\frac{m[pp] = \texttt{unop-wideS}\ v_a, v_b, uop \qquad t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_{b+1}) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

As the heap does not change, $\beta' = \beta$ and $h_1 \sim_\beta h'_1$ implies $h_2 \sim_{\beta'} h'_2$. We need to show that $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$. The only register that is updated in the register state and the register domain assignment is v_a. Hence, given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r_2(v_a) \sim_\beta r'_2(v_a)$ to have $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$.

Assume $\mathsf{rda}'(v_a) = low$, then $\mathsf{rda}(v_b) = \mathsf{rda}(v_{b+1}) = low$ because otherwise t in (tUnopWS) would be *high*. Since the operators represented by the symbols in $\mathcal{CONV}$ all operate on numbers (locations and void are always 32-bit values), v_b and v_{b+1} must store numbers. With $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we have $r_1(v_b) \sim_\beta r'_1(v_b)$ and $r_1(v_{b+1}) \sim_\beta r'_1(v_{b+1})$. Thus, $r_1(v_b) = r'_1(v_b)$ and $r_1(v_{b+1}) = r'_1(v_{b+1})$ by the definition of indistinguishability of values. With the definition of indistinguishability of composed values, we get

$$r_1(v_b) \bullet r_1(v_{b+1}) = r'_1(v_b) \bullet r'_1(v_{b+1}).$$

Since $\underline{uop}$ is a function, we have

$$\underline{uop}(r_1(v_b) \bullet r_1(v_{b+1})) = \underline{uop}(r'_1(v_b) \bullet r'_1(v_{b+1}))$$

and therefore $r_2(v_a) \sim_\beta r'_2(v_a)$ holds.

Case 14 (`unop-wideT` v_a, v_b, uop).

$$\text{rUnopWideT}\frac{m[pp] = \texttt{unop-wideT}\ v_a, v_b, uop \qquad u = \underline{uop}(r(v_b))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{lower}(u), v_{a+1} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{tUnopWT}\frac{m[pp] = \texttt{unop-wideT}\ v_a, v_b, uop \qquad t = \mathsf{rda}(v_b) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

As the heap does not change, $\beta' = \beta$ and $h_1 \sim_\beta h'_1$ implies $h_2 \sim_{\beta'} h'_2$. We need to show that $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$. The registers that are updated in the register state and the register domain assignment are v_a and v_{a+1}. Both are set to the same security domain. Hence, given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r_2(v_a) \sim_\beta r'_2(v_a)$ and $r_2(v_{a+1}) \sim_\beta r'_2(v_{a+1})$ to have $r_2 \sim_{\beta,\mathsf{rda}'} r'_2$.

Assume $\mathsf{rda}'(v_a) = low$, then $\mathsf{rda}(v_b) = low$ because otherwise t in the premise of (tUnopWT) would be *high*. Since the operators represented by the symbols in $\mathcal{CONV}$ all operate on numbers (locations and void are always 32-bit values), v_b must store numbers. With $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we have $r_1(v_b) \sim_\beta r'_1(v_b)$. Thus,

$r_1(v_b) = r_1'(v_b)$ by the definition of indistinguishability of values. Since $\underline{uop}$, upper, and lower are functions,

$$r_2(v_a) = \mathsf{lower}(\underline{uop}(r_1(v_b))) = \mathsf{lower}(\underline{uop}(r_1'(v_b))) = r_2'(v_a),$$

$$r_2(v_{a+1}) = \mathsf{upper}(\underline{uop}(r_1(v_b))) = \mathsf{upper}(\underline{uop}(r_1'(v_b))) = r_2'(v_{a+1}),$$

and therefore $r_2(v_a) \sim_\beta r_2'(v_a)$ and $r_2(v_{a+1}) \sim_\beta r_2'(v_{a+1})$ holds.

Lemma 2 (Locally respect for return). *For all methods $m \in \mathcal{M}_P$ of program P, register states $r_1, r_2 \in \mathcal{R}$, heaps $h_1, h_2, h_1', h_2' \in \mathcal{H}$, program points $pp_1 \in \mathbb{N}_0$, values $u_2, u_2' \in \mathcal{V}$, partial injective functions on locations $\beta \in \mathcal{B}$, register domain assignments $\mathsf{rda} \in \mathcal{RDA}$, security environments $se : \mathbb{N}_0 \to \mathcal{SL}$, and security domains $ret \in \mathcal{SL}$, if*

1. $se(pp_1) = low$,
2. $h_1 \sim_\beta h_1'$,
3. $r_1 \sim_{\beta,\mathsf{rda}} r_1'$,
4. $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}$,
5. $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(0)}_{P,m} \langle u_2, h_2 \rangle$, *and*
6. $\langle h_1', pp_1, r_1' \rangle \rightsquigarrow^{(0)}_{P,m} \langle u_2', h_2' \rangle$,

then there exists some $\beta' \in \mathcal{B}$ with $\beta \subseteq \beta'$ such that $h_2 \sim_{\beta'} h_2'$ and, if $ret = low$, $u_2 \sim_{\beta'} u_2'$.

Proof. Let $m \in \mathcal{M}_P$ be a method of program P, $r_1, r_2 \in \mathcal{R}$ be register states, $h_1, h_2, h_1', h_2' \in \mathcal{H}$ be heaps, $pp_1 \in \mathbb{N}_0$ be a program point, $u_2, u_2' \in \mathcal{V}$ be values, $\beta \in \mathcal{B}$ be a partial injective function on locations, $\mathsf{rda} \in \mathcal{RDA}$ be a register domain assignment, $se : \mathbb{N}_0 \to \mathcal{SL}$ be a security environment, and $ret \in \mathcal{SL}$ be a security domain such that $se(pp_1) = low$, $h_1 \sim_\beta h_1'$, $r_1 \sim_{\beta,\mathsf{rda}} r_1'$, $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}$, $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(0)}_{P,m} \langle u_2, h_2 \rangle$, and $\langle h_1', pp_1, r_1' \rangle \rightsquigarrow^{(0)}_{P,m} \langle u_2', h_2' \rangle$.

Since `return-void` can be seen as a special case of `return`, we show the proof for $m[pp_1] = \texttt{return}\ v_a$.

$$\text{rReturn}\frac{m[pp] = \texttt{return}\ v_a}{\langle h, pp, r \rangle \rightsquigarrow^{(0)}_{P,m} \langle r(v_a), h \rangle}$$

$$\text{tReturn}\frac{m[pp] = \texttt{return}\ v_a \qquad se(pp) \sqcup \mathsf{rda}(v_a) \sqsubseteq ret}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

As the heap does not change, $\beta' = \beta$ and $h_1 \sim_\beta h_1'$ implies $h_2 \sim_\beta h_2'$. We need to show that if $ret = low$, then $u_1 \sim_\beta u_2$.

Assume $ret = low$, then $\mathsf{rda}(v_a) = low$ holds due to the premise $\mathsf{rda}(v_a) \sqsubseteq ret$ of the typing rule (tReturn). With $r_1 \sim_\beta r_1'$, this implies $r_1(v_a) \sim_\beta r_1'(v_a)$. Since $u_1 = r_1(v_a)$ and $u_2 = r_2(v_a)$ by the semantics of `return`, we have $u_1 \sim_\beta u_2$.

The proof of locally respect for invoke requires that the called method already satisfies *TIN-ADL* in order to show that indistinguishability of register states and heaps is preserved. This is achieved by an additional precondition (i.e., Definition 30) that guarantees the security of all method invocations with a number of method calls greater than or equal to the number of method calls of the execution step given in the lemma. When using this lemma in the proof of the security of methods that is shown by induction over the number of occurring method calls, the required guarantee is provided by the induction hypothesis.

Definition 30 (Security of methods up to n calls). *Let n be a natural number. The methods of a typable program P are secure up to n calls if and only if for all methods $m \in \mathcal{M}_P$, security environments $se : \mathbb{N}_0 \to \mathcal{SL}$, register domain assignments $\mathsf{rda}_0, \ldots, \mathsf{rda}_k \in \mathcal{RDA}$ where $k = \mathsf{length}(m) - 1$, partial injective functions $\beta \in \mathcal{B}$, program points $pp_1, pp_2 \in \mathbb{N}_0$, register states $r_1, r_2 \in \mathcal{R}$, heaps $h_1, h_2, h_1', h_2' \in \mathcal{H}$, return values $u_1, u_2 \in \mathcal{V}$, and natural numbers $n_1, n_2 \in \mathbb{N}_0$ such that*

1. $n_1 < n, n_2 < n$
2. $pp_1 = pp_2$,
3. $r_1 \sim_{\beta, \mathsf{rda}_{pp_1}} r_2$,
4. $h_1 \sim_\beta h_2$,
5. $\langle h_1, pp_1, r_1 \rangle \Downarrow_{P,m}^{(n_1)} \langle u_1, h_1' \rangle$,
6. $\langle h_2, pp_2, r_2 \rangle \Downarrow_{P,m}^{(n_2)} \langle u_2, h_2' \rangle$,
7. *for all $i, j \in \mathbb{N}_0$, if $i \to_m j$ there exists a register domain assignment $\mathsf{rda}_j' \in \mathcal{RDA}$ such that the judgment $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}_j'$ is derivable and $\mathsf{rda}_j' \sqsubseteq \mathsf{rda}_j$, and*
8. *for all $i \in \mathbb{N}_0$, if there exists no $j \in \mathbb{N}_0$ such that $i \to_m j$, then the judgment $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}_i$ is derivable.*

there exists a partial injective function on locations $\beta' \in \mathcal{B}$, such that $\beta \subseteq \beta'$, $h_1' \sim_{\beta'} h_2'$ and, if $ret = low$, $u_1 \sim_{\beta'} u_2$.

Lemma 3 (Locally respect for invoke). *For all methods $m \in \mathcal{M}_P$ of a typable program P, register states $r_1, r_2, r_1', r_2' \in \mathcal{R}$, heaps $h_1, h_2, h_1', h_2' \in \mathcal{H}$, program points $pp_1, pp_2, pp_2' \in \mathbb{N}_0$, natural numbers $n_0, n_1, n_2 \in \mathbb{N}_0$, partial injective functions on locations $\beta \in \mathcal{B}$, register domain assignments $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$, security environments $se : \mathbb{N}_0 \to \mathcal{SL}$, and security domains $ret \in \mathcal{SL}$, if*

1. *the methods of P are secure up to n_0 calls,*
2. $n_1 \le n_0$, $n_2 \le n_0$,
3. $se(pp_1) = low$,
4. $h_1 \sim_\beta h_1'$,
5. $r_1 \sim_{\beta, \mathsf{rda}} r_1'$,
6. $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$,
7. $\langle h_1, pp_1, r_1 \rangle \overset{(n_1+1)}{\rightsquigarrow}_{P,m} \langle h_2, pp_2, r_2 \rangle$, *and*
8. $\langle h_1', pp_1, r_1' \rangle \overset{(n_2+1)}{\rightsquigarrow}_{P,m} \langle h_2', pp_2', r_2' \rangle$,

then there exists some $\beta' \in \mathcal{B}$ *with* $\beta \subseteq \beta'$ *such that* $h_2 \sim_{\beta'} h_2'$ *and* $r_2 \sim_{\beta',\mathsf{rda}'} r_2'$.

Proof. Let $m \in \mathcal{M}_P$ be a method of a typable program P, $r_1, r_2, r_1', r_2' \in \mathcal{R}$ be register states, $h_1, h_2, h_1', h_2' \in \mathcal{H}$ be heaps, $pp_1, pp_2, pp_2' \in \mathbb{N}_0$ be program points, $n_0, n_1, n_2 \in \mathbb{N}_0$ be natural numbers, $\beta \in \mathcal{B}$ be a partial injective function on locations, $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$ be register domain assignments, $se : \mathbb{N}_0 \to \mathcal{SL}$ be a security environment, and $ret \in \mathcal{SL}$ be a security domain such that the methods of P are secure up to n_0, $n_1 \leq n_0$, $n_2 \leq n_0$, $se(pp_1) = low$, $h_1 \sim_\beta h_1'$, $r_1 \sim_{\beta,\mathsf{rda}} r_1'$, $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$, $\langle h_1, pp_1, r_1\rangle \overset{(n_1+1)}{\rightsquigarrow}_{P,m} \langle h_2, pp_2, r_2\rangle$, and $\langle h_1', pp_1, r_1'\rangle \overset{(n_2+1)}{\rightsquigarrow}_{P,m} \langle h_2', pp_2', r_2'\rangle$.

We show the case for $m[pp_1] = \texttt{invoke-virtual-range}\ v_k, n, mid$. The cases for other invoke instructions are analogous.

$$\text{rIVR}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-virtual-range}\ v_k, n, mid \qquad r(v_k) \in dom(h) \\ (mid, h(r(v_k)).\mathsf{class}) \in dom(\mathsf{lookup\text{-}virtual}_P) \\ m' = \mathsf{lookup\text{-}virtual}_P(mid, h(r(v_k)).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_k), \ldots r(v_{k+n-1})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h'\rangle \end{array}}{\langle h, pp, r\rangle \overset{(n'+1)}{\rightsquigarrow}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{tIR}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-virtual-range}\ v_k, n, mid \\ (mid, [\mathsf{rda}(v_k), \ldots, \mathsf{rda}(v_{k+n-1})], st) \in \mathsf{mda} \\ se(pp) = low \qquad\qquad \mathsf{rda}(v_k) = low \end{array}}{m, \cdots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto st, result_{upper} \mapsto st]}$$

From the premise of the typing rule (tIR), we know $\mathsf{rda}(v_k) = low$ and, thus, $r_1(v_k) \sim_\beta r_1'(v_k)$ by $r_1 \sim_{\beta,\mathsf{rda}} r_1'$ and the definition of register indistinguishability. By definition of object indistinguishability, this implies that $h_1(r_1(v_k)).\mathsf{class} = h_1'(r_1'(v_k)).\mathsf{class}$. Since mid is hard-coded in the instruction and $\mathsf{lookup\text{-}virtual}_P$ is a function, this implies that

$$\begin{aligned} m' &= \mathsf{lookup\text{-}virtual}_P(mid, h_1(r_1(v_k)).\mathsf{class}) \\ &= \mathsf{lookup\text{-}virtual}_P(mid, h_1'(r_1'(v_k)).\mathsf{class}). \end{aligned}$$

From the semantics rule (rIVR), we know that there exist $u, u' \in \mathcal{V}$ such that

$$\begin{aligned} &\langle h_1, 0, \mathsf{defaultRegisters}([r_1(v_k), \ldots, r_1(v_{k+n-1})])\rangle \Downarrow^{(n_1)}_{P,m'} \langle u, h_2\rangle, \text{ and} \qquad (1) \\ &\langle h_1', 0, \mathsf{defaultRegisters}([r_1'(v_k), \ldots, r_1'(v_{k+n-1})])\rangle \Downarrow^{(n_2)}_{P,m'} \langle u', h_2'\rangle. \end{aligned}$$

From the premises of the typing rule (tIR), we know that there exists the method signature $(mid, [\mathsf{rda}(v_k), \ldots, \mathsf{rda}(v_{k+n-1})], st) \in \mathsf{mda}$. Let $\mathsf{rda}_0 \in \mathcal{RDA}$ be a register domain assignment corresponding to that signature, i.e., for all $i \in \mathbb{N}_0$ with $i < n$ it holds that $\mathsf{rda}_0(v_i) = \mathsf{rda}(v_{k+i})$. Since $r_1 \sim_{\beta,\mathsf{rda}} r_1'$, for each register $v \in \{v_k, \ldots, v_{k+n-1}\}$ either $\mathsf{rda}_0(v) = high$ or $r_1(v) \sim_\beta r_1'(v)$ holds by the definition of register indistinguishability. As the remaining registers in the method arguments are mapped to void by $\mathsf{defaultRegisters}$, and $\mathsf{void} \sim_\beta \mathsf{void}$, we know that

$$\mathsf{defaultRegisters}([r_1(v_k), \ldots, r_1(v_{k+n-1})]) \sim_{\beta,\mathsf{rda}_0} \mathsf{defaultRegisters}([r'_1(v_k), \ldots, r'_1(v_{k+n-1})]). \tag{2}$$

Since the methods of P are secure up to n_0 calls, given the assumptions of this lemma 2., 4., (1), (2), and the typability of program P, we know that there exists some $\beta' \in \mathcal{B}$ with $\beta \subseteq \beta'$ such that $h_2 \sim_{\beta'} h'_2$ and, if $st = low$, $u \sim_{\beta'} u'$.

We still need to show that $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$. The registers that are updated in the register state and the register domain assignment are $result_{lower}$ and $result_{upper}$. Both are set to the same security domain st. Hence, given that $r_1 \sim_{\beta,\mathsf{rda}} r'_1$, we only need to show if $st = low$ that $r_2(result_{lower}) \sim_{\beta'} r'_2(result_{lower})$ and $r_2(result_{upper}) \sim_{\beta'} r'_2(result_{upper})$ to have $r_2 \sim_{\beta',\mathsf{rda}'} r'_2$.

