Bachelor Thesis
Bachelor of Science Informatik

Confidentiality for Multithreaded Dalvik Programs by SIFUM Security

Pascal Wittmann

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Modeling and Analysis of Information Systems

Prüfer: Prof. Dr.-Ing. Heiko Mantel
Betreuer: M.Sc. Matthias Perner

Abgabetermin: 18. März 2013

Erklärung

Hiermit versichere ich gemäß der Allgemeinen Prüfungsbestimmungen der Technischen Universität Darmstadt (APB) §23 (7), die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	(Pascal Wittmann)

Abstract

Non-interference properties on programs describe the absence illicit information flows. We focus on non-interference properties for the byte-code language of the Dalvik virtual machine (DVM). The DVM executes programs on Android, a popular operating system for mobile phones. Most programs for Android make extensive use of multi-threading to ensure smooth responses to user interactions. We introduce an abstract transition system that supports multi-threading and instantiate it with a subset of the DVM. To capture the notion of secure/legal information flow for this abstract transition system, we introduce a bisimulation based non-interference property that exploits the intended data usage of concurrently executed threads.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Contributions . . . 4
- 1.3 Structure . . . 4
- 1.4 Notational Conventions . . . 5

2 Transition System **6**
- 2.1 The Execution Model of the DVM . . . 6
- 2.2 SIFUM Security in Context of the DVM . . . 7
- 2.3 Abstract Transition System . . . 8
  - 2.3.1 Program . . . 8
  - 2.3.2 Call Stack . . . 10
  - 2.3.3 Heap . . . 11
  - 2.3.4 Assumptions and Guarantees . . . 12
  - 2.3.5 Configurations . . . 12
  - 2.3.6 Transition Relation . . . 13

3 SIFUM Security **16**
- 3.1 Security Policy . . . 16
- 3.2 Attacker Model . . . 16
- 3.3 Indistinguishability Relations . . . 17
- 3.4 Register Domain Assignments . . . 21
- 3.5 Security Property . . . 24
- 3.6 Compositionality . . . 28

4 Instances of the Transition System **32**
- 4.1 Sequential Subset of Dalvik . . . 32
- 4.2 Static Thread Structure . . . 42
- 4.3 Dynamic Thread Creation . . . 44

5 Summary **48**
- 5.1 Conclusion . . . 48
- 5.2 Related Work . . . 48
- 5.3 Future Work . . . 49

A Definition Graphs **50**

Bibliography **52**

Chapter 1

Introduction

This chapter motivates the thesis, summarizes the contributions and gives an overview on how the thesis is structured. The last section of this chapter introduces notational coventions that are used through this thesis.

1.1 Motivation

Smartphones have arrived in our daily lives. They help us organizing and planing many aspects of our life, for example by reminding us of our appointments, leading us the way to our leisure location and helping us to stay in touch with our friends via social networks.

These features are provided by programs that can be installed on the smartphone. To make use out of these programs we need to entrust them with our private data. Moreover smartphones are by their nature tightly connected to other devices and networks like the internet. This raises the question whether a program ensures the confidentiality of the entrusted data by *not* leaking them to untrusted parties.

Traditional security mechanisms like access control and encryption are not enough to protect the confidentiality of the entrusted data, because they only secure the access to the data (see [SM03]). After a successful and legitimate access, confidential data (or information about confidential data) can be leaked – accidentally or maliciously – trough the data and control flow of the program, to publicly visible outputs. An attacker can then try to conclude private information by looking at the public outputs. The security mechanism that Android – the most popular operating system for smartphones (see [PvdM12]) – provides to ensure the confidentiality of the entrusted data is an access control mechanism (see [DDSW10]). Thus the confidentiality of the entrusted data is currently not ensured on Android if data is accessed.

Leaks caused by the data and information flow can be described by security properties that classify the data and information flows within programs as legal and illegal. The absence of illegal data and information flows is described with non-interference-like [GM82] properties, which require that public outputs of a program are independent from private inputs. Due to this independence an attacker is not able to conclude any private information by looking at the public outputs.

Current security properties (e.g. [Web12]) for the Dalvik virtual machine (DVM), the software that executes programs on Android, are not adequate for multithreaded programs, although most Android programs use multiple threads (see [WK12]). When constructing a security property that is adequate for multithreaded programs a desirable characteristic is compositionality, i.e. the parallel execution of secure (w.r.t. the security property) threads is secure again.

To achieve parallel compositionality many existing security properties (e.g. [SS00] and [ZM03]) make the worst case assumption that the environment might access every variable at any point in time. This over-approximation classifies intuitively secure and useful programs as insecure. The security property introduced in [MSS11] is more precise, because it makes the intended usage of variables explicit. Thus it does not need to look at all variable accesses as it can be sure that some will not happen. The variable usage pattern is made explicit by allowing each thread to make assumptions on how variables are accessed by other threads and by giving guarantees on how certain variables are access by the thread itself. If all assumptions are matched by the corresponding guarantees and the guarantees hold, it is permissible to temporarily store confidential information in public variables and it is possible to temporarily be sure that the values of certain variables do not change during execution.

The purpose of this thesis is to adapt the SIFUM Security property from [MSS11] to an abstract transition system that is capable of simulating the DVM and to argue for concrete instantiations of the transition system that the security property is adequate.

1.2 Contributions

In this thesis the following contributions were achieved.

- A transition system that is parametric in the instructions and dimensioned for (possibilistic) multithreaded machines with monitors as synchronization concept, such as the DVM.
- Instantiations of the transition system that model parts of the DVM. The covered parts are the object system, methods, dynamic thread creation and synchronization via monitors. Not covered are the class hierarchy, arrays and exceptions.
- An information flow security property for the transition system that takes advantage out of the intended usage of the shared memory to achieve precision and compositionality. This security property is based on [MSS11].

1.3 Structure

Chapter 2 gives a short overview of the DVM, describes how the assumption-guarantee style reasoning fits into this context of the DVM and defines an abstract transition system (ATS) that provides the structure to instantiate (parts of) the DVM.

In Chapter 3 we define the security policy, describe the capabilities of the attacker and introduce the new security property in a step-wise manner. Fist we define indistinguishability relations that capture the capabilities of the attacker, then we deal with the effects of concurrently executed threads on the shared memory and finally we define a strong bisimulation that captures secure information flow from the perspective of a single thread. The last section of this chapter sets the stage for proving parallel compositionality of the security property.

Chapter 4 defines three instantiations of the ATS. A sequential instance, an instance with a static thread structure and an instance with dynamic thread creation. For each instance small example programs illustrate how the security property relates to the instances.

Appendix A contains three graphs that show the dependencies between the definitions.

1.4 Notational Conventions

In this section we introduce notational conventions that are used in this thesis.

The set $\mathbb{N}$ denotes the set of natural numbers *including* 0.

$A \to B$ denotes the set of total functions from A to B.

$A \rightharpoonup B$ denotes the set of partial function from A to B.

$\mathcal{P}(X) := \{U : U \subseteq X\}$ denotes the powerset of X.

$f[a \mapsto b]$ denotes the function $f'(x) := \begin{cases} b & x = a \\ f(x) & otherwise \end{cases}$.

$f \restriction A$ denotes function to the graph $\{(x, f(x)) : x \in A\}$.

$(x :: xs) \in A^{n+1}$ with $x \in A$ and $xs \in A^n$ denotes list construction.

Empty lists are denoted as $\langle\rangle$.

Chapter 2

Transition System

2.1 The Execution Model of the DVM

In this section we identify the information about the Dalvik virtual machine (DVM) that are needed to define an abstract transition system (ATS) as an execution model for the DVM.

The DVM is a object-oriented, multi-threaded virtual register machine that executes byte-code. We model multi-threading, synchronization, the object system and methods. To keep the ATS simple we do not consider arrays, the class hierarchy, static fields and static methods.

At first we introduce the notions that are needed to model the object-oriented aspect of the DVM. A *class* is a template for a data-structure together with a collection of methods and is identified by a class name. A data-structure is a collection of fields, whereas a *field* is an identifier for a data container. In the context of a class fields do not store a value, because we do not consider static fields. An *object* is the instantiation of a class that stores data in the field. Objects are identified by a location on the heap, which is introduced in the following. A *method* is a ordered and enumerated collection of byte-code instructions that is identified within a class by a method name and by the data types of the method parameters. Every instruction within a method can be identified with a *program point*.

The DVM has two kinds of memories. One kind is the *heap* which stores the objects. The other kind of memory are register sets. A *register set* consists of values that are bound to a register identifier. Most computations are done on register sets, whereas the heap is used as a place to store information. Therefore the values on the heap are moved into a register set to do computations.

The *programs* that are executed by the DVM are collections of classes, fields and methods. The execution of a Dalvik program starts in the method `activity.onCreate` which must be provided by every program (see [Pro]). In the following we describe the single-threaded execution of a program.