Assume $st = low$, then we have shown that $u \sim_{\beta'} u'$. With the definition of the indistinguishability of concatenated values follows that

$$\mathsf{lower}(u) \sim_{\beta'} \mathsf{lower}(u'), \text{ and } \mathsf{upper}(u) \sim_{\beta'} \mathsf{upper}(u').$$

Moreover, it holds that $r_2(result_{lower}) = \mathsf{lower}(u)$, $r_2(result_{upper}) = \mathsf{upper}(u)$, $r'_2(result_{upper}) = \mathsf{upper}(u')$, and $r'_2(result_{lower}) = \mathsf{lower}(u')$ by the semantics of `invoke-virtual-range`. Therefore, we have $r_2(result_{lower}) \sim_{\beta'} r'_2(result_{lower})$ and $r_2(result_{upper}) \sim_{\beta'} r'_2(result_{upper})$.

Lemma 4 (Step consistent). *For all methods $m \in \mathcal{M}_P$ of a typable program P, natural numbers $n \in \mathbb{N}_0$, register states $r, r_1, r_2 \in \mathcal{R}$, heaps $h, h_1, h_2 \in \mathcal{H}$, program points $pp_1, pp_2 \in \mathbb{N}_0$, partial injective functions on locations $\beta \in \mathcal{B}$, register domain assignments $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$, security environments $se : \mathbb{N}_0 \to \mathcal{SL}$, and security domains $ret \in \mathcal{SL}$ such that*

1. *$se(pp_1) = high$,*
2. *$h \sim_\beta h_1$,*
3. *$r \sim_{\beta,\mathsf{rda}} r_1$,*
4. *$m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$, and*
5. *$\langle h_1, pp_1, r_1\rangle \leadsto^{(n)}_{P,m} \langle h_2, pp_2, r_2\rangle$,*

then $h \sim_\beta h_2$, and $r \sim_{\beta,\mathsf{rda}'} r_2$.

Proof. Let $m \in \mathcal{M}_P$ be a method of a typable program P, $r, r_1, r_2 \in \mathcal{R}$ be register states, $h, h_1, h_2 \in \mathcal{H}$ be heaps, $pp_1, pp_2 \in \mathbb{N}_0$ be program points, $\beta \in \mathcal{B}$ be a partial injective function on locations, $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$ be register domain assignments, $se : \mathbb{N}_0 \to \mathcal{SL}$ be a security environment, $ret \in \mathcal{SL}$ be a security domain, and $n \in \mathbb{N}_0$ be a natural number such that $se(pp_1) = high$, $h \sim_\beta h_1$, $r \sim_{\beta,\mathsf{rda}} r_1$, $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$, and $\langle h_1, pp_1, r_1\rangle \leadsto^{(n)}_{P,m} \langle h_2, pp_2, r_2\rangle$. To show that $h \sim_\beta h_2$ and $r \sim_{\beta,\mathsf{rda}'} r_2$, we distinguish cases over the the different instructions.

Case 1 (`binop` v_a, v_b, v_c, bop, `const-string` v_a, s, `iget` v_a, v_b, fid, `sget` v_a, fid, `aget` v_a, v_b, v_c, `unop-wideS` v_a, v_b, uop).

$$\text{rBinop}\frac{m[pp] = \texttt{binop}\ v_a, v_b, v_c, bop \qquad x = r(v_b)\ \underline{bop}\ r(v_c)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto x]\rangle}$$

$$\text{tBinop}\frac{m[pp] = \texttt{binop}\ v_a, v_b, v_c, bop \qquad t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

The semantics of all these instructions has in common that

- it does not alter the heap, i.e., $h_2 = h_1$, and
- it updates the register state only in the register v_a, i.e., $r_2 = r_1[v_a \mapsto u]$ for some value $u \in \mathcal{V}$.

Moreover, the typing rules of these instructions all set $\mathsf{rda}' = \mathsf{rda}[v_a \mapsto s]$ where $s = se(pp_1) \sqcup s_0 \sqcup \ldots$ for some $s_0, \dots \in \mathcal{SL}$.

From $h \sim_\beta h_1$ and $h_2 = h_1$ follows immediately that $h \sim_\beta h_2$.

It remains to show that $r \sim_{\beta,\mathsf{rda}'} r_2$. Given that $r \sim_{\beta,\mathsf{rda}} r_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r(v_a) \sim_\beta r_2(v_a)$ to have $r \sim_{\beta,\mathsf{rda}'} r_2$. Since $\mathsf{rda}'(v_a) = se(pp_1) \sqcup s_0 \sqcup \ldots$ for some $s_0, \dots \in \mathcal{SL}$ and $se(pp_1) = high$ by assumption, we have $\mathsf{rda}'(v_a) = high$. Thus, we can conclude $r \sim_{\beta,\mathsf{rda}'} r_2$.

Case 2 (`if-test` v_a, v_b, n, rop).

$$\text{rIfTestTrue}\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad r(v_a)\ \underline{rop}\ r(v_b)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+n, r\rangle}$$

$$\text{rIfTestFalse}\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad \neg(r(v_a)\ \underline{rop}\ r(v_b))}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r\rangle}$$

$$\text{tIfTest}\frac{\begin{array}{c} m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \\ \forall j \in region_m(pp).\mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_b) \sqsubseteq se(j)\end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

From the rules (rIfTestTrue), (rIfTestFalse), and (tIfTest), we know that $h_2 = h_1$, $r_2 = r_1$, and $\mathsf{rda}' = \mathsf{rda}$. Thus, $h \sim_\beta h_2$ and $r \sim_{\beta,\mathsf{rda}'} r_2$ follows immediately from $h \sim_\beta h_1$ and $r \sim_{\beta,\mathsf{rda}} r_1$.

Case 3 (`new-instance` v_a, cl).

$$\text{rNewInstance}\frac{\begin{array}{c} m[pp] = \texttt{new-instance}\ v_a, cl \qquad h \in dom(\mathsf{nextFreeLocation}) \\ l = \mathsf{nextFreeLocation}(h)\end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[l \mapsto \mathsf{defaultObject}(cl)], pp+1, r[v_a \mapsto l]\rangle}$$

$$\text{tNewInstance}\frac{m[pp] = \texttt{new-instance}\ v_a, cl}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto se(pp)]}$$

We first show $h \sim_\beta h_2$. Let $l = \mathsf{nextFreeLocation}(h_1)$. From $h \sim_\beta h_1$, we know that $dom(\beta) \subseteq dom(h)$ and $rng(\beta) \subseteq dom(h_1)$. Moreover, by rule (rNewInstance), we know that $h_2 = h_1[l \mapsto \mathsf{defaultObject}(cl)]$ and, thus, $dom(h_2) = dom(h_1) \cup \{l\}$. Ultimately, we can conclude $dom(\beta) \subseteq dom(h)$ and $rng(\beta) \subseteq dom(h_2)$, which satisfies the first two requirements of the indistinguishability of heaps (Definition 22). It remains to show that for all locations $l \in dom(\beta)$ either $l \in dom_{\mathcal{A}}(h)$, $\beta(l) \in dom_{\mathcal{A}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$ or $l \in dom_{\mathcal{O}}(h)$, $\beta(l) \in dom_{\mathcal{O}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$.

Since $h_2 = h_1[l \mapsto \mathsf{defaultObject}(cl)]$, we know that h_2 differs from h_1 only in location l. As $\mathsf{nextFreeLocation}$ allocates fresh locations on the heap provided as argument, we know that $l \notin dom(h_1)$. Hence, for all locations $l \in dom(h_1)$ it holds that $h_1(l) = h_2(l)$. With $h \sim_\beta h_1$, we have for all locations $l \in dom(\beta)$ either $l \in dom_{\mathcal{A}}(h)$, $\beta(l) \in dom_{\mathcal{A}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$ or $l \in dom_{\mathcal{O}}(h)$, $\beta(l) \in dom_{\mathcal{O}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$.

We still need to show that $r \sim_{\beta,\mathsf{rda}'} r_2$. According to the rules (rNewInstance) and (tNewInstance), the only register that is updated in the register state and the register domain assignment is v_a. Hence, given that $r \sim_{\beta,\mathsf{rda}} r_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r(v_a) \sim_\beta r_2(v_a)$ to have $r \sim_{\beta,\mathsf{rda}'} r_2$. From rule (tNewInstance) follows that $\mathsf{rda}'(v_a) = se(pp_1)$. Since $se(pp_1) = high$, we have $\mathsf{rda}'(v_a) = high$. Thus, we have $r \sim_{\beta,\mathsf{rda}'} r_2$.

*Case 4 (**iput** v_a, v_b, fid, **sput** v_a, fid).*

$$\text{rIput}\ \frac{\begin{array}{cc} m[pp] = \texttt{iput}\ v_a, v_b, fid & fid \in dom(\mathsf{lookup\text{-}field}_P) \\ r(v_b) \in dom(h) & o = h(r(v_b)) \\ f = \mathsf{lookup\text{-}field}_P(fid) & f \in dom(o.\mathsf{fields}) \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[r(v_b) \mapsto o[f \mapsto r(v_a)]], pp+1, r\rangle}$$

$$\text{tIput}\ \frac{\begin{array}{c} m[pp] = \texttt{iput}\ v_a, v_b, fid \qquad \mathsf{fda}(fid) = st \\ \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_b) \sqcup se(pp) \sqsubseteq st \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

The semantics of both instructions has in common that

- it does not change the register state, i.e., $r_2 = r_1$, and
- it updates the heap at one location $l \in \mathcal{L}$ by changing the value of the field $f = \mathsf{lookup\text{-}field}_P(fid)$ of the object $o = h_1(l)$, i.e., $h_2 = h_1[l \mapsto o[f \mapsto u]]$ for some value $u \in \mathcal{V}$.

The typing rules of both instructions require that

- $se(pp_1) \sqcup s_0 \sqcup \ldots \sqsubseteq \mathsf{fda}(fid)$ for some $s_0, \dots \in \mathcal{SL}$, and
- the register domain assignment does not change, i.e., $\mathsf{rda}' = \mathsf{rda}$.

From $r \sim_{\beta,\mathsf{rda}} r_1$, $r_2 = r_1$, and $\mathsf{rda}' = \mathsf{rda}$ follows immediately that $r \sim_{\beta,\mathsf{rda}'} r_2$.

It remains to show that $h \sim_\beta h_2$. Let $l \in \mathcal{L}$, $f \in \mathcal{F}$, $o \in \mathcal{O}$, and $u \in \mathcal{V}$ such that $f = \mathsf{lookup\text{-}field}_P(fid)$, $o = h_1(l)$, and $h_2 = h_1[l \mapsto o[f \mapsto u]]$. Hence, $h_1 = h_2$ except for the object at location l. Moreover, by the definition of heap indistinguishability, two heaps can only be distinguished by the instances that are at locations related by β. Hence, to show that $h \sim_\beta h_2$ given $h \sim_\beta h_1$, it remains to show that $h(\beta^{-1}(l)) \sim_\beta h_2(l)$ if $l \in rng(\beta)$.

Assume $l \in rng(\beta)$. Given that $se(pp_1) \sqcup s_0 \sqcup \ldots \sqsubseteq \mathsf{fda}(fid)$ for some $s_0, \dots \in \mathcal{SL}$, and $se(pp_1) = high$ by assumption, we have $\mathsf{fda}(fid) = high$. Since the class of the modified object is not changed and objects can only be distinguished by public fields, changes to the private field fid leave the resulting object $o[f \mapsto u]$ at l in h_2 indistinguishable from o at l in h_1. Moreover, from the typability of the program also follows that there are no other field names $fid' \in \mathcal{FID}$ such that $\mathsf{lookup\text{-}field}(fid') = f$ and $\mathsf{fda}(fid') = low$. That means $h_1(l) \sim_\beta h_2(l)$ follows from $\mathsf{fda}(fid) = high$ and $h_2(l) = h_1(l)[f \mapsto u]$. Moreover, from $h \sim_\beta h_1$, we know that $h(\beta^{-1}(l)) \sim_\beta h_1(l)$. Finally, $h(\beta^{-1}(l)) \sim_\beta h_2(l)$ follows from $h(\beta^{-1}(l)) \sim_\beta h_1(l)$, $h_1(l) \sim_\beta h_2(l)$, and the transitivity of indistinguishability of objects.

Case 5 (`new-array` v_a, v_b).

$$\text{rNewArray}\,\frac{m[pp] = \texttt{new-array}\ v_a, v_b \qquad h \in dom(\mathsf{nextFreeLocation}) \qquad l = \mathsf{nextFreeLocation}(h) \qquad 0 \leq r(v_b)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[l \mapsto \mathsf{defaultArray}(r(v_b))], pp+1, r[v_a \mapsto l]\rangle}$$

$$\text{tNewA}\,\frac{m[pp] = \texttt{new-array}\ v_a, v_b}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto \mathsf{rda}(v_b) \sqcup se(pp)]}$$

We first show $h \sim_\beta h_2$. Let $l = \mathsf{nextFreeLocation}(h_1)$. From $h \sim_\beta h_1$, we know that $dom(\beta) \subseteq dom(h)$ and $rng(\beta) \subseteq dom(h_1)$. Moreover, by rule (rNewArray), we know that $h_2 = h_1[l \mapsto \mathsf{defaultArray}(n)]$ and, thus, $dom(h_2) = dom(h_1) \cup \{l\}$. Ultimately, we can conclude $dom(\beta) \subseteq dom(h)$ and $rng(\beta) \subseteq dom(h_2)$, which satisfies the first two requirements of the indistinguishability of heaps (Definition 22). It remains to show that for all locations $l \in dom(\beta)$ either $l \in dom_{\mathcal{A}}(h)$, $\beta(l) \in dom_{\mathcal{A}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$ or $l \in dom_{\mathcal{O}}(h)$, $\beta(l) \in dom_{\mathcal{O}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$.

Since $h_2 = h_1[l \mapsto \mathsf{defaultArray}(n)]$, h_2 differs from h_1 only in location l. As $\mathsf{nextFreeLocation}$ allocates fresh locations on the heap provided as argument, we know that $l \notin dom(h_1)$. Hence, for all locations $l \in dom(h_1)$ it holds that $h_1(l) = h_2(l)$. With $h \sim_\beta h_1$, we have for all locations $l \in dom(\beta)$ either $l \in dom_{\mathcal{A}}(h)$, $\beta(l) \in dom_{\mathcal{A}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$ or $l \in dom_{\mathcal{O}}(h)$, $\beta(l) \in dom_{\mathcal{O}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$.

We still need to show that $r \sim_{\beta,\mathsf{rda}'} r_2$. According to the rules (rNewArray) and (tNewA), the only register that is updated in the register state and the register domain assignment is v_a. Hence, given that $r \sim_{\beta,\mathsf{rda}} r_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r(v_a) \sim_\beta r_2(v_a)$ to have $r \sim_{\beta,\mathsf{rda}'} r_2$. From rule (tNewA) follows that $\mathsf{rda}'(v_a) = \mathsf{rda}(v_b) \sqcup se(pp_1)$. Since $se(pp_1) = high$, we have $\mathsf{rda}'(v_a) = high$. Thus, we have $r \sim_{\beta,\mathsf{rda}'} r_2$.

Case 6 (`filled-new-array-range` v_k, n).

$$\text{rFilledNewArrayR}\ \frac{\begin{array}{c} m[pp] = \texttt{filled-new-array-range}\ v_k, n \\ h \in dom(\mathsf{nextFreeLocation}) \\ l = \mathsf{nextFreeLocation}(h) \qquad x = \mathsf{defaultArray}(n) \\ ar = x[0 \mapsto r(v_k), \dots, n-1 \mapsto r(v_{k+n-1})] \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[l \mapsto ar], pp+1, r[result_{lower} \mapsto l]\rangle}$$

$$\text{tFNAR}\ \frac{m[pp] = \texttt{filled-new-array-range}\ v_k, n \qquad \bigsqcup_{i=k}^{k+n-1} \mathsf{rda}(v_i) \sqsubseteq \mathsf{ada}}{m, \dots \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}[result_{lower} \mapsto se(pp), result_{upper} \mapsto se(pp)]}$$

We first show $h \sim_\beta h_2$. Let $l = \mathsf{nextFreeLocation}(h_1)$. From $h \sim_\beta h_1$, we know that $dom(\beta) \subseteq dom(h)$ and $rng(\beta) \subseteq dom(h_1)$. Moreover, by rule (rFilledNewArrayR), we know that $dom(h_2) = dom(h_1) \cup \{l\}$. Ultimately, we can conclude $dom(\beta) \subseteq dom(h)$ and $rng(\beta) \subseteq dom(h_2)$, which satisfies the first two requirements of the indistinguishability of heaps (Definition 22). It remains to show that for all locations $l \in dom(\beta)$ either $l \in dom_{\mathcal{A}}(h)$, $\beta(l) \in dom_{\mathcal{A}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$ or $l \in dom_{\mathcal{O}}(h)$, $\beta(l) \in dom_{\mathcal{O}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$.