The current state of execution consists of a *call stack* and a heap. An *element* of the call stack (also called *stack frame*) consists of a reference to the current instruction, the current method name and a register set. `this` is the reference to the object "in" which the execution currently takes place and is saved at the first register of a register set. On a method call a new call stack element consisting of the reference to the first instruction of the called method, the name of the called method and a fresh register set – which is filled with the values of the methods parameters – is put on the stack. On the exit of a method call the top element of the call stack is removed and the return value of the called method is saved at a special register identifier in the register set of the caller.

The reference to the current instruction is used to know which instruction is currently executed and which one will be executed next. Therefore the program point of the current instruction is also used to continue the execution after a method call. The method name is needed to know which method is currently executed. Every method call is provided with a fresh register set, this allows the previously described parameter and return value passing. The register sets are accessible only within the current method call, with the exception of the register for the return value.

The execution of a multi-threaded program has multiple call stacks, each represents the execution state of a *thread*. If a thread is spawned a new call stack is created and if a thread finishes execution the corresponding call stack is removed. We assume that register sets are not shared between the threads, therefore the only shared memory is the heap. We have to assume this, since we were not able to find an official source. However our assumption is assured by a post [Dev] on the mailing lists of the android project. With this assumption the purpose of registers becomes clear. They are used as "local variables" for which the thread can be sure that they are not modified by concurrently executed threads.

To coordinate the access to the heap, the DVM provides the synchronization concept monitor. A *monitor* provides mutually exclusive access to special parts of methods in the context of an object. The special parts of methods are the *monitor regions* and are bounded by the `monitor-enter` and `monitor-exit` instructions.

A thread acquires a monitor for an object by reaching a monitor region. If no other thread currently owns the monitor for this object, the thread can enter the monitor region and obtain the monitor for this object. If the monitor is currently owned by an other thread, the acquiring thread has to wait until the monitor is released. Threads release monitors if they leave the monitor region of the corresponding monitor. If they have entered the monitor region multiple times while owning the monitor, they need to leave the monitor region as often as they have entered it to release the monitor.

2.2 SIFUM Security in Context of the DVM

The assumption-guarantee style reasoning of SIFUM Security exploits the intended pattern in which threads access variables. Each thread can make explicit if it expects other threads to not read or not write certain shared variables and if it guarantees other threads, that it will not read or write a certain shared variable. These assumptions and guarantees (in the following called *modes*) can be made for specific regions in the code.

The current state of the assumptions and guarantees during the execution of a thread are represented as a *mode state*. A mode state assigns each mode a set of shared variables for which the mode is currently active. The mode states are solely used for the security analysis and will have no other effects on the number of execution steps.

As argued before the only memory in the DVM that is shared between threads is the heap. Thus the only kind of variables that are shared are fields.

2.3 Abstract Transition System

In this section we formally define an abstract transition system (ATS) that provides the basis to instantiate it with virtual register machines, like the DVM that execute byte-code and include features like object-orientation, multi-threading and synchronization via monitors.

The ATS consists of two main parts, the configurations and the the transition relation. We will first define the configurations out of small parts, which correspond to the relevant entities from Section 2.1 and Section 2.2. These parts are then step-wise composed to the final notion of configurations. Secondly we define a transition relation on the configurations, that is parametric in a transition relation for the semantics of the instructions.

2.3.1 Program

The first intermediate goal is to define the notion of a program.

Byte-code languages, like Dalvik, are unstructured and allow arbitrary jumps in the code. Therefore we need a concept that allows us to point to a specific instruction (e.g. to describe the destination of a jump) and to pick an entry point, i.e. the first instruction of a method that shall be executed.

Definition 2.1. *The set of program points is* $\mathcal{PP} = \mathbb{N} \setminus \{0\}$ *with a distinguished entry point* 1.

Program points enumerate all instructions in a method. As we cannot predict how many instructions a method has, $\mathcal{PP}$ is big enough to enumerate all instructions in arbitrary methods.

We assume that the program points are ordered in a way, such that the syntactic successor of an instruction at program point i has the program point $i + 1$.

We assume that the entry point of a method is always enumerated with 1. This is adequate since methods have only one entry point.

Class identifiers are used to create new objects of a specific class and to check if an object is an instance of a certain class. Classes have field identifiers that are used to access the values of fields.

Definition 2.2. *The set of unique class identifiers* $\mathcal{C}$ *contains all class names in a program and the set of unique field identifiers* $\mathcal{F}$ *contains all field names in a program.*

We require that field identifiers are unique within a program. This requirement can easily be fulfilled by e.g. using the class identifiers as a prefix in the field identifiers. In addition we assume, that a program will access a field of an object only if the field belongs to the corresponding class. These requirements allow us to keep the relation between classes and fields in most cases implicit, as no name clashes can occur and no invalid accesses can happen. Nevertheless we will use the not further specified function $fields : \mathcal{C} \rightarrow \mathcal{F}$ that returns the fields of a class, to be able to talk about possible memory changes (i.e. cancel out changes that are not possible due to invalid field accesses).

In the DVM a method is uniquely identified by its name, its corresponding class identifier and its parameter types, and consists of a ordered set of byte-code instructions. Furthermore all instructions of a program are within a method.

Definition 2.3. *The set of methods is $\mathcal{M} = \mathcal{M}_{\text{id}} \to (\mathcal{PP} \rightharpoonup \mathcal{I})$, where $\mathcal{I}$ is a not further specified set of instructions and $\mathcal{M}_{\text{id}}$ is a not further specified set of method identifiers.*

The set $\mathcal{M}_{\text{id}}$ models the *method identifiers* of a program. To allow different criteria for the uniqueness of a method name, we assume that it must be possible to build a unique identifier for every method (i.e. no method overloading and no two classes have methods with the same identifiers). Assuming that every method is called in the correct context and due to the uniqueness of method identifiers we leave the relation between classes and methods implicit.

The total function from method identifiers to instruction sets encodes the uniqueness constraint of the method identifier. First, there is no method without instructions (totality) and second, there is at most one instruction set for a method identifier (functionality).

The *instruction set* of a method is a partial function that maps program points to instructions. It is a partial function because the set of program points $\mathcal{PP}$ is infinite whereas the number of instructions within a method (as a syntactic construct) is finite.

As described in Section 2.1 most computations in the DVM are done on register sets rather than on the heap. To speak about computations on those register sets, we introduce register identifiers to refer to registers. In addition to general purpose register identifiers there is a special register identifier to access the return value of a method call.

Definition 2.4. *The set of register identifiers is $\mathcal{R} = \{v_i : i \in \mathbb{N}\} \cup \{v_{res}\}$. The location of the* `this` *object is always the register identifier v_0.*

The register identifiers v_i are used for standard computations and to provide parameters on method calls and return values. The register identifier v_{res} is only used to store the return value of a method call. This usage pattern is however not enforced by the ATS. The symbols for the register identifiers are borrowed from [Pro07].

Now we combine these definitions to define the notion of a program. Intuitively a program is a collection of classes that contain fields and methods. The methods of a program consist of a sequence of byte-code instructions.

Definition 2.5. *A program P is a triple $(C, F, M) \in \mathcal{C} \times \mathcal{F} \times \mathcal{M}$.*

A program is an instantiation of the set of class identifiers $\mathcal{C}$, the set of field identifiers $\mathcal{F}$ and of the set of methods $\mathcal{M}$. All these sets can be generated completely from the source code of a given program. Thus a program in the sense of the above definition is just a syntactic construct without semantics.

To illustrate the definition of programs, we define the following example program.

Example 1. *We define the example program $E = (C, F, M)$, where*

$$\begin{aligned}
C &= \{c\}\\
F &= \{h, l\}\\
\mathcal{M}_{\text{id}} &= \{m_1, m_2\}\\
\mathcal{I} &= \{\texttt{new}\ r\ c,\ \texttt{if}\ r\ f\ i,\ \texttt{put}\ r\ z\ f,\ \texttt{nop} : f \in \mathcal{F}, r \in \mathcal{R}, c \in C, i \in \mathbb{N}, z \in \mathbb{Z}\}\\
M &= \left\{ \left(m_1, i \mapsto \begin{cases} \texttt{new}\ v_0\ c & \\ \texttt{if}\ v_0\ h\ 3 & i = 1 \\ \texttt{nop} & i = 2 \\ \texttt{put}\ 1\ \ v_0\ l & i = 3 \end{cases} \right), \left(m_2, i \mapsto \begin{cases} \texttt{nop} & i = 1 \\ \texttt{put}\ 0\ v_0\ l & i = 2 \end{cases} \right) \right\}
\end{aligned}$$

This program consists of two methods m_1 and m_2, two field identifier h and l and a class identifier c. An intuitive semantics for the methods could be the following. The first method creates an object of class c and saves the reference to it in register v_0. Then method m_1 writes the number 1 into the field l of the created object and delays this write operation for one execution step if the value of the field h of the created object satisfies some condition. The second method m_2 is scheduled after method m_1 – therefore the instance of c already exists – and writes the number 0 into the field l and delays this operation always for one execution step.