As of rule (rFilledNewArrayR), $h_2 = h_1[l \mapsto \mathsf{defaultArray}(n)[\dots]]$, i.e., h_2 differs from h_1 only in location l. Since $\mathsf{nextFreeLocation}$ allocates fresh locations on the heap provided as argument, we know that $l \notin dom(h_1)$. Hence, for all locations $l \in dom(h_1)$ it holds that $h_1(l) = h_2(l)$. With $h \sim_\beta h_1$, we have for all locations $l \in dom(\beta)$ either $l \in dom_{\mathcal{A}}(h)$, $\beta(l) \in dom_{\mathcal{A}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$ or $l \in dom_{\mathcal{O}}(h)$, $\beta(l) \in dom_{\mathcal{O}}(h_2)$, and $h(l) \sim_\beta h_2(\beta(l))$.

We still need to show that $r \sim_{\beta,\mathsf{rda}'} r_2$. According to the rules (rFilledNewArrayR) and (tFNAR), the only registers that are updated in the register state and the register domain assignment are $result_{lower}$ and $result_{upper}$. Given that $r \sim_{\beta,\mathsf{rda}} r_1$, we only need to show if $\mathsf{rda}'(result_{lower}) = low$, that $r(result_{lower}) \sim_\beta r_2(result_{lower})$ holds and if $\mathsf{rda}'(result_{upper}) = low$, that $r(result_{upper}) \sim_\beta r_2(result_{upper})$. Since $\mathsf{rda}'(result_{lower}) = \mathsf{rda}'(result_{upper}) = se(pp_1)$ according to rule (tFNAR) and $se(pp_1) = high$ by assumption, we have $\mathsf{rda}'(result_{lower}) = \mathsf{rda}'(result_{upper}) = high$. Thus, we can conclude $r \sim_{\beta,\mathsf{rda}'} r_2$.

*Case 7 (**aput** v_a, v_b, v_c).*

$$\text{rAput}\ \frac{\begin{array}{c} m[pp] = \texttt{aput}\ v_a, v_b, v_c \qquad r(v_b) \in dom(h) \qquad ar = h(r(v_b)) \\ x = ar[r(v_c) \mapsto r(v_a)] \qquad\qquad 0 \le r(v_c) < ar.\mathsf{length} \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[r(v_b) \mapsto x], pp+1, r\rangle}$$

$$\text{tAput}\ \frac{m[pp] = \texttt{aput}\ v_a, v_b, v_c \qquad \mathsf{rda}(v_a) \sqcup se(pp) \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqsubseteq \mathsf{ada}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \rightarrow \mathsf{rda}}$$

By the rules (rAput) and (tAput), we have $r_2 = r_1$, and $\mathsf{rda}' = \mathsf{rda}$. With $r \sim_{\beta,\mathsf{rda}} r_1$, it follows immediately that $r \sim_{\beta,\mathsf{rda}'} r_2$.

It remains to show that $h \sim_\beta h_2$. Let $l = r_1(v_r)$. By rule (rAput), we know that $h_2 = h_1[l \mapsto h_1(l)[v_c \mapsto v_a]]$. Hence, $h_1 = h_2$ except for the array at location

l. Moreover, by the definition of heap indistinguishability, two heaps can only be distinguished by the instances that are at locations related by β. Hence, to show that $h \sim_\beta h_2$ given $h \sim_\beta h_1$, it remains to show that $h(\beta^{-1}(l)) \sim_\beta h_2(l)$ if $l \in rng(\beta)$.

Assume $l \in rng(\beta)$. By rule (tAput), we know that $\mathsf{rda}(v_a) \sqcup se(pp_1) \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqsubseteq \mathsf{ada}$. With the assumption $se(pp_1) = high$, we get $\mathsf{ada} = high$. Since arrays can only be distinguished by their content if it is public, we get $h_1(l) \sim_\beta h_2(l)$ from $\mathsf{ada} = high$, $h_2(l) = h_1(l)[v_c \mapsto v_a]$, and $h_1(l).\mathsf{length} = h_1(l)[v_c \mapsto v_a].\mathsf{length}$. Moreover, from $h \sim_\beta h_1$, we know that $h(\beta^{-1}(l)) \sim_\beta h_1(l)$. Finally, $h(\beta^{-1}(l)) \sim_\beta h_2(l)$ follows from $h(\beta^{-1}(l)) \sim_\beta h_1(l)$, $h_1(l) \sim_\beta h_2(l)$, and the transitivity of indistinguishability of arrays.

Case 8 (`unop-wideT` v_a, v_b, uop).

$$\text{rUnopWideT}\ \frac{m[pp] = \texttt{unop-wideT}\ \ v_a, v_b, uop \qquad u = \underline{uop}(r(v_b))}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{lower}(u), v_{a+1} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{tUnopWT}\ \frac{m[pp] = \texttt{unop-wideT}\ \ v_a, v_b, uop \qquad t = \mathsf{rda}(v_b) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

By the rule (rUnopWideT), we know that $h_2 = h_1$. With $h \sim_\beta h_1$, it follows immediately that $h \sim_\beta h_2$.

It remains to show that $r \sim_{\beta,\mathsf{rda}'} r_2$. Given that $r \sim_{\beta,\mathsf{rda}} r_1$, we only need to show if $\mathsf{rda}'(v_a) = low$ that $r(v_a) \sim_\beta r_2(v_a)$ and if $\mathsf{rda}'(v_{a+1}) = low$ that $r(v_{a+1}) \sim_\beta r_2(v_{a+1})$. Since $\mathsf{rda}'(v_a) = \mathsf{rda}'(v_{a+1}) = \mathsf{rda}(v_b) \sqcup se(pp_1)$ according to rule (tUnopWT) and $se(pp_1) = high$ by assumption, we have $\mathsf{rda}'(v_a) = \mathsf{rda}'(v_{a+1}) = high$. Thus, we can conclude $r \sim_{\beta,\mathsf{rda}'} r_2$.

Case 9 (`invoke-virtual-range` v_k, n, mid).

$$\text{rIVR}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-virtual-range}\ \ v_k, n, mid \qquad r(v_k) \in dom(h) \\ (mid, h(r(v_k)).\mathsf{class}) \in dom(\mathsf{lookup\text{-}virtual}_P) \\ m' = \mathsf{lookup\text{-}virtual}_P(mid, h(r(v_k)).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_k), \dots r(v_{k+n-1})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h'\rangle \end{array}}{\langle h, pp, r\rangle \leadsto^{(n'+1)}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{tIR}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-virtual-range}\ \ v_k, n, mid \\ (mid, [\mathsf{rda}(v_k), \dots, \mathsf{rda}(v_{k+n-1})], st) \in \mathsf{mda} \\ se(pp) = low \qquad \mathsf{rda}(v_k) = low \end{array}}{m, \dots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto st, result_{upper} \mapsto st]}$$

Since we assume $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$, rule (tIR) requires that $se(pp_1) = low$. However, as we also assume $se(pp_1) = high$, we have a contradiction and the instruction at program point pp cannot be `invoke-virtual-range`.

Lemma 5 (Step consistent for return). *For all methods $m \in \mathcal{M}_P$ of program P, register states $r, r_1 \in \mathcal{R}$, heaps $h, h_1, h_2 \in \mathcal{H}$, values $u \in \mathcal{V}$, program points $pp_1 \in \mathbb{N}_0$, partial injective functions on locations $\beta \in \mathcal{B}$, register domain assignments $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$, security environments $se : \mathbb{N}_0 \to \mathcal{SL}$, and security domains $ret \in \mathcal{SL}$ such that*

1. $se(pp_1) = high$,
2. $h \sim_\beta h_1$,
3. $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$, *and*
4. $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(0)}_{P,m} \langle u, h_2 \rangle$,

it holds that $h \sim_\beta h_2$ and, if $u \neq \mathsf{void}$, $ret = high$.

Proof. Let $m \in \mathcal{M}_P$ be a method of program P, $r, r_1 \in \mathcal{R}$ be register states, $h, h_1, h_2 \in \mathcal{H}$ be heaps, $pp_1 \in \mathbb{N}_0$ be a program point, $\beta \in \mathcal{B}$ be a partial injective function on locations, $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$ be register domain assignments, $se : \mathbb{N}_0 \to \mathcal{SL}$ be a security environment, $ret \in \mathcal{SL}$ be a security domain, and $u \in \mathcal{V}$ be a value such that $se(pp_1) = high$, $h \sim_\beta h_1$, $r \sim_{\beta,\mathsf{rda}} r_1$, $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$, and $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(n)}_{P,m} \langle u, h_2 \rangle$.

Since `return-void` can be seen as a special case of `return`, we show that $h \sim_\beta h_2$ and, if $u \neq \mathsf{void}$, $ret = high$, by proving the case of $m[pp_1] = \texttt{return}\ v_a$.

$$\text{rReturn}\frac{m[pp] = \texttt{return}\ v_a}{\langle h, pp, r \rangle \rightsquigarrow^{(0)}_{P,m} \langle r(v_a), h \rangle}$$

$$\text{tReturn}\frac{m[pp] = \texttt{return}\ v_a \qquad se(pp) \sqcup \mathsf{rda}(v_a) \sqsubseteq ret}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

By $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(n)}_{P,m} \langle u, h_2 \rangle$ and rule (rReturn), we have $h_2 = h_1$. Hence, with $h \sim_\beta h_1$ follows $h \sim_\beta h_2$. By $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$ and $se(pp_1) = high$, we can conclude that $ret = high$ because of the premise $se(pp_1) \sqcup \mathsf{rda}(v_a) \sqsubseteq ret$ of rule (tReturn).

Lemma 6 (High branching). *For all methods $m \in \mathcal{M}_P$ of program P, register states $r_1, r_2, r'_1, r'_2 \in \mathcal{R}$, heaps $h_1, h_2, h'_1, h'_2 \in \mathcal{H}$, program points $pp_1, pp_2, pp'_2 \in \mathbb{N}_0$, partial injective functions on locations $\beta \in \mathcal{B}$, register domain assignments $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$, security environments $se : \mathbb{N}_0 \to \mathcal{SL}$, and security domains $ret \in \mathcal{SL}$, if*

1. $h_1 \sim_\beta h'_1$,
2. $r_1 \sim_{\beta,\mathsf{rda}} r'_1$,
3. $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$,
4. $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(0)}_{P,m} \langle h_2, pp_2, r_2 \rangle$,
5. $\langle h'_1, pp_1, r'_1 \rangle \rightsquigarrow^{(0)}_{P,m} \langle h'_2, pp'_2, r'_2 \rangle$, *and*
6. $pp_2 \neq pp'_2$,

then $se(pp') = high$ for all $pp' \in region_m(pp_1)$.

Proof. Let $m \in \mathcal{M}_P$ be a method of program P, $r_1, r_2, r_1', r_2' \in \mathcal{R}$ be register states, $h_1, h_2, h_1', h_2' \in \mathcal{H}$ be heaps, $pp_1, pp_2, pp_2' \in \mathbb{N}_0$ be program points, $\beta \in \mathcal{B}$ be a partial injective function on locations, $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$ be register domain assignments, $se : \mathbb{N}_0 \to \mathcal{SL}$ be security environments, and $ret \in \mathcal{SL}$ be a security domain, such that $h_1 \sim_\beta h_1'$, $r_1 \sim_{\beta,\mathsf{rda}} r_1'$, $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp_1 : \mathsf{rda} \to \mathsf{rda}'$, $\langle h_1, pp_1, r_1 \rangle \rightsquigarrow^{(0)}_{P,m} \langle h_2, pp_2, r_2 \rangle$, $\langle h_1', pp_1, r_1' \rangle \rightsquigarrow^{(0)}_{P,m} \langle h_2', pp_2', r_2' \rangle$, and $pp_2 \neq pp_2'$.

The only instruction that may yield different program points after execution is `if-test`.

$$\text{tIfTest}\;\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad \forall j \in region_m(pp). \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_b) \sqsubseteq se(j)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

$$\text{rIfTestTrue}\;\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad r(v_a)\ \underline{rop}\ r(v_b)}{\langle h, pp, r \rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp + n, r \rangle}$$

$$\text{rIfTestFalse}\;\frac{m[pp] = \texttt{if-test}\ v_a, v_b, n, rop \qquad \neg(r(v_a)\ \underline{rop}\ r(v_b))}{\langle h, pp, r \rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp + 1, r \rangle}$$

Due to the premise of the typing rule (tIfTest), $se(pp') = low$ for some $pp' \in region_m(pp_1)$ can only hold if $\mathsf{rda}(v_a) = low$ and $\mathsf{rda}(v_b) = low$. Under this assumption, we have $r_1(v_a) \sim_\beta r_1'(v_a)$ and $r_1(v_b) \sim_\beta r_1'(v_b)$ because $r_1 \sim_{\beta,\mathsf{rda}} r_1'$.

If the registers v_a, v_b store numbers from the set $\mathcal{N}$, this directly implies $r_1(v_a) = r_1'(v_a)$ and $r_1(v_b) = r_1'(v_b)$. Thus, $r_1(v_a)\ \underline{rop}\ r_1(v_b)$ if and only if $r_1'(v_a)\ \underline{rop}\ r_1'(v_b)$.

If v_a, v_b store locations, then $\beta(r_1(v_a)) = r_1'(v_a)$ and $\beta(r_1(v_b)) = r_1'(v_b)$. The only operators applicable to locations are $=$ and $\neq$. Since β is an injective function, $r_1(v_a)\ \underline{rop}\ r_1(v_b)$ if and only if $r_1'(v_a)\ \underline{rop}\ r_1'(v_b)$ for $rop \in \{=, \neq\}$.

Because $r_1(v_a)\ \underline{rop}\ r_1(v_b)$ if and only if $r_1'(v_a)\ \underline{rop}\ r_1'(v_b)$, the same semantic rule (rIfTestTrue) or (rIfTestFalse) is applicable in both executions and, thus, yields the same program point to be executed next. As this is a contradiction to the assumption that $pp_2 \neq pp_2'$, we can conclude that $\mathsf{rda}(v_a) = high$ or $\mathsf{rda}(v_b) = high$. By rule (tIfTest), this implies that $se(pp') = high$ for all $pp' \in region_m(pp_1)$.

Lemma 7 (Indistinguishable after high branch). *For all methods $m \in \mathcal{M}_P$ of a typable program P, register domain assignments $\mathsf{rda}_0, \ldots, \mathsf{rda}_k \in \mathcal{RDA}$ where $k = \mathsf{length}(m) - 1$, security environments $se : \mathbb{N}_0 \to \mathcal{SL}$, security domains $ret \in \mathcal{SL}$, partial injective functions $\beta \in \mathcal{B}$, natural numbers $i \in \mathbb{N}_0$, program points $pp^0, \ldots, pp^i \in \mathbb{N}_0$, register states $r, r^0, \ldots, r^i \in \mathcal{R}$, and heaps $h, h^0, \ldots, h^i \in \mathcal{H}$ such that*

1. *$se(pp^n) = high$ for all $n \in \mathbb{N}_0$ with $n < i$,*
2. *$h \sim_\beta h^0$,*

3. $r \sim_{\beta,\mathsf{rda}_{pp^0}} r^0$,
4. $\langle h^0, pp^0, r^0\rangle \rightsquigarrow^{(0)}_{P,m} \langle h^1, pp^1, r^1\rangle \rightsquigarrow^{(0)}_{P,m} \cdots \rightsquigarrow^{(0)}_{P,m} \langle h^i, pp^i, r^i\rangle$, *and*
5. *for all* $i, j \in \mathbb{N}_0$, *if* $i \rightarrow_m j$ *there exists a register domain assignment* $\mathsf{rda}'_j \in \mathcal{RDA}$ *such that the judgment* $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \rightarrow \mathsf{rda}'_j$ *is derivable and* $\mathsf{rda}'_j \sqsubseteq \mathsf{rda}_j$,

then $h \sim_\beta h^i$, *and* $r \sim_{\beta,\mathsf{rda}_{pp^i}} r^i$.

Proof. Let $m \in \mathcal{M}_P$ be a method of a typable program P, $\mathsf{rda}_0, \ldots, \mathsf{rda}_k \in \mathcal{RDA}$ be register domain assignments where $k = \mathsf{length}(m) - 1$, $se \in \mathbb{N}_0 \rightarrow \mathcal{SL}$ be a security environment, $ret \in \mathcal{SL}$ be a security domain, $\beta \in \mathcal{B}$ be a partial injective function, $i \in \mathbb{N}_0$ be a natural number, $pp^0, \ldots, pp^i \in \mathbb{N}_0$ be program points, $r, r^0, \ldots, r^i \in \mathcal{R}$ be register states, and $h, h^0, \ldots, h^i \in \mathcal{H}$ be heaps such that $se(pp^n) = high$ for all $n \in \mathbb{N}_0, n < i$, $h \sim_\beta h^0$, $r \sim_{\beta,\mathsf{rda}_{pp^0}} r^0$, $\langle h^0, pp^0, r^0\rangle \rightsquigarrow^{(0)}_{P,m} \langle h^1, pp^1, r^1\rangle \rightsquigarrow^{(0)}_{P,m} \cdots \rightsquigarrow^{(0)}_{P,m} \langle h^i, pp^i, r^i\rangle$, and for all $i, j \in \mathbb{N}_0$, if $i \rightarrow_m j$ there exists a register domain assignment $\mathsf{rda}'_j \in \mathcal{RDA}$ such that the judgment $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \rightarrow \mathsf{rda}'_j$ is derivable and $\mathsf{rda}'_j \sqsubseteq \mathsf{rda}_j$.