2.3.2 Call Stack

The next intermediate goal is to define the notion of a call stack.

Locations are used to identify the elements that are saved on the heap, they correspond to the physical addresses of the heap.

Definition 2.6. *The set of locations is $\mathcal{L} = \{l_i : i \in \mathbb{N}\}$.*

The set $\mathcal{L}$ contains all locations on the heap. We assume that we have no special locations and no restriction on the number of locations. Due to the infinite supply of locations, we can never run out of memory.

The DVM uses different kinds of values to do computations and to refer to values in the memory. Furthermore there is the value *null* to describe not yet instantiated objects.

Definition 2.7. *The set of values is $\mathcal{V} = \mathcal{L} \cup \mathbb{V} \cup \{null\}$, where $\mathbb{V}$ is not further specified, $\mathcal{L} \cap \mathbb{V} = \emptyset$ and $null \notin \mathbb{V}$.*

The set $\mathcal{V}$ is used to model all values that can be saved in registers and fields. A value can either be a location on the heap, an element of the not further specified set $\mathbb{V}$ or *null*. The set $\mathbb{V}$ depends on the instantiation of the ATS and is intended to be used for primitive data types. The value *null* is used to describe objects that are not yet instantiated.

A call stack is responsible for the execution of methods, the storage of register sets of methods and for the bookkeeping of method calls.

Definition 2.8. *The set of call stacks is $\mathcal{CS} = (\mathcal{PP} \times \mathcal{M}_{\text{id}} \times (\mathcal{R} \rightharpoonup \mathcal{V}))^*$. A call stack with n elements is denoted as $\langle f_1, \ldots, f_n \rangle$ and the empty call stack is denoted as $\langle\rangle$.*

An element of the call stack (also referred to as *stack frame*) is a triple consisting of a program point, a method identifier and a register set. A register set is a partial function that maps register identifiers to values. This function is defined on all register identifiers that are initialized with some value and undefined on uninitialized register identifiers. This models the behavior of the DVM adequately since there is no general default value for register, that could be used to initialize all register on method calls (i.e. making the function total).

The behavior of the call stack is modeled as a Kleene closure. On a method call a new stack element is appended at the top of the current stack. This element contains the information that are provided by the method call. On the termination of a method the top element is removed. If the call stack is empty the execution has terminated. This simplification has the consequence, that no return values can be passed if the call stack contains only one element.

The following example shows an call stack that has a call depth of two.

Example 2. *Let* $m_1, m_2 \in \mathcal{M}_{\text{id}}$ *and* $rs, rs' \in (\mathcal{R} \rightharpoonup \mathcal{V})$ *then*

$$(1, m_2, rs) :: (34, m_1, rs') :: \langle\rangle \in \mathcal{CS}$$

Method m_2 was just called (since its current execution is the initial program point) and method m_1 will continue with instruction at program point 34 after method m_2 has returned.

2.3.3 Heap

In the DVM objects are saved on the heap. These objects consist of a class identifier (e.g. to check if an object is an instance of a certain class) and of a field assignment. The heap is a dynamically allocated memory.

Definition 2.9. *The set of heaps is* $\mathcal{H} = \mathcal{L} \rightharpoonup (\mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V}))$.

The heap is modeled as a partial function that is only defined at a location $l \in \mathcal{L}$ if an object is stored at location l to enable dynamic allocation. An *heap element* (i.e. an object) consists of a class name together with a function that maps field identifier to values. We map only field identifiers to values if the field actually belongs to the class and is initialized. If a field does not belong to the class or is uninitialized the function is undefined. Therefore the function is partial.

The functions π_1 and π_2 are projections for the class part and the fields part of a heap element.

The following example shows an heap, where the fields l_c and h_c belong to the class c and the field $f_{c'}$ belongs to the class c'. The object at location l_1 has the uninitialized field h_c. The field assignments are given by the graphs of the partial functions.

Example 3. *Let* $l_c, h_c, f_{c'} \in \mathcal{F}$ *and* $c, c' \in \mathcal{C}$ *then* $h \in \mathcal{H}$ *where*

$$h : l \mapsto \begin{cases} (c, \{(l_c, 2), (h_c, 3)\}) & l = l_0 \\ (c, \{(l_c, 5)\}) & l = l_1 \\ (c', \{(f_{c'}, 7)\}) & l = l_5 \end{cases}$$

2.3.4 Assumptions and Guarantees

In this subsection we model the assumptions and guarantees of SIFUM Security which was introduced in [MSS11].

The modes represent the assumptions and guarantees threads can make about the shared variables in programs. In our model the heap is the only shared memory, therefore the fields are the only variables for which assumptions and guarantees make sense. A thread can assume that a field is not read (*asm-noread*) or written (*asm-nowrite*) by other threads. As the counterpart of the assumptions a thread can guarantee that it does not read (*guar-noread*) or write (*guar-nowrite*) a certain field.

Definition 2.10. *The set of modes is*

$$\mathit{Mod} = \{\mathit{asm\text{-}noread}, \mathit{asm\text{-}nowrite}, \mathit{guar\text{-}noread}, \mathit{guar\text{-}nowrite}\}.$$

The set *Mod* contains all possible modes, where each mode represents the possible assumptions and guarantees. The relation between fields and modes is called *mode state*. A mode state is intuitively a snapshot of all assumptions and guarantees at a point in the execution of a thread.

Definition 2.11. *The set of mode states is* $\mathit{Mds} = \mathit{Mod} \to \mathcal{P}(\mathcal{F})$.

A mode state is a function that associates to every mode $m \in \mathit{Mod}$ a set of field identifiers. Since field identifiers are unique within a program, we do not need the class identifier. In this thesis the mode states are not object sensitive, i.e. if an assumption or guarantee is made for a field $f \in \mathcal{F}$ then this assumption or guarantee is made for all class instances to which f belongs to.

The initial mode state mds_0 is defined as $\mathit{mds}_0(m) = \emptyset$ for all $m \in \mathit{Mod}$.

2.3.5 Configurations

Now we can define the configurations of the ATS based on the preceding definitions.

A local configuration represents a snapshot of the transition system from the perspective of a single thread. A thread keeps track of its execution state, of the monitors it currently owns and of its current mode states. In addition a thread can access the heap, because the heap is shared between all threads.

Definition 2.12. *The set of local configurations is*

$$\mathit{Conf}_l = \mathcal{CS} \times (\mathcal{L} \to \mathbb{N}) \times \mathit{Mds} \times \mathcal{H} \times \mathrm{P}\,.$$

The first part of the local configuration is a call stack, which keeps track of the execution state. The thread has terminated if the call stack is empty. The second part models the owned monitors. A thread obtains a monitor for an object by reaching the corresponding monitor region. Since we identify objects by locations on the heap and an object can have at most one monitor, we model the owned monitors as a total function from locations to natural numbers. Intuitively the function has the following semantics. If a location is mapped to 0 then the thread does not own the monitor for the object at this location. If the location is mapped to a value $n > 0$ then the thread owns the monitor for the object at this location and has entered the monitor region n times. For simplicity locations at which not objects exists are mapped to 0 and the semantic ensures that monitors can only be acquired for locations at which objects exist. The initial monitor state mon_0 is defined as $mon_0(l) = 0$ for all $l \in \mathcal{L}$.

The third and fourth part is a mode state and a heap, respectively. The fifth part is the program that is currently executed. It is needed to obtain the map between program points an instructions for each method, since the call stack saves the program points and method identifiers only.

A global configuration is a snapshot of the entire transition system. It contains the snapshots of all threads that were created until now and a snapshot of the heap.

Definition 2.13. *The set of global configurations of the ATS is*

$$Conf = (\mathcal{CS} \times (\mathcal{L} \to \mathbb{N}) \times Mds)^* \times \mathcal{H} \times \mathrm{P}\,.$$

The first part of the global configuration is a list of threads. Each thread consists of a call stack, monitors and a mode state. The number of threads is not bounded to support dynamic thread creation. The second and third part consists of a snapshot of the heap and the program, respectively. It is important to keep track of terminated threads, to access their mode states for the security analysis. Otherwise it would not be possible to impose certain restrictions on these mode states, e.g. that they are empty.

2.3.6 Transition Relation

In this section we define transition relations on local and global configurations.

The local transition relation describes the possible execution steps of a single thread. It is not further specified, to be instantiated with the semantics of the instructions in an instruction set $\mathcal{I}$.

To model the interaction of a thread with the rest of the transition system (i.e. synchronization and thread creation) we use events that are emitted on the execution of instructions. The set of events is:

$$\mathrm{E} = \{\varepsilon\} \cup \{\blacklozenge l, \lozenge l : l \in \mathcal{L}\} \cup (\mathcal{CS} \times (\mathcal{L} \to \mathbb{N}) \times Mds)$$

An instruction can emit no event, which is represented by ε, or it can signalize the beginning or end of monitor region by $\blacklozenge l$ and $\lozenge l$, respectively. The location of the corresponding object is send as l. The third event is used to create a new thread, the event itself is the new thread and contains all information that were available at creation.