We have to show that $h \sim_\beta h^i$ and $r \sim_{\beta,\mathsf{rda}_{pp^i}} r^i$. We conduct the proof by induction over the length of the execution sequence i.

Base case. Assume $i = 0$. Then $h \sim_\beta h^0$ and $r \sim_{\beta,\mathsf{rda}_{pp^0}} r^0$ hold by assumption.

Induction hypothesis. We assume that the property holds for execution sequences that are strictly shorter than i.

Induction step. Assume $i > 0$. We inspect the first execution step in the sequence $\langle h^0, pp^0, r^0\rangle \rightsquigarrow^{(0)}_{P,m} \langle h^1, pp^1, r^1\rangle$. With $h \sim_\beta h^0$, $r \sim_{\beta,\mathsf{rda}_{pp^0}} r^0$, $se(pp^0) = high$, and the fact that each program point is typable, we can apply Lemma 4 (step consistent) and Lemma 20 (monotonicity of the indistinguishability of register states), to conclude $h \sim_\beta h^1$ and $r \sim_{\beta,\mathsf{rda}_{pp^1}} r^1$.

Since the remainder of the execution sequence $\langle h^1, pp^1, r^1\rangle \rightsquigarrow^{(0)}_{P,m} \cdots \rightsquigarrow^{(0)}_{P,m} \langle h^i, pp^i, r^i\rangle$ has now $i-1$ steps remaining, we can apply the induction hypothesis with $h \sim_\beta h^1$, $r \sim_{\beta,\mathsf{rda}_{pp^1}} r^1$ and the premises (1) and (5) to conclude $h \sim_\beta h^i$ and $r \sim_{\beta,\mathsf{rda}_{pp^i}} r^i$.

Lemma 8 (Security of typable sequences). *For all methods* $m \in \mathcal{M}_P$ *of a typable program* P, *register domain assignments* $\mathsf{rda}_0, \ldots, \mathsf{rda}_k \in \mathcal{RDA}$ *where* $k = \mathsf{length}(m) - 1$, *security environments* $se : \mathbb{N}_0 \rightarrow \mathcal{SL}$, *security domains* $ret \in \mathcal{SL}$, *partial injective functions* $\beta \in \mathcal{B}$, *natural numbers* $i, j, n_2^0, \ldots, n_2^j \in \mathbb{N}_0$, *program points* $pp_1^0, \ldots, pp_1^i, pp_2^0, \ldots, pp_2^j \in \mathbb{N}_0$, *register states* $r_1^0, \ldots, r_1^i, r_2^0, \ldots, r_2^j \in \mathcal{R}$, *heaps* $h_1^0, \ldots, h_0^i, h_2^0, \ldots, h_2^j, h_2 \in \mathcal{H}$, *and values* $u_2 \in \mathcal{V}$ *such that*

1. $pp_1^0 = pp_2^0$,

2. $r_1^0 \sim_{\beta,\mathsf{rda}_{pp_1^0}} r_2^0$,
3. $h_1^0 \sim_\beta h_2^0$,
4. $se(pp_1^i) = low$,
5. $\langle h_1^0, pp_1^0, r_1^0\rangle \rightsquigarrow_{P,m}^{(0)} \langle h_1^1, pp_1^1, r_1^1\rangle \rightsquigarrow_{P,m}^{(0)} \cdots \rightsquigarrow_{P,m}^{(0)} \langle h_1^i, pp_1^i, r_1^i\rangle$,
6. $\langle h_2^0, pp_2^0, r_2^0\rangle \rightsquigarrow_{P,m}^{(n_2^0)} \langle h_2^1, pp_2^1, r_2^1\rangle \rightsquigarrow_{P,m}^{(n_2^1)} \cdots \langle h_2^j, pp_2^j, r_2^j\rangle \rightsquigarrow_{P,m}^{(n_2^j)} \langle u_2, h_2\rangle$,
7. *for all* $i, j \in \mathbb{N}_0$, *if* $i \to_m j$ *there exists a register domain assignment* $\mathsf{rda}'_j \in \mathcal{RDA}$ *such that the judgment* $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}'_j$ *is derivable and* $\mathsf{rda}'_j \sqsubseteq \mathsf{rda}_j$, *and*
8. *for all* $i \in \mathbb{N}_0$, *if there exists no* $j \in \mathbb{N}_0$ *such that* $i \to_m j$, *then the judgment* $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}_i$ *is derivable.*

there exists a natural number $d \in \mathbb{N}_0$ *and a partial injective function on locations* $\beta' \in \mathcal{B}$, *such that*

1. $d \leq j$,
2. $pp_1^i = pp_2^d$,
3. $\beta \subseteq \beta'$,
4. $h_1^i \sim_{\beta'} h_2^d$,
5. $r_1^i \sim_{\beta',\mathsf{rda}_{pp_1^i}} r_2^d$, *and*
6. *for all* $c \in \mathbb{N}_0, c < d$ *it holds that* $n_2^c = 0$.

Proof. Let $m \in \mathcal{M}_P$ be a method of a typable program P, $\mathsf{rda}_0, \ldots, \mathsf{rda}_k \in \mathcal{RDA}$ be register domain assignments where $k = \mathsf{length}(m) - 1$, $se \in \mathbb{N}_0 \to \mathcal{SL}$ be a security environment, $ret \in \mathcal{SL}$ be a security domain, $\beta \in \mathcal{B}$ be a partial injective function, $i, j, n_2^0, \ldots, n_2^j \in \mathbb{N}_0$ be natural numbers, $pp_1^0, \ldots, pp_1^i, pp_2^0, \ldots, pp_2^j \in \mathbb{N}_0$ be program points, $r_1^0, \ldots, r_1^i, r_2^0, \ldots, r_2^j \in \mathcal{R}$ be register states, $h_1^0, \ldots, h_0^i, h_2^0, \ldots, h_2^j, h_2 \in \mathcal{H}$ be heaps, and $u_2 \in \mathcal{V}$ be a value such that $pp_1^0 = pp_2^0$, $r_1^0 \sim_{\beta,\mathsf{rda}_{pp_1^0}} r_2^0$, $h_1^0 \sim_\beta h_2^0$, $se(pp_1^i) = low$, $\langle h_1^0, pp_1^0, r_1^0\rangle \rightsquigarrow_{P,m}^{(0)} \langle h_1^1, pp_1^1, r_1^1\rangle \rightsquigarrow_{P,m}^{(0)} \cdots \rightsquigarrow_{P,m}^{(0)} \langle h_1^i, pp_1^i, r_1^i\rangle$, $\langle h_2^0, pp_2^0, r_2^0\rangle \rightsquigarrow_{P,m}^{(n_2^0)} \langle h_2^1, pp_2^1, r_2^1\rangle \rightsquigarrow_{P,m}^{(n_2^1)} \cdots \langle h_2^j, pp_2^j, r_2^j\rangle \rightsquigarrow_{P,m}^{(n_2^j)} \langle u_2, h_2\rangle$, for all $i, j \in \mathbb{N}_0$, if $i \to_m j$ there exists a register domain assignment $\mathsf{rda}'_j \in \mathcal{RDA}$ such that the judgment $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}'_j$ is derivable and $\mathsf{rda}'_j \sqsubseteq \mathsf{rda}_j$, and for all $i \in \mathbb{N}_0$, if there exists no $j \in \mathbb{N}_0$ such that $i \to_m j$, then the judgment $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}_i$ is derivable.

We have to show that there exists a natural number $d \in \mathbb{N}_0$ and a partial injective function on locations $\beta' \in \mathcal{B}$, such that $d \leq j$, $pp_1^i = pp_2^d$, $\beta \subseteq \beta'$, $h_1^i \sim_{\beta'} h_2^d$, $r_1^i \sim_{\beta',\mathsf{rda}_{pp_1^i}} r_2^d$, and for all $c \in \mathbb{N}_0, c < d$ it holds that $n_2^c = 0$.

We prove this by induction over the number i of execution steps in the first sequence.

Base case. Assume $i = 0$. Then $d = i = 0$ and $\beta' = \beta$ and all goals are fulfilled by assumption.

Induction hypothesis. We assume that the property holds for execution sequences that are strictly shorter than i.

Induction step. Assume $i > 0$. Since $pp_1^0 = pp_2^0$ and the instruction at pp_1^0 is not a return statement (it does not lead to a final state), we know that also the second sequence must make at least one step that is not terminating. We distinguish whether the security environment at the first program point is low or high.

Case 1 ($se(pp_1^0) = low$). We can apply locally respect (Lemma 1) and Lemma 20 to obtain a $\beta'' \in \mathcal{B}$ such that $r_1^1 \sim_{\beta'',\mathsf{rda}_{pp_1^1}} r_2^1$, $h_1^1 \sim_\beta h_2^1$, and $n_2^0 = 0$. If $pp_1^1 = pp_2^1$, we can apply the induction hypothesis and conclude all goals.

Otherwise, $pp_1^1 \neq pp_2^1$ and the instruction at program point pp_1^0 was a branching with a condition involving secrets. With Lemma 6 (high branching) and the three SOAP properties of control dependence regions of branching instructions (Definition 26), we know that all program points whose execution depends on the given branching (all $pp \in region_m(pp_1^0)$) have a high security environment. Since we also know that the security environment of pp_1^i is low by assumption, there has to be a natural number $c \in \mathbb{N}_0$ with $c < i$ which is the smallest number such that $pp_1^c = jun_m(pp_1^0)$. Hence, the program point of this state is the junction point of the different possible executions originating from the branching at pp_1^0 and this program point is not in $region_m(pp_1^0)$. According to SOAP 3, this junction point is only defined if no return statement occurs in the control dependence region of pp_1^0. Hence, the second execution must also pass this junction point pp_1^c before terminating. Thus, we define d as the smallest number such that $pp_1^c = pp_2^d = jun_m(pp_1^0)$.

As all states before $jun_m(pp_1^0)$ have a high security environment and no methods can be called in high security environments required by the typability of the program, we have $n_2^1 = \ldots = n_2^{d-1} = 0$. Hence, we can apply Lemma 7 (indistinguishable after high branch) to show for the execution sequences

$$\langle h_1^1, pp_1^1, r_1^1\rangle \rightsquigarrow_{P,m}^{(0)} \cdots \langle h_1^c, pp_1^c, r_1^c\rangle \text{ and}$$
$$\langle h_2^1, pp_2^1, r_2^1\rangle \rightsquigarrow_{P,m}^{(n_2^1)} \cdots \langle h_2^d, pp_2^d, r_2^d\rangle$$

that $h_1^1 \sim_{\beta''} h_1^c$, $h_2^1 \sim_{\beta''} h_2^d$, and with $h_1^1 \sim_\beta h_2^1$ and $\beta \subseteq \beta''$ we have $h_1^c \sim_{\beta''} h_2^d$ with the transitivity and symmetry of the indistinguishability of heaps. Moreover, we get from Lemma 7 that $r_1^1 \sim_{\beta'',\mathsf{rda}_{pp_1^c}} r_1^c$, $r_1^1 \sim_{\beta'',\mathsf{rda}_{pp_1^c}} r_2^d$ (since $pp_1^c = pp_2^d$). With the symmetry and transitivity of the indistinguishability of register states, we have $r_1^c \sim_{\beta'',\mathsf{rda}_{pp_1^c}} r_2^d$. Since the length of the remainder of the first execution sequence is smaller than i, we can now apply the induction hypothesis and conclude all goals.

Case 2 ($se(pp_1^0) = high$). If the security environment is high at pp_1^0, then there exists some program point $pp \in \mathbb{N}_0$ that is a branching on secrets with $pp_0^1 \in region_m(pp)$ and that has a junction point $jun_m(pp)$ which is not in a high security environment. Otherwise, $se(pp_1^i)$ could not be low. The rest of this case is shown analogously to the second case of $se(pp_1^0) = high$.

Lemma 9 (Security of typable methods). *For all methods $m \in \mathcal{M}_P$ of a typable program P, method names $mid \in \mathcal{MID}_P$, and security domains $p_0, \dots p_n$, $ret \in \mathcal{SL}$ for some $n \in \mathbb{N}_0$, if $(mid, [p_0, \dots, p_n], ret) \in \mathsf{mda}$ and m is typable with respect to $(mid, [p_0, \dots, p_n], ret)$, then m satisfies TIN-ADL with respect to the method signature $(mid, [p_0, \dots, p_n], ret)$.*

Proof. Let $m \in \mathcal{M}_P$ be an arbitrary method of a typable program P, $mid \in \mathcal{MID}_P$ be a method name, and $p_0, \dots p_n, ret \in \mathcal{SL}$ for some $n \in \mathbb{N}_0$ be security domains such that $(mid, [p_0, \dots, p_n], ret) \in \mathsf{mda}$ and m is typable with respect to $(mid, [p_0, \dots, p_n], ret)$.

To show that m satisfies *TIN-ADL* with respect to $(mid, [p_0, \dots, p_n], ret)$, we have to show that there exists a register domain assignment $\mathsf{rda} \in \mathcal{RDA}$ with $p_i \sqsubseteq \mathsf{rda}(v_i)$ for all $i \in \mathbb{N}_0$, $i \leq n$ and for all partial injective functions $\beta \in \mathcal{B}$, register states $r_1, r_2 \in \mathcal{R}$, heaps $h_1, h_2, h_1', h_2' \in \mathcal{H}$, return values $u_1, u_2 \in \mathcal{V}$, and natural numbers $n_1, n_2 \in \mathbb{N}_0$ such that

$$r_1 \sim_{\beta,\mathsf{rda}} r_2,$$
$$h_1 \sim_\beta h_2,$$
$$\langle h_1, 0, r_1 \rangle \Downarrow_{P,m}^{(n_1)} \langle u_1, h_1' \rangle, \text{ and}$$
$$\langle h_2, 0, r_2 \rangle \Downarrow_{P,m}^{(n_2)} \langle u_2, h_2' \rangle,$$

there exists a partial injective function on locations $\beta' \in \mathcal{B}$, such that $\beta \subseteq \beta'$, $h_1' \sim_{\beta'} h_2'$ and, if $ret = low$, $u_1 \sim_{\beta'} u_2$.

By the typability of m and Definition 28, we obtain $\mathsf{rda}_0 \in \mathcal{RDA}$ such that for all $i \in \mathbb{N}_0$, $i \leq n$ it holds that $p_i \sqsubseteq \mathsf{rda}_0(v_i)$ and set $\mathsf{rda} = \mathsf{rda}_0$.

For the remaining goals, we show a more general case where executions of arbitrary methods start at arbitrary positions.

Let $m \in \mathcal{M}_P$ be an arbitrary method of a typable program P, $se \in \mathbb{N}_0 \to \mathcal{SL}$ be a security environment, $\mathsf{rda}_0, \dots, \mathsf{rda}_k \in \mathcal{RDA}$ be register domain assignments where $k = \mathsf{length}(m) - 1$, $\beta \in \mathcal{B}$ be a partial injective function, $pp_1^0, pp_2^0 \in \mathbb{N}_0$ be program points, $r_1^0, r_2^0 \in \mathcal{R}$ be register states, $h_1^0, h_2^0, h_1, h_2 \in \mathcal{H}$ be heaps, $u_1, u_2 \in \mathcal{V}$ be return values, and $n_1, n_2 \in \mathbb{N}_0$ be natural numbers such that

1. $pp_1^0 = pp_2^0$,
2. $r_1^0 \sim_{\beta,\mathsf{rda}_{pp_1^0}} r_2^0$,
3. $h_1^0 \sim_\beta h_2^0$,
4. $\langle h_1^0, pp_1^0, r_1^0 \rangle \Downarrow_{P,m}^{(n_1)} \langle u_1, h_1 \rangle$,
5. $\langle h_2^0, pp_2^0, r_2^0 \rangle \Downarrow_{P,m}^{(n_2)} \langle u_2, h_2 \rangle$,
6. for all $i, j \in \mathbb{N}_0$, if $i \to_m j$ there exists a register domain assignment $\mathsf{rda}_j' \in \mathcal{RDA}$ such that the judgment $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}_j'$ is derivable and $\mathsf{rda}_j' \sqsubseteq \mathsf{rda}_j$, and
7. for all $i \in \mathbb{N}_0$, if there exists no $j \in \mathbb{N}_0$ such that $i \to_m j$, then the judgment $m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash i : \mathsf{rda}_i \to \mathsf{rda}_i$ is derivable.

We have to show that there exists a partial injective function on locations $\beta' \in \mathcal{B}$, such that $\beta \subseteq \beta'$, $h_1 \sim_{\beta'} h_2$ and, if $ret = low$, $u_1 \sim_{\beta'} u_2$.

By unfolding the semantics of methods, we know that the executions of m are of the form

$$\langle h_1^0, pp_1^0, r_1^0\rangle \rightsquigarrow_{P,m}^{(n_1^0)} \langle h_1^1, pp_1^1, r_1^1\rangle \cdots \langle h_1^i, pp_1^i, r_1^i\rangle \rightsquigarrow_{P,m}^{(n_1^i)} \langle u_1, h_1\rangle$$

and

$$\langle h_2^0, pp_2^0, r_2^0\rangle \rightsquigarrow_{P,m}^{(n_2^0)} \langle h_2^1, pp_2^1, r_2^1\rangle \cdots \langle h_2^j, pp_2^j, r_2^j\rangle \rightsquigarrow_{P,m}^{(n_2^j)} \langle u_2, h_2\rangle$$

for natural numbers $i, j, n_1^0, n_2^0, \ldots, n_1^i, n_2^j \in \mathbb{N}_0$, heaps $h_1^1, h_2^1, \ldots, h_1^i, h_2^j \in \mathcal{H}$, register states $r_1^1, r_2^1, \ldots, r_1^i, r_2^j \in \mathcal{R}$, and program points $pp_1^1, pp_2^1, \ldots, pp_1^i, pp_2^j \in \mathbb{N}_0$ such that $n_1^0 + \ldots + n_1^i = n_1$ and $n_2^0 + \ldots + n_2^j = n_2$.

We show the goal by induction over an upper bound for the number of method calls $n_0 \in \mathbb{N}_0$ where $n_1 \leq n_0$ and $n_2 \leq n_0$.

Base case. Assume $n_0 = 0$. Then also $n_1 = n_2 = 0$. We distinguish cases over the security environment of the program point of the return instruction pp_1^i.

Case 1 ($se[pp_1^i] = low$). Then, by the security of typable execution sequences without invocation (Lemma 8), we know that that there exists a $d \in \mathbb{N}_0$ such that $pp_1^i = pp_2^d$ and that there exists a $\beta'' \in \mathcal{B}$ such that $\beta \subseteq \beta''$, $h_1^i \sim_{\beta''} h_2^d$, and $r_1^i \sim_{\beta'', \mathsf{rda}_{pp_1^i}} r_2^d$. Since $pp_1^i = pp_2^d$ and pp_1^i is a return statement, also the second sequence terminates with execution step d, i.e., $d = j$. With premise 7 of this lemma and locally respect for return (Lemma 2), we conclude for the last execution step that there exists a partial injective function on locations $\beta' \in \mathcal{B}$, such that $\beta'' \subseteq \beta'$, $h_1 \sim_{\beta'} h_2$ and, if $ret = low$, $u_1 \sim_{\beta'} u_2$. The goal $\beta \subseteq \beta'$ follows from $\beta \subseteq \beta''$ and $\beta'' \subseteq \beta'$.

Case 2 ($se[pp_1^i] = high$). Then there is a state $\langle h_1^c, pp_1^c, r_1^c\rangle$ for some $c \in \mathbb{N}_0$ such that either there is a preceding state with $se(pp_1^{c-1}) = low$ or $\langle h_1^c, pp_1^c, r_1^c\rangle = \langle h_1^0, pp_1^0, r_1^0\rangle$, and for all $n \in \mathbb{N}_0$ with $c \leq n \leq i$ it holds that $se(pp_1^n) = high$, i.e., no more state in a low security environment occurs in the remainder of the first execution before termination.

In the first case, by security of typable execution sequences without invocation (Lemma 8), we know that there exists a state $\langle h_2^{d-1}, pp_2^{d-1}, r_2^{d-1}\rangle$ for some $d \in \mathbb{N}_0$ in the second execution such that $pp_1^{c-1} = pp_2^{d-1} = pp$ for some $pp \in \mathbb{N}_0$ and that there exists a $\beta''' \in \mathcal{B}$ such that $\beta \subseteq \beta'''$, $h_1^{c-1} \sim_{\beta'''} h_2^{d-1}$, and $r_1^{c-1} \sim_{\beta''', \mathsf{rda}_{pp}} r_2^{d-1}$. The state $\langle h_2^{d-1}, pp_2^{d-1}, r_2^{d-1}\rangle$ must also be the last state in a low security environment before termination of the second execution, as otherwise Lemma 8 could be applied to derive another state with a program point in a low security environment in the first execution, which is a contradiction to our assumption. By locally respect (Lemma 1), we know that the execution of pp_1^{c-1} and pp_2^{d-1} in indistinguishable register states and heaps yields states $\langle h_1^c, pp_1^c, r_1^c\rangle$, and $\langle h_2^d, pp_2^d, r_2^d\rangle$ such that there exists a $\beta'' \in \mathcal{B}$ such that $\beta''' \subseteq \beta''$, $h_1^c \sim_{\beta''} h_2^d$, $r_1^c \sim_{\beta'', \mathsf{rda}_{pp_1^c}} r_2^d$, and $r_1^c \sim_{\beta'', \mathsf{rda}_{pp_2^d}} r_2^d$.

In the second case, we have $\beta'' = \beta$, $pp = pp_1^c = pp_1^0 = pp_2^0 = pp_2^d$, $r_1^c = r_1^0 \sim_{\beta'', \mathsf{rda}_{pp}} r_2^0 = r_2^d$, and $h_1^c = h_1^0 \sim_{\beta''} h_2^0 = h_2^d$ by assumption. As before, this implies that also the second execution sequence does not contain a state with a program point in a low security environment, as otherwise a contradiction would be derivable.

In both cases, we can apply Lemma 7 (indistinguishable after high branch) to the execution sequences

$$\langle h_1^c, pp_1^c, r_1^c\rangle \rightsquigarrow_{P,m}^{(0)} \cdots \langle h_1^i, pp_1^i, r_1^i\rangle \text{ and}$$
$$\langle h_2^d, pp_2^d, r_2^d\rangle \rightsquigarrow_{P,m}^{(0)} \cdots \langle h_2^j, pp_2^j, r_2^j\rangle$$

and obtain $h_1^c \sim_{\beta''} h_1^i$ and $h_2^d \sim_{\beta''} h_2^j$. With $h_1^c \sim_{\beta''} h_2^d$ and symmetry and transitivity of indistinguishability immediately follows $h_1^i \sim_{\beta''} h_2^j$.

We can apply step consistent for return (Lemma 5) to the final execution step in both sequences and get $h_1 \sim_{\beta''} h_1^i$, $h_2 \sim_{\beta''} h_2^j$, $ret = high$ if $u_1 \neq \mathsf{void}$, and $ret = high$ if $u_2 \neq \mathsf{void}$. From $h_1 \sim_{\beta''} h_1^i$, $h_2 \sim_{\beta''} h_2^j$, and $h_1^i \sim_{\beta''} h_2^j$, we know that $h_1 \sim_{\beta''} h_2$ given the symmetry and transitivity of the indistinguishability relation for heaps.

Finally, if $ret = low$, we know from $ret = high$ if $u_1 \neq \mathsf{void}$, and $ret = high$ if $u_2 \neq \mathsf{void}$, that $u_1 = u_2 = \mathsf{void}$ and, thus, $u_1 \sim_{\beta''} u_2$.

Induction hypothesis. We assume that the property made explicit by Definition 30 holds for terminating execution sequences in arbitrary methods m with strictly less than n_0 method calls, given that they start in the same program point and indistinguishable register states and heaps.

Induction step. Assume $n_0 > 0$. Let $\langle h_1^c, pp_1^c, r_1^c\rangle$ for some $c \in \mathbb{N}_0$ be the first state in the first execution sequence in which $m[pp_1^c]$ is a method call instruction. As m is typable, we know that $se(pp_1^c) = low$ as required by the typing rules for method invocation instructions.

By Lemma 8, we know that a state $\langle h_2^d, pp_2^d, r_2^d\rangle$ for some $d \in \mathbb{N}_0$ in the second execution exists, such that no method invoking instruction occurs in the second execution before this state, $pp_1^c = pp_2^d = pp$ for some $pp \in \mathbb{N}_0$, and a function β'' exists, such that $\beta \subseteq \beta''$, $h_1^c \sim_{\beta''} h_2^d$, and $r_1^c \sim_{\beta'', \mathsf{rda}_{pp}} r_2^d$.

As $m[pp]$ is a method invoking instruction, the next execution steps from the states $\langle h_1^c, pp, r_1^c\rangle$ and $\langle h_2^d, pp, r_2^d\rangle$ are of the form

$$\langle h_1^c, pp, r_1^c\rangle \rightsquigarrow_{P,m}^{(z_1+1)} \langle h_1^{c+1}, pp+1, r_1^{c+1}\rangle \text{ and}$$
$$\langle h_2^d, pp, r_2^d\rangle \rightsquigarrow_{P,m}^{(z_2+1)} \langle h_2^{d+1}, pp+1, r_2^{d+1}\rangle$$

where $z_1, z_2 \in \mathbb{N}_0$. As the total amount of method calls is not greater than n_0, we know that $z_1 < n_0$ and $z_2 < n_0$. With the induction hypothesis, the lemma locally respect for methods (Lemma 3), and monotonicity of the indistinguishability of register states (Lemma 20), we know that there exists a $\mathsf{rda}' \in \mathcal{RDA}$ and a $\beta''' \in \mathcal{B}$ with $\beta'' \subseteq \beta'''$ such that $h_1^{c+1} \sim_{\beta'''} h_2^{d+1}$ and $r_1^{c+1} \sim_{\beta''', \mathsf{rda}'} r_2^{d+1}$. With premise (6) of this lemma, we get $r_1^{c+1} \sim_{\beta''', \mathsf{rda}_{pp+1}} r_2^{d+1}$. Since the remaining executions can only contain $n_1 - (z_1 + 1)$ and $n_2 - (z_2 + 1)$ method invoking instructions, we can apply the induction hypothesis and conclude that there exists a partial injective function on locations $\beta' \in \mathcal{B}$, such that $\beta \subseteq \beta'$, $h_1 \sim_{\beta'} h_2$ and, if $ret = low$, $u_1 \sim_{\beta'} u_2$.

With the last lemma, security of typable methods, Theorem 1 can be proven.

Proof (Soundness of the type system). We assume that program P is typable. To show that P then also satisfies *TIN-ADL*, we have to show

1. for all method names of entry points $mid \in \mathsf{EP}_P$ there exists $p_0, \ldots p_n, ret \in \mathcal{SL}$ for some $n \in \mathbb{N}_0$ such that $(mid, [p_0, \ldots p_n], ret) \in \mathsf{mda}$, and
2. for all method names of entry points $mid \in \mathsf{EP}_P$, methods $m \in \mathcal{M}_P$, classes $c \in \mathcal{CID}_P$, and security domains $p_0, \ldots p_n, ret \in \mathcal{SL}$ such that $(mid, [p_0, \ldots p_n], ret) \in \mathsf{mda}$, if
   - $m = \mathsf{lookup\text{-}static}(mid)$,
   - $m = \mathsf{lookup\text{-}direct}(mid, c)$,
   - $m = \mathsf{lookup\text{-}super}(mid, c)$, or
   - $m = \mathsf{lookup\text{-}virtual}(mid, c)$

   holds, then m must satisfy *TIN-ADL* with respect to $(mid, [p_0, \ldots p_n], ret)$.

The satisfaction of condition 1 directly follows from the definition of typability of programs. It remains to show the satisfaction of condition 2.

Let $mid \in \mathsf{EP}_P$ be a method name of an entry point, $m \in \mathcal{M}_P$ be a method of P, $c \in \mathcal{CID}_P$ be a class name, and $p_0, \ldots p_n, ret \in \mathcal{SL}$ be security domains such that $(mid, [p_0, \ldots p_n], ret) \in \mathsf{mda}$, and $m = \mathsf{lookup\text{-}static}(mid)$, $m = \mathsf{lookup\text{-}direct}(mid, c)$, $m = \mathsf{lookup\text{-}super}(mid, c)$, or $m = \mathsf{lookup\text{-}virtual}(mid, c)$. We have to show that m satisfies *TIN-ADL* with respect to $(mid, [p_0, \ldots p_n], ret)$.

Since $\mathsf{EP}_P \subseteq \mathcal{MID}_P$, it follows from the typability of program P that m is either a framework method or it is typable with respect to $(mid, [p_0, \ldots p_n], ret)$.

If $m \in \mathsf{framework}$, then m satisfies *TIN-ADL* with respect to all applicable method signatures by assumption.

If $m \notin \mathsf{framework}$, then it is typable with respect to $(mid, [p_0, \ldots p_n], ret)$. Hence, m also satisfies *TIN-ADL* with respect to $(mid, [p_0, \ldots p_n], ret)$ using Lemma 9.

Thus, if program P is typable, then P also satisfies *TIN-ADL*.

6 Related Work

The objective of our work was the development of an information-flow analysis that soundly prevents information leakage through the bytecode of Android apps. Thus, the related work for this report comprises research on language-based information-flow security and on Android security. Language-based information-flow security has a long tradition, and a comrehensive overview to the field was given by Sabelfeld and Myers [SM03]. As for the Android security research, various methods have been proposed to prevent information leakage through apps, and Enck [Enc11] provides a broad overview of different directions here.

In this section, we first focus on relevant type-based information-flow security analyses with proven soundness, then we take a closer look on static analyses for the detection of information leaks in Android apps.

The first security-type analysis equipped with a formal proof of soundness was proposed by Volpano, Irvine, and Smith [VIS96]. They developed a security-type system for an imperative high-level programming language with formal semantics and proved that if a program in the given language is typable, it satisfies a noninterference-like security condition. Banerjee and Naumann [BN05] adopted this concept to define a sound security-type analysis for programs written in a fragment of JavaCard that supports objects and method invocation including dynamic dispatch. Although such type systems for high-level languages could be used to analyze the source code of Android apps, they are hard to apply if the source code is not available. For some apps, the source code even cannot be obtained using decompilers [EOMC11]. Our security type system was developed specifically to analyze Dalvik bytecode, such that access to the source code of an app or decompilation of Dalvik binaries are not necessary.

The first type-based information-flow analysis with proven soundness for a low-level language was proposed by Kobayashi and Shirane [KS02]. They analyzed a subset of the Java virtual machine language, i.e., Java bytecode, without objects and method calls and proved the soundness of the type system with respect to a noninterference-like security property. Barthe, Pichardie, and Rezk [BPR08] provided a type system and operational semantics for a larger subset of Java bytecode that includes objects, method calls, arrays, and exceptions. They defined a notion of noninterference for the execution semantics of Java bytecode and proved that the proposed type system enforces this notion of noninterference. Some aspects of our security-type system were adopted from [BPR08], e.g., the handling of indirect information flows in an unstructured bytecode language and some definitions of indistinguishability. Yet, there exist nontrivial differences between Java bytecode and Dalvik bytecode that have to be considered when defining a sound analysis method for Dalvik. For example, Dalvik programs have multiple potential entry points at which execution of the program may start while Java programs have a single main-method. Moreover, Dalvik bytecode operates on registers whereas the Java virtual machine uses an operand stack for computation results and parameters.

There exist different tools for the static detection of information leaks in Android apps, e.g., [FCF09, GCEC12, KYYS12, LLW+12, MS12, YY12, ZO12, FAR+13, OMJ+13]. However, only few come with a proof of soundness, i.e., formal guarantees to what extent their analyses enforce information-flow security. We are aware of two such tools.

The tool SCanDroid [FCF09] was developed based on a security-type analysis for a language in which the communication of apps with other apps can be captured [Cha09]. The goal of this analysis is to check that apps cannot circumvent their access permissions by colluding with other apps. To this end, the security-type analysis uses Android access permissions as security types and tracks information-flows across different apps. The soundness of the analysis was proven with respect to an operational semantics for the language. In addition to tracking information-flows across apps, which is not the focus of our work, SCanDroid can also track data flows within apps. Here, our analysis has two advantages: it does not only detect direct data leaks but also indirect leaks through control flow dependencies, and the security types in our analysis are independent of the statically declared Android permissions such that the same program can be analyzed with respect to different information-flow requirements.

ScanDal [KYYS12] is a static analysis tool to detect data leaks in Android apps. The analysis of ScanDal is based on abstract interpretation of programs represented in an own intermediate language, Dalvik Core, and it has been proven sound with respect to the formal semantics of Dalvik Core. In addition to detecting data leaks in Android apps, our security-type analysis takes indirect information leaks through control-flow dependencies into account.

7 Conclusion

In this report we presented the type-based information-flow analysis for Dalvik bytecode — together with its soundness result — that is implemented in Cassandra. This analysis not only supports the detection of direct information leaks but also of indirect leaks through control-flow dependencies on secrets. Such indirect leaks disclose private information at least partially and, if exploited repeatedly, may even leak complex information.