Definition 2.14. *The local transition relation is a labeled transition relation on local configurations $\twoheadrightarrow \subseteq Conf_l \times \mathrm{E} \times Conf_l$. We write $c_1 \overset{e}{\twoheadrightarrow} c_2$ for $(c_1, e, c_2) \in \twoheadrightarrow$.*

The global transition relation describes the possible execution steps of the whole transition system and models thread creation and synchronization.

Definition 2.15. *The global transition relation is the smallest binary relation on global configurations $\rightsquigarrow \subseteq Conf \times Conf$ that satisfies the following rules.*

NO-EVENT

$$\frac{(cs_i, mon_i, mds_i, h)_P \overset{\varepsilon}{\twoheadrightarrow} (cs_i', mon_i, mds_i', h')_P}{((t_1, \ldots, (cs_i, mon_i, mds_i), \ldots, t_n), h)_P \rightsquigarrow ((t_1, \ldots, (cs_i', mon_i, mds_i'), \ldots, t_n), h')_P}$$

THREAD-CREATE

$$\frac{(cs_i, mon_i, mds_i, h)_P \xrightarrow{t} (cs'_i, mon_i, mds'_i, h')_P \qquad t \in (CS \times (\mathcal{L} \to \mathbb{N}) \times Mds)}{((t_1, \ldots, (cs_i, mon_i, mds_i), \ldots, t_n), h)_P \rightsquigarrow ((t_1, \ldots, (cs'_i, mon_i, mds'_i), \ldots, t_n, t), h')_P}$$

MONITOR-ENTER

$$\frac{\begin{array}{c}(cs_i, mon_i, mds_i, h)_P \xrightarrow{\blacklozenge l} (cs'_i, mon_i, mds'_i, h')_P \\ l \in dom(h) \qquad \forall j \in \mathbb{N} : (1 \leq j \leq n \land j \neq i) \implies \pi_{mon}(t_j)(l) = 0 \\ mon'_i = mon_i[l \mapsto mon_i(l) + 1]\end{array}}{((t_1, \ldots, (cs_i, mon_i, mds_i), \ldots, t_n), h)_P \rightsquigarrow ((t_1, \ldots, (cs'_i, mon'_i, mds'_i), \ldots, t_n), h')_P}$$

where $\pi_{mon}((cs, mon, mds)) = mon$

MONITOR-EXIT

$$\frac{\begin{array}{c}(cs_i, mon_i, mds_i, h)_P \xrightarrow{\lozenge l} (cs'_i, mon_i, mds'_i, h')_P \\ mon_i(l) > 0 \qquad mon'_i = mon_i[l \mapsto mon_i(l) - 1]\end{array}}{((t_1, \ldots, (cs_i, mon_i, mds_i), \ldots, t_n), h)_P \rightsquigarrow ((t_1, \ldots, (cs_i, mon'_i, mds'_i), \ldots, t_n), h')_P}$$

The global transition relation executes the individual threads in a possibilistic manner and models therefore a possibilistic scheduler. There are only four types of execution steps, one for each event, that are modeled as inference rules.

In the first rule a thread can make a step, if a step in the local transition relation is possible that does not emit an event. In addition the monitor state of that thread needs to remain unchanged, since the monitors shall only change if certain requirements are met.

The second rule allows the creation of threads. While creating a new thread, the current thread can do modifications of its state, except for the monitor state. The new thread is appended to the thread pool. Since the execution is possibilistic the position of the new thread in the thread pool is not relevant. The position at the end of the thread pool is merely used for space reasons.

The last two rules are concerned with the monitors. The rule MONITOR-ENTER handles he acquiring of monitors. A monitor can be acquired if there is an object at the location specified by the event and if there is no other thread that currently owns a monitor for this object. If these conditions are satisfied a transition is possible and the monitor state of the acquiring thread is increased by one at the location of the object.

The fourth rule models the case where a thread leaves a monitor region. A thread can only leave a monitor region if it currently owns a monitor for the specific object. If this is the case, the monitor state of the thread will be decreased by one for the objects location. The reentering of a thread into a monitor region is adequately captured with the natural number representing the number of times the thread has entered a monitor region.

Now we can define the abstract transition system.

Definition 2.16. *An ATS is a transition system that is parametric in the initial state, the program, the values, the instructions and the local transition relation* $ATS(conf_0, \mathrm{P}, \mathbb{V}, \mathcal{I}, \rightharpoonup) = (Conf, conf_0, Conf_{\mathrm{F}}, \rightsquigarrow, \rightharpoonup, \mathbb{V}, \mathrm{P})$*, where* $conf_0 \in Conf$ *and* $Conf_{\mathrm{F}} = \{((\langle\rangle, mon_1, mds_1), \ldots, (\langle\rangle, mon_n, mds_n), h) : h \in Heap,$ $mon_i \in (\mathcal{L} \to \mathbb{N}), mds_i \in Mds, 1 \leq i \leq n\}$.

An ATS can be instantiated to a transition system by supplying an initial configuration, a program, a set of values, instructions and a local transition relation that models the semantics of the instructions. The set of configurations is fixed by the ATS, as well as the global transition relation and the final configurations. The final configurations are configurations in which all threads have an empty call stack. That are exactly those threads that “returned” from the initially called method.

Chapter 3

SIFUM Security

In this chapter we define the new assumption and guarantee style security property for the ATS that was defined in the previous chapter.

3.1 Security Policy

With security properties we want to classify which programs preserve the confidentiality of entrusted data. Therefore we need to define what confidentiality is.

We consider the security domains *low* and *high* to partition the data containers into publicly accessible and into private containers. The security domain low is assigned to a container that may store confidential information, whereas the security domain low is assigned to containers that may store public information only.

Intuitively a program preserves the confidentiality of the entrusted data, if no confidential information flows into containers with security domain low.

A *security policy* defines the notion of confidentiality for a concrete program. It contains the assignment of security domains to containers and the intended information flow between the security domains.

Definition 3.1. *A two-level security policy is a lattice* $(\mathcal{D}, \leq, low, high, \sqcup, \sqcap)$ *with a field domain assignment* $ft : \mathcal{F} \to \mathcal{D}$ *where* $\mathcal{D} = \{high, low\}$ *and* $\leq$ *is the least partial order satisfying* $low \leq high$.

The containers that hold the data are in our case the fields of objects. The field domain assignment is not object sensitive to reduce complexity, therefore every object of a certain class has the same security domains. If $ft(f_1) \leq ft(f_2)$ for some fields f_1 and f_2 then information are allowed to flow from the field f_1 into the field f_2. All other information flow is forbidden by the policy.

3.2 Attacker Model

Assuming that access control works correctly, we consider an attacker that can view the *low* fields and class identifiers of objects at visible locations before and after the execution of the program. A location is visible to an attacker if the object at this location has at least one field with a *low* security domain.

In addition he is able to modify the initial values of low fields, to observe the duration of the execution and to study the source code of the program.

A potential attack scenario could be the following. The attacker builds a malicious store application that has the ability to display advertisements and distributes the application. To make use out of this application the user has to provide it with his credit card number. The user believes that his credit card number is only send to the store (e.g. Amazon), in which he trusts. However the attacker has designed the application to leak the users credit card number through the advertisement part of the application to an untrusted party. The untrusted parties are "reachable", because the application needs full access to the internet to fetch the advertisements.

When applying this scenario to our formal model, we have to simulate input/output behavior by reading/storing the data in certain fields on the heap, because our formal model does not support input/output behavior. The credit card number is stored in a high field in the initial state. We now assume that the credit card number of the user is stored on the heap before program execution in a high field. During the program execution the credit card number is leaked directly into the low field that models the sending of data to the advertisement servers.

3.3 Indistinguishability Relations

The capabilities of the attacker determine which parts the program state he is able to see. In consequence an attacker can only distinguish states up to his capabilities. We capture the attackers capabilities by defining relations that describe when two states look equivalent to the attacker.

The objects on the heap consist usually of more than one field and locations can be saved within fields, thus it is not obvious which locations are visible to the attacker. We approximated the visibility of locations by rendering all locations visible that contain at least one field with security domain *low*, following [HP06]. This is an over-approximation since the objects could be completely invisible to the attacker if their location is saved in a high field. Further the heap stores dynamically allocated data structures, thus the allocation of locations might be different in two – otherwise identical – program runs.

Both aspects are captured using a partial map $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ that is isomorphic on the locations that contain at least one field with a *low* security domain. Heap equivalence will be defined only on those locations that are in the domain of β.

The following definition captures when a partial bijection is an isomorphism between the visible locations of two heaps.

Definition 3.2. *A partial bijection $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ between heaps $h_1, h_2 \in \mathcal{H}$ is safe if and only if the following conditions are satisfied.*

1. $dom(\beta) = \{l \in dom(h_1) : \exists f \in dom(\pi_2(h_1(l))).ft(f) = low\}$
2. $codom(\beta) = \{l \in dom(h_2) : \exists f \in dom(\pi_2(h_2(l))).ft(f) = low\}$

The requirement on such a partial bijection is that the bijection is only defined on locations that point to an object with at least one low field. An object at a location that is in the domain of β is also called visible object, all other objects are called invisible objects.

Another approach would be to consider all locations in a register set that originate from a low field as visible, together with all locations that are reachable through the low fields of the objects at these locations. This would allow to hide objects with low fields by storing their location in a high field. However the downside of this approach is, that concurrent executed threads could render previously hidden locations visible by moving a location from a low field into a register, that is not contained in the register sets of the other threads. Therefore the visible locations differ depending on the current register set of a thread and it cannot be ensured that the objects at the visible locations of all register sets are indistinguishable to an attacker.

Now we define indistinguishability on values up to a partial bijection β.

Definition 3.3. *Two values $v_1, v_2 \in \mathcal{V}$ are indistinguishable with respect to a partial bijection $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ (written $v_1 \sim_\beta v_2$) if and only if the following conditions is satisfied:*

1. $v_1, v_2 \in \mathbb{V} \cup \{null\} \implies v_1 = v_2$
2. $v_1, v_2 \in \mathcal{L} \implies v_1 \in dom(\beta) \wedge \beta(v_1) = v_2$

Elements of $\mathbb{V}$ and *null*'s are indistinguishable if they are equal. If the values v_1 and v_2 are locations, they are indistinguishable for the attacker if they can be associated via β, i.e. the locations belong to low objects.

The attacker can distinguish heap elements in two cases. First, if the objects are instances of different classes (i.e. the class identifier differs). Second, if there is a field with a low security domain and with distinguishable values.

Definition 3.4. *Two heap elements $(c_1, f_1), (c_2, f_2) \in (\mathcal{C} \times \mathcal{F} \rightharpoonup \mathcal{V})$ are indistinguishable with respect to a partial bijection $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ and a mode state $mds \in Mds$ (written $(c_1, f_1) \sim_\beta^{mds} (c_2, f_2)$) if and only if the following conditions are satisfied*

1. $c_1 = c_2$
2. $\forall fid \in dom(f_1) : (\mathrm{ft}(fid) = low \wedge fid \notin mds(asm\text{-}noread))$
   $\implies fid \in dom(f_2) \wedge f_1(fid) \sim_\beta f_2(fid)$
3. $\forall fid \in dom(f_2) : (\mathrm{ft}(fid) = low \wedge fid \notin mds(asm\text{-}noread))$
   $\implies fid \in dom(f_1) \wedge f_1(fid) \sim_\beta f_2(fid)$

This definition captures on the one hand the capability of the attacker to observe values of low fields in the initial and final states. The fields are only indistinguishable if they are initialized in the same low fields and these fields contain indistinguishable values. On the other hand it captures the possibility to temporarily store confidential data in a low field, by requiring indistinguishability of fields only for fields without a no-read assumption. Note that this is only adequate if this assumption is matched by the corresponding guarantees, i.e. no concurrent thread reads that field.

As stated in Section 3.2 the attacker can distinguish all elements at locations containing at least one field with a low security level. That means, that heaps can differ in the allocation of objects containing only high fields. This intuition is captured in the following definition.

Definition 3.5. *Two heaps $h_1, h_2 \in \mathcal{H}$ are indistinguishable modulo modes with respect to a partial bijection $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ that is safe on h_1 and h_2 (written $h_1 \sim_\beta^{mds} h_2$) if and only if*

$$\forall l \in dom(\beta) : h_1(l) \sim_\beta^{mds} h_2(\beta(l))$$

Two indistinguishable heaps have to store an object at all visible locations (i.e. all locations in the domain of β) and these objects must be indistinguishable. Otherwise an attacker could either distinguish the objects or tell from the absence of one object that the value of some high field has changed.

Example 4. *Let $l, h, noread, hidden \in \mathcal{F}$ and $c, c', c'' \in \mathcal{C}$ then $h_1, h_2 \in \mathcal{H}$ where*

$$h_1 : l \mapsto \begin{cases} (c, \{(l, l_1), (h, 3)\}) & l = l_0 \\ (c, \{(l, l_1)\}) & l = l_1 \\ (c', \{(noread, 5)\}) & l = l_4 \end{cases} \quad h_2 : l \mapsto \begin{cases} (c, \{(l, l_4), (h, 5)\}) & l = l_2 \\ (c, \{(l, l_4)\}) & l = l_4 \\ (c', \{(noread, 1)\}) & l = l_5 \\ (c'', \{(hidden, 42)\}) & l = l_6 \end{cases}$$

For $ft(l_c) = ft(noread) = low$, $ft(h_c) = ft(hidden) = high$, $noread \in mds(asm\text{-}noread)$ and $\beta = \{(l_0, l_2), (l_1, l_4), (l_4, l_5)\}$ we have $h_1 \sim_\beta^{mds} h_2$.

The following theorem shows that locations that are saved in a low field without no-read assumptions point always to visible objects.

Theorem 1. *If β is safe for h_1 and h_2 and $h_1 \sim_\beta^{mds} h_2$ then for all $l \in dom(h_1)$*

$$\forall f \in dom(\pi_2(h_1(l))) : (ft(f) = low \wedge f \notin mds(asm\text{-}noread) \wedge h_1(l)(f) \in \mathcal{L}) \\ \implies h_1(l)(f) \in dom(\beta)$$

Proof. Let $h_1, h_2 \in \mathcal{H}$ and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ arbitrary such that β is safe for h_1 and h_2 and $h_1 \sim_\beta^{mds} h_2$ for some arbitrary $mds \in Mds$. Let $l \in dom(h_1)$.

Case $l \notin dom(\beta)$: From the definition of a safe partial bijection we know that $l \in dom(\beta) \Leftrightarrow \exists f \in dom(\pi_2(h_1(l))) : ft(f) = low$ thus we have $l \notin dom(\beta) \Leftrightarrow \neg\exists f \in dom(\pi_2(h_1(l))) : ft(f) = low$. Therefore there exists no $f \in dom(\pi_2(h_1(l)))$ such that $ft(f) = low$, thus the proposition is trivially satisfied.

Case $l \in dom(\beta)$: From the definition of a safe partial bijection we know that there is some $f \in dom(\pi_2(h_1(l)))$. We now assume that $ft(f) = low$, $f \notin mds(asm\text{-}noread)$, $h_1(l)(f) \in \mathcal{L}$ and $h_1(l)(f) \notin dom(\beta)$. From the definition of heap and object indistinguishability and $h_1 \sim_\beta^{mds} h_2$ we get that $h_1(l)(f) \sim_\beta h_2(\beta(l))(f)$. From $h_1(l)(f) \in \mathcal{L}$ and condition (2) of value indistinguishability we get that $h_1(l)(f) \in dom(\beta)$ which is a contradiction to our assumption. Therefore the proposition holds.

□

In the DVM the heap is primarily used as a storage. To do computations on heap elements, these elements need to be moved into a register set. Within this register set the actual computations are done and the result is afterwards moved back onto the heap.

This usage of the register sets requires, that the security domains of the fields are reflected in security domains for register identifiers. Even though the attacker cannot directly access the values of registers, it will become necessary for defining the security property to have a notion of indistinguishability on register sets to adequately capture the possible changes of registers that depend on high fields.

Definition 3.6. *Two register sets $rs_1, rs_2 \in (\mathcal{R} \to \mathcal{V})$ are indistinguishable with respect to a register domain assignment $rt : \mathcal{R} \to \mathcal{D}$ and a partial bijection $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ (written $rs_1 \sim_\beta^{rt} rs_2$) if and only if the following conditions hold*

1. $\forall r \in dom(rs_1) : rt(r) = low \Rightarrow r \in dom(rs_2) \wedge rs_1(r) \sim_\beta rs_2(r)$
2. $\forall r \in dom(rs_2) : rt(r) = low \Rightarrow r \in dom(rs_1) \wedge rs_1(r) \sim_\beta rs_2(r)$

Two register sets are indistinguishable if they agree on all initialized register identifiers with a low security domain. The security domain assignment for register identifiers is induced completely by the field domain assignment and the flow from the heap into the register set. How such a register domain assignment is constructed is described in detail in the next section.

If the value of a field with a no-read assumption is moved into a field, the no-read assumption is approximated by a high security domain. This is reasonable, because concurrently executed threads cannot directly access the values of registers. Therefore the register do not have to be set to a public value when the corresponding guarantee is released.

To ensure that no data is leaked on method calls, we lift the notion of indistinguishability on register sets to call stacks.