We carefully modeled the operational semantics for ADL, an abstract version of the Dalvik bytecode language, formalized the desired noninterference-like security property, defined the corresponding security-type system, and proved its soundness with respect to the security property and the semantics. Interestingly, conducting this soundness proof not only increased the confidence in the security guarantees that Cassandra provides, but also helped to detect and correct mistakes in Cassandra's implementation. For example, a leak of secret array indices was not detected, values could be leaked through fields inherited from classes of the Android framework, and objects on which methods were invoked could be leaked.

At the moment of writing this report, Cassandra as well as the presented underlying theory covered 211 out of 218 Dalvik bytecode instructions. Support for missing instructions (`check-cast`, `monitor-enter`, `monitor-exit`, `move-exception`, `packet-switch`, `sparse-switch`, and `throw`) is the subject of our ongoing work. Using Cassandra, we observed that a frequent cause of imprecision of the underlying type-based security analyses is the lack of object-sensitivity. In the future, we plan to investigate how our type system could be adapted to increase the precision in such cases without losing soundness.

Acknowledgements. We thank Alexander Lux and Jens Sauer for their suggestions during the development of the presented security-type system. This work was supported by the DFG under the project RSCP (MA3326/4-2) in the Priority Program RS^3, and by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE.

References

BN05. A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

BPR07. G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Noninterference Java Bytecode Verifier. In *European Symposium on Programming*, pages 125–140, 2007.

BPR08. G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. Technical report, INRIA Sophia Antipolis, France, 2008.

Cha09. A. Chaudhuri. Language-based Security on Android. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–7, 2009.

DD77. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

Enc11. W. Enck. Defending Users Against Smartphone Apps: Techniques and Future Directions. In *International Conference on Information Systems Security*, pages 49–70, 2011.

EOMC11. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, pages 315–330, 2011.

FAR+13. C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Highly Precise Taint Analysis for Android Applications. Technical report, TU Darmstadt, Germany, 2013.

FCF09. A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, University of Maryland, USA, 2009.

GCEC12. C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307, 2012.

GM82. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

KS02. N. Kobayashi and K. Shirane. Type-Based Information Analysis for Low-Level Languages. In *Asian Workshop on Programming Languages and Systems*, pages 302–316, 2002.

KYYS12. J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *Mobile Security Technologies*, 2012.

Lim. F-Droid Limited. F-Droid. `https://f-droid.org/`. Accessed in March 2014.

LLW+12. L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *ACM Conference on Computer and Communications Security*, pages 229–240, 2012.

Man11. C. Mann. A Static Framework for Privacy Analysis of Android Applications, 2011. Bachelor's Thesis, TU Darmstadt, Germany.

MS12. C. Mann and A. Starostin. A Framework for Static Detection of Privacy Leaks in Android Applications. In *Symposium on Applied Computing*, pages 1457–1462, 2012.

OMJ+13. D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *USENIX Conference on Security*, pages 543–558, 2013.

Proa. Android Open Source Project. Activities. `http://developer.android.com/guide/components/activities.html`. Accessed in March 2014.

Prob. Android Open Source Project. Android Security Overview. `http://source.android.com/devices/tech/security/`. Accessed in March 2014.

Proc. Android Open Source Project. Bytecode for the Dalvik VM. `http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`. Accessed in March 2014.

Prod. Android Open Source Project. Dalvik Bytecode Verifier Notes. `https://github.com/android/platform_dalvik/blob/c1b54205471ea7824c87e53e0d9e6d4c30518007/docs/verifier.html`. Accessed in March 2014.

SM03. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

TK10. S. Thurm and Y. Iwatani Kane. Your Apps Are Watching You. Homepage of Wall Street Journal, `http://online.wsj.com/news/articles/SB10001424052748704368004576027751867039730`, 2010. Accessed in March 2014.

VIS96. D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

YY12. Z. Yang and M. Yang. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *Third World Congress on Software Engineering*, pages 101–104, 2012.

ZO12. Z. Zhao and F. C. Colón Osorio. "TrustDroid": Preventing the Use of SmartPhones for Information Leaking in Corporate Networks through the Used of Static Analysis Taint Tracking. In *International Conference On Malicious And Unwanted Software*, pages 135–143, 2012.

Appendices

A Properties of Indistinguishability Relations

We prove twelve auxiliary lemmas in order to establish symmetry and transitivity of the indistinguishability relations. We refrain from explicitly proving reflexivity of the indistinguishability relations as it is apparent from their definition. In addition, we prove monotonicity of the indistinguishability of register states with respect to register security domains.

Lemma 10 (Symmetry of value indistinguishability). *Let $x, y \in \mathcal{V}$ and $\beta \in \mathcal{B}$ be a partial injective function on locations. Then $x \sim_\beta y$ implies $y \sim_{\beta^{-1}} x$.*

Proof. If $x = y = \mathsf{void}$ or $x, y \in \mathcal{N}$ with $x = y$, we have $y \sim_{\beta^{-1}} x$ by definition. If $x, y \in \mathcal{L}$, we have $\beta(x) = y$. As β is an injective function, this implies $y \in dom(\beta^{-1})$ and $\beta^{-1}(y) = x$. Therefore, we can conclude that $y \sim_{\beta^{-1}} x$.

Lemma 11 (Transitivity of value indistinguishability). *Let $x, y, z \in \mathcal{V}$ and $\beta, \beta' \in \mathcal{B}$ be a partial injective function on locations. Then $x \sim_\beta y$ and $y \sim_{\beta'} z$ imply $x \sim_{\beta' \circ \beta} z$.*

Proof. If $x = y = z = \mathsf{void}$ or $x, y, z \in \mathcal{N}$ with $x = y = z$, we have $x \sim_{\beta' \circ \beta} z$ by definition. If $x, y, z \in \mathcal{L}$, we have $\beta(x) = y$ and $\beta'(y) = z$, which implies $\beta'(\beta(y)) = z$. Therefore, we can conclude that $x \sim_{\beta' \circ \beta} z$.

Lemma 12 (Symmetry of register indistinguishability). *For $r, r' \in \mathcal{R}$, a given function $\mathsf{rda} \in \mathcal{RDA}$ and a partial injective function on locations $\beta \in \mathcal{B}$, $r \sim_{\beta,\mathsf{rda}} r'$ implies $r' \sim_{\beta^{-1},\mathsf{rda}} r$.*

Proof. $r \sim_{\beta,\mathsf{rda}} r'$ means that for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$ it holds that if $\mathsf{rda}(x) = low$ then $r(x) \sim_\beta r'(x)$. We need to show that for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$ it holds that if $\mathsf{rda}(x) = low$ then $r'(x) \sim_{\beta^{-1}} r(x)$. Let $x \in \mathcal{X} \cup \mathcal{X}_{res}$ and $\mathsf{rda}(x) = low$. We know that $r(x) \sim_\beta r'(x)$ and can apply Lemma 10 so that we have $r'(x) \sim_{\beta^{-1}} r(x)$. Hence, we can conclude that $r' \sim_{\beta^{-1},\mathsf{rda}} r$.

Lemma 13 (Transitivity of register indistinguishability). *For $r, r', r'' \in \mathcal{R}$, a given function $\mathsf{rda} \in \mathcal{RDA}$ and partial injective functions on locations $\beta, \beta' \in \mathcal{B}$, $r \sim_{\beta,\mathsf{rda}} r'$ and $r' \sim_{\beta',\mathsf{rda}} r''$ imply $r \sim_{\beta' \circ \beta,\mathsf{rda}} r''$.*

Proof. $r \sim_{\beta,\mathsf{rda}} r'$ and $r' \sim_{\beta',\mathsf{rda}} r''$ mean that for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$ it holds that if $\mathsf{rda}(x) = low$ then $r(x) \sim_\beta r'(x)$ and $r'(x) \sim_{\beta'} r''(x)$. We need to show that for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$ it holds that if $\mathsf{rda}(x) = low$ then $r(x) \sim_{\beta' \circ \beta} r''(x)$. Let $x \in \mathcal{X} \cup \mathcal{X}_{res}$ and $\mathsf{rda}(x) = low$. We know that $r(x) \sim_\beta r'(x)$ and $r'(x) \sim_{\beta'} r''(x)$ and can apply Lemma 11 to have $r(x) \sim_{\beta' \circ \beta} r''(x)$. Hence, we can conclude that $r \sim_{\beta' \circ \beta,\mathsf{rda}} r''$.

Lemma 14 (Symmetry of object indistinguishability). *Let $o_1, o_2 \in \mathcal{O}$ be two objects in a program P with the lookup function $\textsf{lookup-field}_P$ and $\beta \in \mathcal{B}$ a partial injective function on locations such that $o_1 \sim_\beta o_2$. Then $o_2 \sim_{\beta^{-1}} o_1$ holds.*

Proof. We know that $o_1.\mathsf{class} = o_2.\mathsf{class}$ and for all $f \in dom(o_1.\mathsf{fields})$ it holds that there exists $\mathit{fid} \in dom(\mathsf{lookup\text{-}field})$ such that if $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ and $\mathsf{fda}(\mathit{fid}) = low$ then $o_1.f \sim_\beta o_2.f$. As $o_2.\mathsf{class} = o_1.\mathsf{class}$ is satisfied by symmetry of equality, we still need to show for all $f \in dom(o_2.\mathsf{fields})$ that there exists $\mathit{fid} \in dom(\mathsf{lookup\text{-}field})$ such that if $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ and $\mathsf{fda}(\mathit{fid}) = low$ then $o_2.f \sim_{\beta^{-1}} o_1.f$. Let $f \in dom(o_2.\mathsf{fields})$ and $\mathit{fid} \in dom(\mathsf{lookup\text{-}field})$ such that $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ and $\mathsf{fda}(\mathit{fid}) = low$. As o_1 and o_2 are objects of the same class, $dom(o_2.\mathsf{fields}) = dom(o_1.\mathsf{fields})$. Thus, we have $o_1.f \sim_\beta o_2.f$ and can apply Lemma 10 to get $o_2.f \sim_{\beta^{-1}} o_1.f$. We can conclude that $o_2 \sim_{\beta^{-1}} o_1$.

Lemma 15 (Transitivity of object indistinguishability). *Let $o_1, o_2, o_3 \in \mathcal{O}$ be three objects in a program P with the lookup function $\mathsf{lookup\text{-}field}_P$ and $\beta, \beta' \in \mathcal{B}$ be two partial injective functions on locations, such that $o_1 \sim_\beta o_2$ and $o_2 \sim_{\beta'} o_3$. Then $o_1 \sim_{\beta' \circ \beta} o_3$ holds.*

Proof. We know that $o_1.\mathsf{class} = o_2.\mathsf{class}$ and for all $f \in dom(o_1.\mathsf{fields})$ it holds that there exists $\mathit{fid} \in dom(\mathsf{lookup\text{-}field})$ such that if $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ and $\mathsf{fda}(\mathit{fid}) = low$ then $o_1.f \sim_\beta o_2.f$. We also have $o_2.\mathsf{class} = o_3.\mathsf{class}$ and for all $f \in dom(o_2.\mathsf{fields})$ it holds that there exists $\mathit{fid} \in dom(\mathsf{lookup\text{-}field})$ such that if $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ and $\mathsf{fda}(\mathit{fid}) = low$ then $o_2.f \sim_{\beta'} o_3.f$. As $o_1.\mathsf{class} = o_3.\mathsf{class}$ is satisfied by transitivity of equality, we still need to show for all $f \in dom(o_1.\mathsf{fields})$ it holds that there exists $\mathit{fid} \in dom(\mathsf{lookup\text{-}field})$ such that if $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ and $\mathsf{fda}(\mathit{fid}) = low$ then $o_1.f \sim_{\beta' \circ \beta} o_3.f$. Let $f \in dom(o_1.\mathsf{fields})$ and $\mathit{fid} \in dom(\mathsf{lookup\text{-}field})$ such that $f = \mathsf{lookup\text{-}field}(\mathit{fid})$ and $\mathsf{fda}(\mathit{fid}) = low$. As o_1, o_2, and o_3 are objects of the same class, $dom(o_1.\mathsf{fields}) = dom(o_2.\mathsf{fields})$, i.e., $f \in dom(o_2.\mathsf{fields})$. Thus, we have $o_1.f \sim_\beta o_2.f$ and $o_2.f \sim_{\beta'} o_3.f$ and can apply Lemma 11 to get $o_1.f \sim_{\beta' \circ \beta} o_3.f$. We can conclude that $o_1 \sim_{\beta' \circ \beta} o_3$.

Lemma 16 (Symmetry of array indistinguishability). *Let $a_1, a_2 \in \mathcal{A}$ be two arrays and $\beta \in \mathcal{B}$ be a partial injective function on locations such that $a_1 \sim_\beta a_2$. Then $a_2 \sim_{\beta^{-1}} a_1$ holds.*

Proof. We know that $a_1.\mathsf{length} = a_2.\mathsf{length}$ and $\mathsf{ada} = low$ implies for all indices $i \in \mathbb{N}_0$ such that $i < a_1.\mathsf{length}$ that $a_1[i] \sim_\beta a_2[i]$. As $a_2.\mathsf{length} = a_1.\mathsf{length}$ is satisfied by symmetry of equality, we still need to show that $\mathsf{ada} = low$ implies $a_2[i] \sim_{\beta^{-1}} a_1[i]$ for all $i \in \mathbb{N}_0$ with $i < a_2.\mathsf{length}$. Let $\mathsf{ada} = low$ and $i \in \mathbb{N}_0$ such that $i < a_2.\mathsf{length}$. As the lengths are equal, we have $a_1[i] \sim_\beta a_2[i]$. By Lemma 10, this implies $a_2[i] \sim_{\beta^{-1}} a_1[i]$. Thus, we can conclude that $a_2 \sim_{\beta^{-1}} a_1$.

Lemma 17 (Transitivity of Array Indistinguishability). *Let $a_1, a_2, a_3 \in \mathcal{A}$ be three arrays and $\beta, \beta' \in \mathcal{B}$ be two partial injective functions on locations, such that $a_1 \sim_\beta a_2$ and $a_2 \sim_{\beta'} a_3$. Then $a_1 \sim_{\beta' \circ \beta} a_3$ holds.*

Proof. We know that $a_1.\mathsf{length} = a_2.\mathsf{length}$ and $\mathsf{ada} = low$ implies for all indices $i \in \mathbb{N}_0$ with $i < a_1.\mathsf{length}$ that $a_1[i] \sim_\beta a_2[i]$. We also know that $a_2.\mathsf{length} = a_3.\mathsf{length}$ and $\mathsf{ada} = low$ implies that for all indices $i \in \mathbb{N}_0$ with $i < a_2.\mathsf{length}$ that $a_2[i] \sim_{\beta'} a_3[i]$. As $a_1.\mathsf{length} = a_3.\mathsf{length}$ is satisfied by transitivity of equality, we still need to show that $\mathsf{ada} = low$ implies for all $i \in \mathbb{N}_0$ with $i < a_1.\mathsf{length}$

that $a_1[i] \sim_{\beta' \circ \beta} a_3[i]$. Let $\mathsf{ada} = \mathit{low}$ and $i \in \mathbb{N}_0$ such that $i < a_1.\mathsf{length}$. As the lengths are equal, we have $i < a_2.\mathsf{length}$ and therefore $a_1[i] \sim_\beta a_2[i]$ and $a_2[i] \sim_{\beta'} a_3[i]$. By Lemma 11, this implies $a_1[i] \sim_{\beta' \circ \beta} a_3[i]$. Thus, we can conclude that $a_1 \sim_{\beta' \circ \beta} a_3$.

Lemma 18 (Symmetry of heap indistinguishability). *Let $h_1, h_2 \in \mathcal{H}$ be two heaps and $\beta \in \mathcal{B}$ be a partial injective function on locations such that $h_1 \sim_\beta h_2$. Then $h_2 \sim_{\beta^{-1}} h_1$ holds.*

Proof. As β is a partial injective function on locations with $\mathit{dom}(\beta) \subseteq \mathit{dom}(h_1)$ and $\mathit{rng}(\beta) \subseteq \mathit{dom}(h_2)$, β^{-1} is a partial injective function on locations with $\mathit{dom}(\beta^{-1}) \subseteq \mathit{dom}(h_2)$ and $\mathit{rng}(\beta^{-1}) \subseteq \mathit{dom}(h_1)$. We need to show that for all $l \in \mathit{dom}(\beta^{-1})$, either $l \in \mathit{dom}_\mathcal{O}(h_2)$ and $\beta^{-1}(l) \in \mathit{dom}_\mathcal{O}(h_1)$ and $h_2(l) \sim_{\beta^{-1}} h_1(\beta^{-1}(l))$, or $l \in \mathit{dom}_\mathcal{A}(h_2)$ and $\beta^{-1}(l) \in \mathit{dom}_\mathcal{A}(h_1)$ and $h_2(l) \sim_{\beta^{-1}} h_1(\beta^{-1}(l))$. If $l \in \mathit{dom}(\beta^{-1})$ and $l \in \mathit{dom}_\mathcal{O}(h_2)$ and $\beta^{-1}(l) \in \mathit{dom}_\mathcal{O}(h_1)$, we know that $h_1(\beta^{-1}(l)) \sim_\beta h_2(l)$, because β is a partial injective function on locations, and can apply Lemma 14 to get $h_2(l) \sim_{\beta^{-1}} h_1(\beta^{-1}(l))$. If $l \in \mathit{dom}_\mathcal{A}(h_2)$ and $\beta^{-1}(l) \in \mathit{dom}_\mathcal{A}(h_1)$, we know that $h_1(\beta^{-1}(l)) \sim_\beta h_2(l)$ because β is a partial injective function on locations and can apply Lemma 16 to get $h_2(l) \sim_{\beta^{-1}} h_1(\beta^{-1}(l))$. Thus, we can conclude that $h_2 \sim_{\beta^{-1}} h_1$.