Definition 3.7. *Two call stacks $cs_1, cs_2 \in \mathcal{CS}$ are indistinguishable with respect to a stack of register domain assignments $rts \in (\mathcal{R} \to \mathcal{D})^*$ and a partial bijection $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ (written $cs_1 \sim_\beta^{rts} cs_2$) if and only if both call stacks are empty or $cs_1 = (p_1, mid_1, rs_1) :: cs_1'$, $cs_2 = (p_2, mid_2, rs_2) :: cs_2'$, $rts = rt :: rts'$ and the following conditions are satisfied*

1. $rs_1 \sim_\beta^{rt} rs_2$
2. $cs_1' \sim_\beta^{rts'} cs_2'$

Two call stacks are indistinguishable if they have the same height and if the register sets are point-wise indistinguishable. This ensures that no information is leaked during a method call into a lower stack frame, because the lower register sets need to be indistinguishable. The requirement that both call stacks must be at the same height implies that the number of method calls cannot differ while preserving call stack indistinguishability. This requirement is an over-approximation to simplify the definition of call stack indistinguishability. The only methods that would produce indistinguishable call stacks, where the top-most register sets are indistinguishable are methods, that call them self before doing any register modifications.

3.4 Register Domain Assignments

In this section we discuss how register domain assignments can be constructed for a given program and security policy.

There are at least two approaches for defining register domain assignments: A static and a dynamic approach. The goal of both approaches is to reflect whether a register may contain confidential information.

The static approach needs for every method a register domain assignment and a method signature. The register domain assignments are constructed by analyzing the flow into the registers. If a register has a high field as (direct or indirect) source (i.e. if the value of a high field is moved into a register, or if the value of such a register is propagated into another register) then the register has a high security domain.

Example 5. *If $ft(h) = high$ and $ft(l) = low$ and m is a method with*

$$m : i \mapsto \begin{cases} put\ v_0\ l & i = 1 \\ put\ v_0\ h & i = 2 \\ const\ v_1\ 5 & i = 3 \\ add\ v_2\ v_0\ v_1 & i = 4 \end{cases}$$

then the register domain assignment is $rt(v_0) = rt(v_2) = high$ and $rt(r) = low$ for all $r \in \mathcal{R} \setminus \{v_0, v_2\}$.

The method signatures provide security domains for all registers that have no field (indirectly or directly) as a source, to obtain a total register domain assignment. In the previous example the method signature assigns *low* to all registers except v_0 and v_2.

The dynamic approach does not fix some register domain assignment for a method, but rather uses some initial register domain assignment and changes the security domains of the registers appropriately by observing the flow into the registers. The advantages of the dynamic approach are, that it is more precise in the sense that it captures only the sources that are relevant at the current instruction. Whereas the static approach over-approximates over all sources at all instructions in the method. In the previous example the register domain assignment at instruction 1 would be $rt(r) = low$ for all $r \in \mathcal{R}$ if the initial register domain assignment assigns *low* to all registers. The precision of the static approach could be increased by program transformation, i.e. if a register has high and low fields as sources, one could introduce a new register and use one for the high and the other for the low sources (as long as fresh registers are available).

The other advantage is that except for some initial register domain assignment we do not need method signatures, because we can create a new register domain assignment for a specific method call from the security domains of the method parameters.

The disadvantage of the dynamic approach is, that we need to pull the analysis of the data flow into the security property. The static approach would also require a program analysis to obtain the register domain assignments, but since these are fixed, the analysis could be separated from the security property.

We use the dynamic approach to take advantage of the increased precision and to get rid of method signatures, since they would be not directly related to our notion of the security policy.

The field domain assignments are fixed by the security policy. Therefore we have to observe the flow into the registers to compute a register domain assignment. According to the definition of indistinguishability for register sets and call stacks, we distinguish four classes of instructions that produce data flow into registers.

The first class contains instructions that manipulate registers in the register set of the top-most stack frame (e.g. instructions that move the values of fields into registers). The second class contains instructions that put new stack frames on the call stack (e.g. method calls) and the third class contains instructions that remove stack frames from the call stack (e.g. return instructions). The fourth class contains instructions that create new singleton call stacks (i.e. thread creation).

We use events that are emitted on the execution of instructions to know to which class the executed instruction belongs to. Therefore we extend the set of events E with the following events.

$$\begin{aligned}E' =& \{r_1 \dots r_n; f_1 \dots f_m; c_1 \dots c_p \triangleright r : r_1, \dots, r_n \in \mathcal{R}, f_1, \dots, f_m \in \mathcal{F}, c_1, \dots, c_p \in \mathcal{C}\} \cup \\ & \{r_1 \dots r_n : r_1, \dots, r_n \in \mathcal{R}\} \cup \\ & \{\triangledown r : r \in \mathcal{R}\} \cup \{\triangledown\} \\ E'' =& \{cs; r_1 \dots r_n : cs \in \mathcal{CS}, r_1, \dots, r_n \in \mathcal{R}\}\end{aligned}$$

The event $r_1 \dots r_n; f_1 \dots f_m; c_1 \dots c_p \triangleright r$ models that the register identifier r is influenced by the register, field and class identifiers mentioned in the event. The case $n = m = p = 0$ is legal and models that the register r depends only on constant values.

$r_1 \dots r_n$ is emitted by an instruction from class two and contains the registers that are copied (as parameters) into the new register identifier. The event $\triangledown$ is emitted on return instructions that do not pass return values to the caller and the event $\triangledown r$ on return instructions that pass the register identifier r to the caller. The event $cs; r_1 \dots r_n$ is emitted on thread creation and replaces the old event for thread creation. The intuition of this event is that the register set of the created thread is initialized with the values of the register identifiers $r_1, \dots, r_n$.

To treat these new events in the global transition relation, we need to replace the rule NO-EVENT and THREAD-CREATE by the following rules.

LOCAL-EVENT-NEW

$$\frac{(cs_i, mon_i, mds_i, h)_P \overset{\alpha}{\twoheadrightarrow} (cs'_i, mon_i, mds'_i, h')_P \qquad \alpha \in E' \cup \{\varepsilon\}}{((t_1, \dots, (cs_i, mon_i, mds_i), \dots, t_n), h)_P \rightsquigarrow ((t_1, \dots, (cs'_i, mon_i, mds'_i), \dots, t_n), h')_P}$$

THREAD-CREATE-NEW

$$\frac{(cs_i, mon_i, mds_i, h)_P \overset{cs}{\twoheadrightarrow} (cs'_i, mon_i, mds'_i, h')_P \qquad cs; r_1 \dots r_n \in E''}{((t_1, \dots, (cs_i, mon_i, mds_i), \dots, t_n), h)_P \rightsquigarrow ((t_1, \dots, (cs'_i, mon_i, mds'_i), \dots, t_n, (cs, mon_0, mds_0)), h')_P}$$

Our next goal is to establish a link between the emitted events and the register domain assignments. This link is the *update* function which updates a register domain assignment according to an event. The events and the *update* function deals only with direct flows into registers. Indirect flows are captured in the security property. This leads to an over-approximation, because a program that has an indirect flow of confidential data into a register is rendered insecure, although the attacker could only distinguish these states if the register content were copied to the heap.

Definition 3.8. *The function update* $: E' \times (\mathcal{R} \to \mathcal{D})^* \times Mds \to (\mathcal{R} \to \mathcal{D})^*$ *is defined by case distinction on* $\alpha \in E'$ *and* $rts \in (\mathcal{R} \to Dom)^*$.

1. *if* $\alpha = r_1 \ldots r_n f_1 \ldots f_m c_1 \ldots c_p \rhd r$ *and* $rts = rt :: rts'$ *then* $rt[r \mapsto \bigsqcup\{rt(r_1), \ldots, rt(r_n), ft(f_1), \ldots, ft(f_m), ct(c_1), \ldots, ct(c_n)\} :: rts'$
2. *if* $\alpha = r_1 \ldots r_n f_1 \ldots f_m c_1 \ldots c_p \rhd r$, $rts = rt :: rts'$ *and* $\exists 0 < i \leq m : f_i \in mds(asm\text{-}noread))$ *then* $rt[r \mapsto high] :: rts'$.
3. *if* $\alpha = r_1 \ldots r_n$ *then* $\{(v_i, rt(r_{i+1})) : 0 \leq i < n\} \cup \{(r_i, low) : n \leq i\} :: rts$
4. *if* $\alpha = \nabla r$ *and* $rts = rt_1 :: rt_2 :: rts'$ *then* $rs_2[v_{res} \mapsto rt_1(r)] :: rts'$
5. *if* $\alpha = \nabla$ *and* $rts = rt :: rts'$ *then* rts'
6. *otherwise* rts

where $ct : \mathcal{C} \to \mathcal{D}$ *is defined as* $ct(c) = \bigsqcap\{ft(f) : f \in fields(c)\}$.

The *update* function considers only events that effect the register domain assignments of the current thread. The new event for thread creation is handled separately in the next section.