Lemma 19 (Transitivity of heap indistinguishability). *Let $h_1, h_2, h_3 \in \mathcal{H}$ be three heaps and $\beta, \beta' \in \mathcal{B}$ be two partial injective functions on locations such that $h_1 \sim_\beta h_2$ and $h_2 \sim_{\beta'} h_3$. Then $h_1 \sim_{\beta' \circ \beta} h_3$ holds.*

Proof. As β is a partial injective function on locations with $\mathit{dom}(\beta) \subseteq \mathit{dom}(h_1)$ and $\mathit{rng}(\beta) \subseteq \mathit{dom}(h_2)$ and β' is a partial injective function on locations with $\mathit{dom}(\beta') \subseteq \mathit{dom}(h_2)$ and $\mathit{rng}(\beta') \subseteq \mathit{dom}(h_3)$, $\beta' \circ \beta$ is a partial injective function on locations with $\mathit{dom}(\beta' \circ \beta) \subseteq \mathit{dom}(h_1)$ and $\mathit{rng}(\beta' \circ \beta) \subseteq \mathit{dom}(h_3)$. We need to show that for all $l \in \mathit{dom}(\beta' \circ \beta)$, either $l \in \mathit{dom}_\mathcal{O}(h_1)$ and $\beta'(\beta(l)) \in \mathit{dom}_\mathcal{O}(h_3)$ and $h_1(l) \sim_{\beta' \circ \beta} h_3(\beta'(\beta(l)))$, or $l \in \mathit{dom}_\mathcal{A}(h_1)$ and $\beta'(\beta(l)) \in \mathit{dom}_\mathcal{A}(h_3)$ and $h_1(l) \sim_{\beta' \circ \beta} h_3(\beta'(\beta(l)))$.

If $l \in \mathit{dom}(\beta' \circ \beta)$ and $l \in \mathit{dom}_\mathcal{O}(h_1)$, we know that $\beta(l) \in \mathit{dom}_\mathcal{O}(h_2)$ because $h_1 \sim_\beta h_2$ and furthermore $\beta'(\beta(l)) \in \mathit{dom}_\mathcal{O}(h_3)$ because $h_2 \sim_{\beta'} h_3$. Moreover we can apply Lemma 15 to $h_1(l) \sim_\beta h_2(\beta(l))$ and $h_2(\beta(l)) \sim_{\beta'} h_3(\beta'(\beta(l)))$, which both holds by assumption and the definition of heap indistinguishablility, and conclude $h_1(l) \sim_{\beta' \circ \beta} h_3(\beta'(\beta(l)))$.

If $l \in \mathit{dom}(\beta' \circ \beta)$ and $l \in \mathit{dom}_\mathcal{A}(h_1)$, we know that $\beta(l) \in \mathit{dom}_\mathcal{A}(h_2)$ because $h_1 \sim_\beta h_2$ and furthermore $\beta'(\beta(l)) \in \mathit{dom}_\mathcal{A}(h_3)$ because $h_2 \sim_{\beta'} h_3$. Moreover, we can apply Lemma 17 to $h_1(l) \sim_\beta h_2(\beta(l))$ and $h_2(\beta(l)) \sim_{\beta'} h_3(\beta'(\beta(l)))$, which both holds by assumption and the definition of heap indistinguishablility, and conclude $h_1(l) \sim_{\beta' \circ \beta} h_3(\beta'(\beta(l)))$. Thus, we can conclude that $h_1 \sim_{\beta' \circ \beta} h_3$.

Lemma 20 (Monotonicity of register state indistinguishability). *Let $r_1, r_2 \in \mathcal{R}$ be two register states, $\beta \in \mathcal{B}$ be a partial injective function on locations, and $\mathsf{rda} \in \mathcal{RDA}$ such that $r_1 \sim_{\beta,\mathsf{rda}} r_2$. Then for all $\mathsf{rda}' \in \mathcal{RDA}$ it holds that if $\mathsf{rda} \sqsubseteq \mathsf{rda}'$ then $r_1 \sim_{\beta,\mathsf{rda}'} r_2$.*

Proof. Let $r_1, r_2 \in \mathcal{R}$ be two register states, $\beta \in \mathcal{B}$ be a partial injective function on locations, and $\mathsf{rda}, \mathsf{rda}' \in \mathcal{RDA}$ such that $r_1 \sim_{\beta,\mathsf{rda}} r_2$ and $\mathsf{rda} \sqsubseteq \mathsf{rda}'$. We need to show that $r \sim_{\beta,\mathsf{rda}'} r'$.

We know for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$ that if $\mathsf{rda}(x) = \mathit{low}$ then $r(x) \sim_\beta r'(x)$. As $\mathsf{rda} \sqsubseteq \mathsf{rda}'$, for each register either $\mathsf{rda}(x) = \mathsf{rda}'(x)$ or $\mathsf{rda}(x) \sqsubseteq \mathsf{rda}'(x)$ holds. In the former case, indistinguishability is satisfied by assumption. In the latter case, $\mathsf{rda}'(x)$ must be *high* and, thus, indistinguishability is trivially satisfied. Therefore, we know for all $x \in \mathcal{X} \cup \mathcal{X}_{res}$ that if $\mathsf{rda}'(x) = \mathit{low}$ then $r(x) \sim_\beta r'(x)$. Hence, we can conclude $r \sim_{\beta,\mathsf{rda}'} r'$.

B Mapping of Dalvik Opcodes to Abstract Instructions

Table 1. Control flow instructions

Abstract Instruction	Concrete Dalvik Opcodes
`nop`	`nop`
`goto` n	`goto, goto/16, goto/32`
`if-test` v_a, v_b, n, rop	`if-eq, if-ne, if-lt, if-ge, if-gt, if-le`
`if-testz` v_a, n, rop	`if-eqz, if-nez, if-ltz, if-gez, if-gtz, if-lez`

Table 2. Arithmetic Instructions (1)

Abstract Instruction	Concrete Dalvik Opcodes
`move` v_a, v_b	`move, move/from16, move/16, move-object, move-object/from16, move-object/16`
`move-wide` v_a, v_b	`move-wide, move-wide/from16, move-wide/16`
`const` v_a, n	`const, const/4, const/16, const/high16`
`const-wide` v_a, n	`const-wide/16, const-wide/32, const-wide, const-wide/high16`
`cmp` v_a, v_b, v_c	`cmpl-float, cmpg-float`
`cmp-wide` v_a, v_b, v_c	`cmpl-double, cmpg-double, cmp-long`
`unop` v_a, v_b, uop	`neg-int, not-int, neg-float, int-to-float, float-to-int, int-to-byte, int-to-char, int-to-short`
`unop-wide` v_a, v_b, uop	`neg-long, not-long, neg-double, long-to-double, double-to-long`
`unop-wideS` v_a, v_b, uop	`long-to-int, long-to-float, double-to-int, double-to-float`
`unop-wideT` v_a, v_b, uop	`int-to-long, int-to-double, float-to-long, float-to-double`
`binop` v_a, v_b, v_c, bop	`add-int, sub-int, mul-int, div-int, rem-int, and-int, or-int, xor-int, shl-int, shr-int, ushr-int, add-float, sub-float, mul-float, div-float, rem-float`
`binop-wide` v_a, v_b, v_c, bop	`add-long, sub-long, mul-long, div-long, rem-long, and-long, or-long, xor-long, shl-long, shr-long, ushr-long, add-double, sub-double, mul-double, div-double, rem-double`

Table 3. Arithmetic Instructions (2)

Abstract Instruction	Concrete Dalvik Opcodes
`binop-2addr` v_a, v_b, bop	`add-int/2addr`, `sub-int/2addr`, `mul-int/2addr`, `div-int/2addr`, `rem-int/2addr`, `and-int/2addr`, `or-int/2addr`, `xor-int/2addr`, `shl-int/2addr`, `shr-int/2addr`, `ushr-int/2addr`, `add-float/2addr`, `sub-float/2addr`, `mul-float/2addr`, `div-float/2addr`, `rem-float/2addr`
`binop-2addr-wide` v_a, v_b, bop	`add-long/2addr`, `sub-long/2addr`, `mul-long/2addr`, `div-long/2addr`, `rem-long/2addr`, `and-long/2addr`, `or-long/2addr`, `xor-long/2addr`, `shl-long/2addr`, `shr-long/2addr`, `ushr-long/2addr`, `add-double/2addr`, `sub-double/2addr`, `mul-double/2addr`, `div-double/2addr`, `rem-double/2addr`
`binop-lit` v_a, v_b, n, bop	`add-int/lit16`, `rsub-int`, `mul-int/lit16`, `div-int/lit16`, `rem-int/lit16`, `and-int/lit16`, `or-int/lit16`, `xor-int/lit16`, `add-int/lit8`, `rsub-int/lit8`, `mul-int/lit8`, `div-int/lit8`, `rem-int/lit8`, `and-int/lit8`, `or-int/lit8`, `xor-int/lit8`, `shl-int/lit8`, `shr-int/lit8`, `ushr-int/lit8`

Table 4. Array-Related Instructions

Abstract Instruction	Concrete Dalvik Opcodes
`array-length` v_a, v_b	`array-length`
`new-array` v_a, v_b	`new-array`
`filled-new-array` $v_a, v_b, v_c, v_d, v_e, n$	`filled-new-array`
`filled-new-array-range` v_a, n	`filled-new-array/range`
`fill-array-data` $v_a, u_0, \ldots u_n$	`fill-array-data`
`aget` v_a, v_b, v_c	`aget`, `aget-object`, `aget-boolean`, `aget-byte`, `aget-char`, `aget-short`
`aget-wide` v_a, v_b, v_c	`aget-wide`
`aput` v_a, v_b, v_c	`aput`, `aput-object`, `aput-boolean`, `aput-byte`, `aput-char`, `aput-short`
`aput-wide` v_a, v_b, v_c	`aput-wide`

Table 5. Object-Related Instructions

Abstract Instruction	Concrete Dalvik Opcodes
`instance-of` v_a, v_b, cl	`instance-of`
`new-instance` v_a, cl	`new-instance`
`const-string` v_a, s	`const-string, const-string/jumbo`
`const-class` v_a, cl	`const-class`
`iget` v_a, v_b, fid	`iget, iget-object, iget-boolean, iget-byte, iget-char, iget-short`
`iget-wide` v_a, v_b, fid	`iget-wide`
`iput` v_a, v_b, fid	`iput, iput-object, iput-boolean, iput-byte, iput-char, iput-short`
`iput-wide` v_a, v_b, fid	`iput-wide`
`sget` v_a, fid	`sget, sget-object, sget-boolean, sget-byte, sget-char, sget-short`
`sget-wide` v_a, fid	`sget-wide`
`sput` v_a, fid	`sput, sput-object, sput-boolean, sput-byte, sput-char, sput-short`
`sput-wide` v_a, fid	`sput-wide`

Table 6. Method-Related Instructions

Abstract Instruction	Concrete Dalvik Opcodes
`invoke-virtual` $v_a, v_b, v_c, v_d, v_e, n, mid$	`invoke-virtual`
`invoke-super` $v_a, v_b, v_c, v_d, v_e, n, mid$	`invoke-super`
`invoke-direct` $v_a, v_b, v_c, v_d, v_e, n, mid$	`invoke-direct`
`invoke-interface` $v_a, v_b, v_c, v_d, v_e, n, mid$	`invoke-interface`
`invoke-static` $v_a, v_b, v_c, v_d, v_e, n, mid$	`invoke-static`
`invoke-virtual-range` v_a, n, mid	`invoke-virtual/range`
`invoke-super-range` v_a, n, mid	`invoke-super/range`
`invoke-direct-range` v_a, n, mid	`invoke-direct/range`
`invoke-interface-range` v_a, n, mid	`invoke-interface/range`
`invoke-static-range` v_a, n, mid	`invoke-static/range`
`move-result` v_a	`move-result, move-result-object`
`move-result-wide` v_a	`move-result-wide`
`return-void`	`return-void`
`return` v_a	`return, return-object`
`return-wide` v_a	`return-wide`

C Semantics of Further Instructions

$$\text{rMoveWide}\ \frac{m[pp] = \texttt{move-wide}\ v_a, v_b}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto r(v_b), v_{a+1} \mapsto r(v_{b+1})]\rangle}$$

$$\text{rConstWide}\ \frac{m[pp] = \texttt{const-wide}\ v_a, n}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{lower}(n), v_{a+1} \mapsto \mathsf{upper}(n)]\rangle}$$

$$\text{rCmpWide>}\ \frac{m[pp] = \texttt{cmp-wide}\ v_a, v_b, v_c \qquad (r(v_b) \bullet r(v_{b+1})) > (r(v_c) \bullet r(v_{c+1}))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto -1]\rangle}$$

$$\text{rCmpWide=}\ \frac{m[pp] = \texttt{cmp-wide}\ v_a, v_b, v_c \qquad (r(v_b) \bullet r(v_{b+1})) = (r(v_c) \bullet r(v_{c+1}))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto 0]\rangle}$$

$$\text{rCmpWide<}\ \frac{m[pp] = \texttt{cmp-wide}\ v_a, v_b, v_c \qquad (r(v_b) \bullet r(v_{b+1})) < (r(v_c) \bullet r(v_{c+1}))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto 1]\rangle}$$

$$\text{rUnopWide}\ \frac{m[pp] = \texttt{unop-wide}\ v_a, v_b, uop \qquad x = \underline{uop}(r(v_b) \bullet r(v_{b+1}))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{lower}(x), v_{a+1} \mapsto \mathsf{upper}(x)]\rangle}$$

$$\text{rUnopWideS}\ \frac{m[pp] = \texttt{unop-wideS}\ v_a, v_b, uop \qquad x = \underline{uop}(r(v_b) \bullet r(v_{b+1}))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto x]\rangle}$$

$$\text{rUnopWideT}\ \frac{m[pp] = \texttt{unop-wideT}\ v_a, v_b, uop \qquad x = \underline{uop}(r(v_b))}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{lower}(x), v_{a+1} \mapsto \mathsf{upper}(x)]\rangle}$$

$$\text{rBinopWide}\ \frac{\begin{array}{c} m[pp] = \texttt{binop-wide}\ v_a, v_b, v_c, bop \\ x = (r(v_b) \bullet r(v_{b+1}))\ \underline{bop}\ \ (r(v_c) \bullet r(v_{c+1})) \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{lower}(x), v_{a+1} \mapsto \mathsf{upper}(x)]\rangle}$$

$$\text{rBinopWide2addr}\ \frac{\begin{array}{c} m[pp] = \texttt{binop-wide-2addr}\ v_a, v_b, bop \\ x = (r(v_a) \bullet r(v_{a+1}))\ \underline{bop}\ \ (r(v_b) \bullet r(v_{b+1})) \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \mathsf{lower}(x), v_{a+1} \mapsto \mathsf{upper}(x)]\rangle}$$

Figure 14. Semantics of instructions for 64 bit values (1)

$$\text{rIgetWide}\frac{\begin{array}{cc} m[pp] = \texttt{iget-wide}\ v_a, v_b, \mathit{fid} & \mathit{fid} \in \mathit{dom}(\textsf{lookup-field}_P) \\ r(v_b) \in \mathit{dom}(h) & o = h(r(v_b)) \\ f = \textsf{lookup-field}_P(\mathit{fid}) & f \in \mathit{dom}(o.\textsf{fields}) \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \textsf{lower}(o.f), v_{a+1} \mapsto \textsf{upper}(o.f)]\rangle}$$

$$\text{rIputWide}\frac{\begin{array}{cc} m[pp] = \texttt{iput-wide}\ v_a, v_b, \mathit{fid} & \mathit{fid} \in \mathit{dom}(\textsf{lookup-field}_P) \\ r(v_b) \in \mathit{dom}(h) & o = h(r(v_b)) \\ f = \textsf{lookup-field}_P(\mathit{fid}) & f \in \mathit{dom}(o.\textsf{fields}) \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[r(v_b) \mapsto o[f \mapsto (r(v_a) \bullet r(v_{a+1}))]], pp+1, r\rangle}$$

$$\text{rSgetWide}\frac{\begin{array}{c} m[pp] = \texttt{sget-wide}\ v_a, \mathit{fid} \qquad \mathit{fid} \in \mathit{dom}(\textsf{nameToReference}) \\ l = \textsf{nameToReference}(\mathit{fid}) \qquad \mathit{fid} \in \mathit{dom}(\textsf{lookup-field}_P) \\ f = \textsf{lookup-field}_P(\mathit{fid}) \qquad f \in \mathit{dom}(h(l).\textsf{fields}) \qquad x = h(l).f \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \textsf{lower}(x), v_{a+1} \mapsto \textsf{upper}(x)]\rangle}$$

$$\text{rSputWide}\frac{\begin{array}{c} m[pp] = \texttt{sput-wide}\ v_a, \mathit{fid} \qquad \mathit{fid} \in \mathit{dom}(\textsf{nameToReference}) \\ l = \textsf{nameToReference}(\mathit{fid}) \qquad \mathit{fid} \in \mathit{dom}(\textsf{lookup-field}_P) \\ f = \textsf{lookup-field}_P(\mathit{fid}) \qquad x = h(l) \qquad f \in \mathit{dom}(x.\textsf{fields}) \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[l \mapsto x[f \mapsto (r(v_a) \bullet r(v_{a+1}))]], pp+1, r\rangle}$$

$$\text{rAgetWide}\frac{\begin{array}{c} m[pp] = \texttt{aget-wide}\ v_a, v_b, v_c \qquad r(v_b) \in \mathit{dom}(h) \qquad ar = h(r(v_b)) \\ x = ar[r(v_c)] \qquad 0 \leq r(v_c) < ar.\textsf{length} \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto \textsf{lower}(x), v_{a+1} \mapsto \textsf{upper}(x)]\rangle}$$

$$\text{rAputWide}\frac{\begin{array}{c} m[pp] = \texttt{aput-wide}\ v_a, v_b, v_c \qquad r(v_b) \in \mathit{dom}(h) \qquad ar = h(r(v_b)) \\ x = ar[r(v_c) \mapsto (r(v_a) \bullet r(v_{a+1}))] \qquad 0 \leq r(v_c) < ar.\textsf{length} \end{array}}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h[r(v_b) \mapsto x], pp+1, r\rangle}$$

$$\text{rReturnWide}\frac{m[pp] = \texttt{return-wide}\ v_a}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle r(v_a) \bullet r(v_{a+1}), h\rangle}$$

$$\text{rMoveRW}\frac{m[pp] = \texttt{move-result-wide}\ v_a}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto r(result_{lower}), v_{a+1} \mapsto r(result_{upper})]\rangle}$$

Figure 15. Semantics of instructions for 64 bit values (2)

$$\text{rCmp>}\frac{m[pp] = \texttt{cmp}\ v_a, v_b, v_c \qquad r(v_b) > r(v_c)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto -1]\rangle}$$

$$\text{rCmp=}\frac{m[pp] = \texttt{cmp}\ v_a, v_b, v_c \qquad r(v_b) = r(v_c)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto 0]\rangle}$$

$$\text{rCmp<}\frac{m[pp] = \texttt{cmp}\ v_a, v_b, v_c \qquad r(v_b) < r(v_c)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto 1]\rangle}$$

$$\text{rBinop2addr}\frac{m[pp] = \texttt{binop-2addr}\ v_a, v_b, bop \qquad x = r(v_a)\ \underline{bop}\ r(v_b)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto x]\rangle}$$

$$\text{rBinopLit}\frac{m[pp] = \texttt{binop-lit}\ v_a, v_b, n, bop \qquad x = r(v_b)\ \underline{bop}\ n}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r[v_a \mapsto x]\rangle}$$

$$\text{rIfTestzTrue}\frac{m[pp] = \texttt{if-testz}\ v_a, n, rop \qquad r(v_a)\ \underline{rop}\ 0}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+n, r\rangle}$$

$$\text{rIfTestzFalse}\frac{m[pp] = \texttt{if-testz}\ v_a, n, rop \qquad \neg(r(v_a)\ \underline{rop}\ 0)}{\langle h, pp, r\rangle \leadsto^{(0)}_{P,m} \langle h, pp+1, r\rangle}$$

$$\text{rIS}\frac{\begin{array}{c} m[pp] = \texttt{invoke-super}\ v_{k_0}, v_{k_1}, v_{k_2}, v_{k_3}, v_{k_4}, n, mid \\ (mid, h(r(v_{k_0})).\mathsf{class}) \in dom(\mathsf{lookup\text{-}super}_P) \\ m' = \mathsf{lookup\text{-}super}_P(mid, h(r(v_{k_0})).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_{k_0}), \ldots, r(v_{k_{n-1}})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h'\rangle \end{array}}{\langle h, pp, r\rangle \leadsto^{(n'+1)}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{rISR}\frac{\begin{array}{c} m[pp] = \texttt{invoke-super-range}\ v_k, n, mid \\ (mid, h(r(v_k)).\mathsf{class}) \in dom(\mathsf{lookup\text{-}super}_P) \\ m' = \mathsf{lookup\text{-}super}_P(mid, h(r(v_k)).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_k), \ldots r(v_{k+n-1})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h'\rangle \end{array}}{\langle h, pp, r\rangle \leadsto^{(n'+1)}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

Figure 16. Semantics of other instructions (1)

$$\text{rID}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-direct}\ v_{k_0}, v_{k_1}, v_{k_2}, v_{k_3}, v_{k_4}, n, mid \qquad r(v_{k_0}) \in dom(h) \\ (mid, h(r(v_{k_0})).\mathsf{class}) \in dom(\mathsf{lookup\text{-}direct}_P) \\ m' = \mathsf{lookup\text{-}direct}_P(mid, h(r(v_{k_0})).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_{k_0}), \ldots, r(v_{k_{n-1}})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h' \rangle \end{array}}{\langle h, pp, r\rangle \overset{(n'+1)}{\rightsquigarrow}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{rIDR}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-direct-range}\ v_k, n, mid \\ (mid, h(r(v_k)).\mathsf{class}) \in dom(\mathsf{lookup\text{-}direct}_P) \qquad r(v_k) \in dom(h) \\ m' = \mathsf{lookup\text{-}direct}_P(mid, h(r(v_k)).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_k), \ldots r(v_{k+n-1})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h' \rangle \end{array}}{\langle h, pp, r\rangle \overset{(n'+1)}{\rightsquigarrow}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{rII}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-interface}\ v_{k_0}, v_{k_1}, v_{k_2}, v_{k_3}, v_{k_4}, n, mid \qquad r(v_{k_0}) \in dom(h) \\ (mid, h(r(v_{k_0})).\mathsf{class}) \in dom(\mathsf{lookup\text{-}virtual}_P) \\ m' = \mathsf{lookup\text{-}virtual}_P(mid, h(r(v_{k_0})).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_{k_0}), \ldots, r(v_{k_{n-1}})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h' \rangle \end{array}}{\langle h, pp, r\rangle \overset{(n'+1)}{\rightsquigarrow}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{rIIR}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-interface-range}\ v_k, n, mid \qquad r(v_k) \in dom(h) \\ (mid, h(r(v_k)).\mathsf{class}) \in dom(\mathsf{lookup\text{-}virtual}_P) \\ m' = \mathsf{lookup\text{-}virtual}_P(mid, h(r(v_k)).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_k), \ldots r(v_{k+n-1})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h' \rangle \end{array}}{\langle h, pp, r\rangle \overset{(n'+1)}{\rightsquigarrow}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{rISt}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-static}\ v_{k_0}, v_{k_1}, v_{k_2}, v_{k_3}, v_{k_4}, n, mid \\ mid \in dom(\mathsf{lookup\text{-}static}_P) \qquad m' = \mathsf{lookup\text{-}static}_P(mid) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_{k_0}), \ldots, r(v_{k_{n-1}})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h' \rangle \end{array}}{\langle h, pp, r\rangle \overset{(n'+1)}{\rightsquigarrow}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

$$\text{rIV}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-virtual}\ v_{k_0}, v_{k_1}, v_{k_2}, v_{k_3}, v_{k_4}, n, mid \qquad r(v_{k_0}) \in dom(h) \\ (mid, h(r(v_{k_0})).\mathsf{class}) \in dom(\mathsf{lookup\text{-}virtual}_P) \\ m' = \mathsf{lookup\text{-}virtual}_P(mid, h(r(v_{k_0})).\mathsf{class}) \\ \langle h, 0, \mathsf{defaultRegisters}([r(v_{k_0}), \ldots, r(v_{k_{n-1}})])\rangle \Downarrow^{(n')}_{P,m'} \langle u, h' \rangle \end{array}}{\langle h, pp, r\rangle \overset{(n'+1)}{\rightsquigarrow}_{P,m} \langle h', pp+1, r[result_{lower} \mapsto \mathsf{lower}(u), result_{upper} \mapsto \mathsf{upper}(u)]\rangle}$$

Figure 17. Semantics of other instructions (2)

$$\text{rFilledNewArray}\ \frac{\begin{array}{c} m[pp] = \texttt{filled-new-array}\ v_{k_0}, v_{k_1}, v_{k_2}, v_{k_3}, v_{k_4}, n \qquad 0 \le n \\ h \in dom(\mathsf{nextFreeLocation}) \qquad\qquad l = \mathsf{nextFreeLocation}(h) \\ x = \mathsf{defaultArray}(n) \qquad ar = x[0 \mapsto r(v_{k_0}), \ldots, n-1 \mapsto r(v_{k_{n-1}})] \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[l \mapsto ar], pp+1, r[result_{lower} \mapsto l]\rangle}$$

$$\text{rFillArrayData}\ \frac{\begin{array}{c} m[pp] = \texttt{fill-array-data}\ v_a, u_0, \ldots, u_n \\ r(v_a) \in dom(h) \qquad ar = h(r(v_a)) \qquad 0 \le n < ar.\mathsf{length} \\ x = ar[0 \mapsto u_0, \ldots, n \mapsto u_n] \text{ for all } 0 \le i \le n \end{array}}{\langle h, pp, r\rangle \rightsquigarrow^{(0)}_{P,m} \langle h[r(v_a) \mapsto x], pp+1, r\rangle}$$

Figure 18. Semantics of other instructions (3)

D Typing Rules of Further Instructions

$$\text{tMoveW}\frac{m[pp] = \texttt{move-wide } v_a, v_b \qquad t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_{b+1}) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

$$\text{tCW}\frac{m[pp] = \texttt{const-wide } v_a, n}{m, \ldots, ret, se(pp) \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto se(pp), v_{a+1} \mapsto se(pp)]}$$

$$\text{tCmpW}\frac{\begin{array}{c} m[pp] = \texttt{cmp-wide } v_a, v_b, v_c \\ t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_{b+1}) \sqcup \mathsf{rda}(v_c) \sqcup \mathsf{rda}(v_{c+1}) \sqcup se(pp) \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

$$\text{tUnopW}\frac{m[pp] = \texttt{unop-wide } v_a, v_b, uop \qquad t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_{b+1}) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

$$\text{tBinopW}\frac{\begin{array}{c} m[pp] = \texttt{binop-wide } v_a, v_b, v_c, bop \\ t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_{b+1}) \sqcup \mathsf{rda}(v_c) \sqcup \mathsf{rda}(v_{c+1}) \sqcup se(pp) \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

$$\text{tBinop2W}\frac{\begin{array}{c} m[pp] = \texttt{binop-2addr-wide } v_a, v_b, bop \\ t = \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_{a+1}) \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_{b+1}) \sqcup se(pp) \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

$$\text{tIgetWide}\frac{\begin{array}{c} m[pp] = \texttt{iget-wide } v_a, v_b, fid \qquad \mathsf{fda}(fid) = st \\ t = \mathsf{rda}(v_b) \sqcup st \sqcup se(pp) \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

$$\text{tIputWide}\frac{\begin{array}{c} m[pp] = \texttt{iput-wide } v_a, v_b, fid \qquad \mathsf{fda}(fid) = st \\ \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_{a+1}) \sqcup \mathsf{rda}(v_b) \sqcup se(pp) \sqsubseteq st \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

$$\text{tSgetWide}\frac{m[pp] = \texttt{sget-wide } v_a, fid \qquad \mathsf{fda}(fid) = st \qquad t = st \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

$$\text{tSputWide}\frac{\begin{array}{c} m[pp] = \texttt{sput-wide } v_a, fid \qquad \mathsf{fda}(fid) = st \\ \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_{a+1}) \sqcup se(pp) \sqsubseteq st \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

Figure 19. Security typing rules for instructions for 64 bit values (1)

$$\text{tAgetW}\ \frac{m[pp] = \texttt{aget-wide}\ \ v_a, v_b, v_c \qquad t = se(pp) \sqcup \mathsf{ada} \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

$$\text{tAputW}\ \frac{\begin{array}{c} m[pp] = \texttt{aput-wide}\ \ v_a, v_b, v_c \\ \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_{a+1}) \sqcup se(pp) \sqcup \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqsubseteq \mathsf{ada} \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

$$\text{tMoveResW}\ \frac{\begin{array}{c} m[pp] = \texttt{move-result-wide}\ \ v_a \\ t = \mathsf{rda}(result_{lower}) \sqcup \mathsf{rda}(result_{upper}) \sqcup se(pp) \end{array}}{m, \dots \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t, v_{a+1} \mapsto t]}$$

$$\text{tReturnW}\ \frac{m[pp] = \texttt{return-wide}\ \ v_a \qquad se(pp) \sqcup \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_{a+1}) \sqsubseteq ret}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

Figure 20. Security typing rules for instructions for 64 bit values (2)

$$\text{tCmp}\ \frac{m[pp] = \texttt{cmp}\ \ v_a, v_b, v_c \qquad t = \mathsf{rda}(v_b) \sqcup \mathsf{rda}(v_c) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

$$\text{tBinop2}\ \frac{m[pp] = \texttt{binop-2addr}\ \ v_a, v_b, bop \qquad t = \mathsf{rda}(v_a) \sqcup \mathsf{rda}(v_b) \sqcup se(pp)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto t]}$$

$$\text{tBinopL}\ \frac{m[pp] = \texttt{binop-lit}\ \ v_a, v_b, n, bop}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}[v_a \mapsto \mathsf{rda}(v_b) \sqcup se(pp)]}$$

$$\text{tIfTestz}\ \frac{m[pp] = \texttt{if-testz}\ \ v_a, n, rop \qquad \forall j \in region_m(pp).\mathsf{rda}(v_a) \sqsubseteq se(j)}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

$$\text{tFNA}\ \frac{\begin{array}{c} m[pp] = \texttt{filled-new-array}\ \ v_a, v_b, v_c, v_d, v_e, n \\ x = [v_a, v_b, v_c, v_d, v_e] \qquad \bigsqcup_{i=1}^{n} \mathsf{rda}(x[i]) \sqsubseteq \mathsf{ada} \end{array}}{m, \dots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto se(pp), result_{upper} \mapsto se(pp)]}$$

$$\text{tFillAData}\ \frac{\begin{array}{c} m[pp] = \texttt{fill-array-data}\ \ v_a, u_0, \dots u_n \\ \mathsf{rda}(v_a) \sqcup se(pp) \sqsubseteq \mathsf{ada} \end{array}}{m, region_m, \mathsf{mda}, \mathsf{fda}, \mathsf{ada}, ret, se \vdash pp : \mathsf{rda} \to \mathsf{rda}}$$

Figure 21. Security typing rules for other instructions (1)

$$\text{tI}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-*}\ v_a, v_b, v_c, v_d, v_e, n, mid \qquad x = [v_a, \ldots, v_e] \\ (mid, [\mathsf{rda}(x[1]), \ldots, \mathsf{rda}(x[n])], st) \in \mathsf{mda} \\ \mathsf{rda}(v_a) = low \qquad\qquad\qquad se(pp) = low \end{array}}{m, \cdots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto st, result_{upper} \mapsto st]}$$

$$\text{tIR}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-*-range}\ v_a, n, mid \\ (mid, [\mathsf{rda}(v_a), \ldots, \mathsf{rda}(v_{a+n-1})], st) \in \mathsf{mda} \\ \mathsf{rda}(v_a) = low \qquad\qquad se(pp) = low \end{array}}{m, \cdots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto st, result_{upper} \mapsto st]}$$

$$\text{tIS}\ \frac{\begin{array}{c} m[pp] = \texttt{invoke-static}\ v_a, v_b, v_c, v_d, v_e, n, mid \\ x = [v_a, \ldots, v_e] \qquad (mid, [\mathsf{rda}(x[1]), \ldots, \mathsf{rda}(x[n])], st) \in \mathsf{mda} \\ se(pp) = low \end{array}}{m, \cdots \vdash pp : \mathsf{rda} \to \mathsf{rda}[result_{lower} \mapsto st, result_{upper} \mapsto st]}$$

Figure 22. Security typing rules for other instructions (2)

The rules `invoke-*` and `invoke-*-range` apply to all non-static invoke instructions.