The first case of the *update* function covers flows into a register at the topmost stack frame. We have to regard everything as sources that has a security domain. That are prominently register and field identifiers, but also class identifiers have some kind of security domain. Namely a class identifier is considered as high if the associate field identifiers have solely high security domains and low if they have a low field identifier. This captures the visibility of class instances with security domains. This intuition is captured in the definition of the function *ct*.

The security domain of the influence register is the least upper bound of the involved register, field and class identifiers. This is adequate, because if at least one source has a high security domain, the influenced register will also have a high domain. If the new value is a constant (i.e. if no source is given) the new security domain is $low = \bigsqcup \emptyset$ because the value is directly visible in the source code, to which the attacker has access. It is important to regard the class identifiers here, because otherwise the event for an instruction that creates a new instance of a class c and saves the location of the created object in a register r would be $\rhd r$. Thus the register security domain of r would be low. This would make it impossible to create an invisible object and to save its location in register r, since the location of the invisible object would not be related by β and thus those register sets would be in every case distinguishable.

The second case approximates no-read assumptions. If a register is influenced by at least one field with a no-read assumption, then the security domain of the influenced register is high. The intuition is, that a field with a no-read assumption can temporarily store confidential values which is taken into account with the high security domain of the influenced register.

The third case covers instructions for method invocations. The event consists of the registers that are used as parameters. Intuitively the register set of the called method is influenced by the method parameters. Here we make the assumption that those parameters are always copied into the first n registers of the register set. This assumption is true for our semantics of the DVM in Section 4.1. Therefore we assign the security domains of the registers send in the event to the first n register of the called method. All other register identifiers are assigned *low*, because we assume that the event correctly reflects the behavior of the instruction and therefore copies values only in the first n registers.

The forth case treats the return from a method call with providing a return value. To ensure that the security domain of the return value is correctly reflected in the result register of the caller, we adopt its security domain. In addition we remove the top register domain assignment to ensure that the height of the register domain assignment stack corresponds to the height call stack. The fifth case is similar to the third, but does not treat return values.

The initial register domain assignment rt_0 is defined as $rt_0(r) = low$ for all $r \in \mathcal{R}$. Further rts_0 is the singleton register stack with rt_0.

Example 6. *Suppose the instruction* `doStuff` v_0 v_1 f_1 f_2 *(with* $v_0, v_1 \in R$ *and* $f_1, f_2 \in \mathcal{F}$*) emits the event* $v_1 f_1 f_2 \rhd v_0$*. This event indicates that the value at register identifier* v_0 *depends now on the values stored at the register identifier* v_1 *and the field identifiers* f_1 *and* f_2*. The update function approximates relative to an register domain assignment and a field domain assignment the security domain of the register* v_0*. If* $rt(v_0) = high$*,* $rt(v_1) = low$*,* $ft(f_1) = low$ *and* $ft(f_2) = high$ *the register domain assignment with the approximation of* v_0 *security domain is* $update(rt, v_1 f_1 f_2 \rhd v_0) = rt[v_0 \mapsto \bigsqcup\{rt(v_1), ft(f_1), ft(f_2)] = rt[v_0 \mapsto high]$*.*

3.5 Security Property

We now use the indistinguishability relations from Section 3.3 to define a bisimulation relation on local configurations, that relates local configurations if they have secure information flow.

The idea is that the complete execution of two related local configurations that are indistinguishable to a low observer, remain indistinguishable in each execution step even if the heap is modified by concurrently executed threads – that respect the no-write assumptions of the local configurations – between the execution steps.

In formalizing this idea we follow the approach of [MSS11] and distinguish between the execution steps of the thread's environment and the execution steps of the thread itself.

First we define two closure conditions that describe the heap modifications by concurrently executed threads. These closure conditions take only those memory modifications into account that can actually occur by exploiting the no-write assumptions. A field with a no-write assumption cannot be changed by a concurrently executed thread that respects the no-write assumption. The first closure condition covers modifications on already existing objects and the second closure condition covers the creation of objects.

Definition 3.9. *A relation R on local configurations with equal mode states is closed under globally consistent field modifications if whenever*

$$(cs_1, mon_1, mds, h_1) R^{rts}_{\beta} (cs_2, mon_2, mds, h_2)$$

and β is safe for h_1 and h_2, for all $l \in \mathcal{L}$ and $f \in \mathcal{F}$ the following conditions hold:

1. $(l \in dom(h_1) \wedge f \in fields(\pi_1(h(l))) \wedge f \notin mds(asm\text{-}nowrite) \wedge ft(f) = high) \implies \forall v \in \mathcal{V} (cs_1, mon_1, mds, h_1[f \mapsto v]) R^{rts}_{\beta} (cs_2, mon_2, mds, h_2)$

2. $(l \in dom(\beta) \wedge f \in fields(\pi_1(h(l))) \wedge f \notin mds(asm\text{-}nowrite) \wedge ft(f) = low) \implies \forall v \in \mathbb{V} \cup \{null\} \cup dom(\beta)$
$(cs_1, mon_1, mds, h_1[f \mapsto v]) R^{rts}_{\beta} (cs_2, mon_2, mds, h_2[f \mapsto \tau(v)])$

3. $(cs_2, mon, mds, h_2) R^{rts}_{\beta^{-1}} (cs_1, mon, mds, h_1)$

where $\tau(v) := \begin{cases} \beta(v) & v \in dom(\beta) \\ v & otherwise \end{cases}$

The first case covers the modification of high fields without no-write assumption. Such fields can be modified arbitrarily. Since the number of high objects in indistinguishable heaps can differ, it is not possible to associate high objects in the related heaps. Therefore the heaps can be modified independently from each other in high fields without no-write assumptions. The updates of the heap of the second local configuration are covered by the symmetry condition (3). Since heap indistinguishability is defined with respect to a partial bijection β, the symmetric case needs to be defined with respect to the inverse of β. The first condition also applies to the high fields of low objects, since the high fields can differ in the initialization state.

The second case covers the modification of low fields with no-write assumption. Every low field is contained by definition in a low object, thus the objects must be associated via β. Since the new value of the low fields must not contain a secret, it is modified in an indistinguishable way in both heaps. Thus the new value can either be a primitive value from $\mathbb{V}$, *null* or a location of a low object. The function τ chooses in case of a location, the location of the corresponding object in the other heap.

The second closure condition covers the creation of new objects by concurrently executed threads.

Definition 3.10. *A relation R on local configurations with equal mode states is closed under globally consistent object creations if whenever*

$$(cs_1, mon_1, mds, h_1) R^{rts}_{\beta} (cs_2, mon_2, mds, h_2)$$

and β is safe for h_1 and h_2, for all $l \in \mathcal{L}$, $f \in \mathcal{F}$ and $c \in \mathcal{C}$ the following conditions hold

1. $(l \notin dom(h_1) \wedge (\forall f \in fields(c) : ft(f) = high \wedge f \notin mds(asm\text{-}nowrite)))$
$\implies (\forall fs \in (\mathcal{F} \rightharpoonup \mathcal{V}) : \neg(dom(fs) \subseteq fields(c)) \vee (cs_1, mon_1, mds, h_1[l \mapsto (c, fs)]) R^{rts}_{\beta} (cs_2, mon_2, mds, h_2))$

2. $(l \notin dom(h_1) \wedge (\exists f \in fields(c) : ft(f) = low) \wedge (\forall f \in fields(c) : f \notin mds(asm\text{-}nowrite))) \implies (\forall l' \notin dom(h_2) : \forall fs_1, fs_2 \in (\mathcal{F} \rightharpoonup \mathcal{V}) :$
$\neg(dom(fs_1) \subseteq fields(c) \wedge dom(fs_2) \subseteq fields(c)$
$\wedge\, fs_1 \restriction fields_{low}(c)(= \times \sim_\beta) fs_2 \restriction fields_{low}(c))$
$\vee (cs_1, mon_1, mds, h_1[l \mapsto (c, fs_1)]) R^{rts}_{\beta \cup \{(l,l')\}} (cs_2, mon_2, mds, h_2[l' \mapsto (c, fs_2)]))$

3. $(cs_2, mon, mds, h_2) R^{rts}_{\beta^{-1}} (cs_1, mon, mds, h_1)$

where $fields_{low}(c) = \{f \in fields(c) : ft(f) = low\}$

The other kind of modifications that concurrently executed threads can do is object creation. It is necessary to cover the creation of objects, firstly to ensure that no secrets are leaked into the heap and secondly because these new objects can be directly visible in the local configurations by modifying/overwriting pointers.

The first case covers the creation of high objects. The only case in which a high object can be created by concurrent threads, is if there is no field with a no-write assumption, that belongs to the class of the object that shall be created. If there were such a field the no-write assumption would be violated, because modes are not object sensitive and after the creation of the object a (maybe uninitialized) field appears. If these conditions are satisfied a new object on both heaps (due to the symmetry condition (3)) with an arbitrary assignment of the fields, belonging to the class of the object, can be created.

The second case covers the creation of low objects. We first ensure that the class contains at least one low field and search for a location that is not already used. As in the case for the high object we forbid the creation if some field has a no-write assumption. If these conditions are satisfied a new object can be created in both heaps at an unused location. As the newly created object is a low object, we have to extend the bijection β with the new locations to obtain a safe β again. In addition we have to ensure that the low fields of both objects are created equally, i.e. either a field is in both heaps uninitialized or has an indistinguishable value. This is modeled with the product of equality on field identifiers and indistinguishability on values $(= \times \sim_\beta)$. The intuition is that all field that are initialized in both objects (i.e. the identifier is contained in the graph field assignments) have indistinguishable values. All high fields that belong to the corresponding class can be arbitrary.

One might think that the object insensitivity of assumptions and guarantees would cause problems when releasing no-read assumptions of low fields. If a no-read assumption is released the (possible) secret must be overwritten by a public value, otherwise the attacker could distinguish them. The object insensitivity implies that we need to overwrite the field in every instance it occurs, which leads (with most semantics) to multiple computation steps. One could now argue that concurrently executed threads could "redo" these changes, but this is not possible, since they can only change low fields in an indistinguishable way.

Now we define a strong bisimulation that captures the memory modifications of a single thread. This bisimulation takes advantage of the no-read assumptions to determine whether secrets may be stored in low fields.

Definition 3.11. *A relation R on local configurations with equal mode and monitor states that is closed under globally consistent field assignments and closed under object creations is a strong low bisimulation modulo modes if whenever*

$(cs_1, mon, mds, h_1) R^{rts}_\beta (cs_2, mon, mds, h_2)$ the following conditions hold if β is safe for h_1 and h_2:

1. *$h_1 \sim^{mds}_\beta h_2$*
2. *$cs_1 \sim^{rts}_\beta cs_2$*
3. *if $(cs_1, mon, mds, h_1) \overset{\alpha}{\rightharpoonup} (cs'_1, mon', mds', h'_1)$ then there exists $cs'_2 \in \mathcal{CS}$, $h'_2 \in \mathcal{H}$ and $\beta' \supseteq \beta$ such that the following holds*
   (a) *$(cs_2, mon, mds, h_2) \overset{\gamma}{\rightharpoonup} (cs'_2, mon', mds', h'_2)$, β' is safe for h'_1 and h'_2 and $(cs'_1, mon', mds', h'_1) R^{update(\alpha, rts)}_{\beta'} (cs'_2, mon', mds', h'_2)$*
   (b) *if $\alpha = cs; r_1 \ldots r_n$ and $\gamma = cs'; r_1 \ldots r_n$ then $mds' = mds_0$ and $(cs, mon_0, mds_0, h_2) R^{update(r_1 \ldots r_n, \langle\rangle)}_{\beta'} (cs', mon_0, mds_0, h'_2)$*
4. *$(cs_2, mon, mds, h_2) R^{rts}_{\beta^{-1}} (cs_1, mon, mds, h_1)$*

The relation $\approx$ is the union of all such bisimulations.

The idea is that if two local configurations are related by such a bisimulation they have secure information flow if they are executed in an environment that respects no-write assumptions.

The requirement that the environment (i.e. the concurrently executed threads) respects no-write assumptions is captured by requiring that the relation is closed under globally consistent field assignments and object creations.

If two local configurations are related by such a bisimulation then their heaps are due to condition (1) indistinguishable to an low observer.

One property of low bisimulations is that if two configurations are related then only the low parts of the memory need to be indistinguishable, high parts can be chosen arbitrarily. This is useful to reflect that in the case of a branching instruction on a high guard, both branches are considered. In our case register sets influence the control flow inherently, because most instructions of the DVM use them. Therefore we require indistinguishable register sets with respect to register domain assignments that capture in which registers secrets are kept. Condition (2) requires that all register sets that are contained in the call stacks are pair-wise indistinguishable and that the call stacks have equal height.

Condition (3a) of the definition captures the aspect that condition (1) and (2) shall be preserved under execution. If the first configuration cannot do a step (i.e. the thread has terminated), then the condition is satisfied trivially. If it is possible to do a step, the second configuration needs to do a step such that both resulting configurations are related by $R^{rts'}_{\beta'}$. The annotations to the relations are used to reflect changes on the heap and the register sets. To reflect the possible changes to the register sets we use the function *update* to create a new register domain assignment out of the old one. We use the event α to do this. The event γ can be different to α, the only requirement is, that the call stacks of the resulting configurations are indistinguishable with respect to register domain assignment created from α. The changes on the heap are captured by extending β in a save way. This expansion of β captures the creation of new low objects.

Condition (3b) covers the case of dynamic thread creation. Thread creation is only allowed if both configurations spawn a new thread, thus the number of threads does not depend on a high value. With regard to compositionality it is important that no assumptions are active when creating a new thread, because the new thread would have to fulfill all guarantees for all made assumptions, as threads can be created everywhere in the program.

Condition (4) is a symmetry condition. Since heap indistinguishability is defined with respect to a partial bijection β, the symmetric case needs to be defined with respect to the inverse of β.

Due to condition (4) and condition (3a) we require that either both configurations can do a step or none. This ensures that no timing leaks can occur in the sequential case. However due to concurrency and the presence of synchronization we have to ensure that the monitor states are equal. Otherwise it would be possible to create a program like the one in the following example.

Example 7. *For an instruction set* $\mathcal{I} = \{\texttt{if}\ l\ f\ i, \texttt{goto}\ i, \texttt{put}\ n\ f, \texttt{enter}\ l, \texttt{exit}\ l, \texttt{nop} : f \in \mathcal{F}, l \in \mathcal{L}, i, n \in \mathbb{N}\}$ *we define the example program* $E = (C, F, M)$*, where*

$$C = \{c\}$$
$$F = \{h, l\}$$
$$M = \left\{ \left(m_1, i \mapsto \begin{cases} \texttt{put}\ 0\ l_0\ l & i = 1 \\ \texttt{if}\ l_0\ h\ 5 & i = 1 \\ \texttt{enter}\ l_0 & i = 2 \\ \texttt{exit}\ l_0 & i = 3 \\ \texttt{goto}\ 7 & i = 4 \\ \texttt{nop} & i = 5 \\ \texttt{nop} & i = 6 \\ \texttt{put}\ 1\ l_0\ l & i = 7 \end{cases} \right), \left(m_2, i \mapsto \begin{cases} \texttt{put}\ 1\ l_0\ l & i = 1 \\ \texttt{enter}\ l_0 & i = 2 \\ \texttt{nop} & i = 3 \\ \texttt{exit}\ l_0 & i = 4 \\ \texttt{put}\ 0\ l_0\ l & i = 5 \end{cases} \right) \right\}$$

The instructions `enter` *and* `exit` *model the acquiring and releasing of monitors. The semantics of all other instructions is standard.*

The execution of the program E *leads to distinguishable states when starting with the heaps* $h_1 = \{(l_0, (c, \{(h, 0)\}))\}$ *and* $h_2 = \{(l_0, (c, \{(h, 1)\}))\}$ *under a round-robin scheduler that picks method* m_1 *first.*

Using the definition of strong low bisimulation modulo modes, we define when two programs are secure.

Definition 3.12. *Two programs* P *and* Q *starting the execution in method* m *and* n *are indistinguishable for the mode state mds (denoted by* $(P, m) \sim_{low}^{mds} (Q, n)$*) if for all* $h_1, h_2 \in \mathcal{H}$ *such that there exists some safe bijection* $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ *and* $h_1 \sim_\beta^{mds} h_2$ *and for all* $rs_1, rs_2 \in \mathcal{R} \to \mathcal{V}$ *such that* $rs_1 \sim_\beta^{rt_0} rs_2$ *and the following holds*

$$(\langle 1, m, rs_1 \rangle, mon_0, mds, h_1)_P \approx_\beta^{rts_0} (\langle 1, n, rs_2 \rangle, mon_0, mds, h_2)_Q$$

A program P *starting in method* m *is secure if* $(P, m) \sim_{low}^{mds_0} (P, m)$*.*

3.6 Compositionality

This section introduces a reference security property that captures secure information flow modulo modes for multithreaded programs and claims compositionality for multithreaded programs that use assumptions and guarantees in a sound way.

As in [MSS11] we define security for multithreaded programs by a non-interference-like security property. Intuitively a multithreaded program is secure, if after arbitrary many execution steps the values of low fields are independent of secrets if no thread makes a no-read assumptions for them.

Definition 3.13. *Two program P in the methods $m_1, \ldots, m_n$ is secure, if for all $h_1, h_2 \in \mathcal{H}$ and for some safe $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ such that $h_1 \sim_{\beta}^{mds_0} h_2$ and for all $rs_1, rs_2 \in \mathcal{R} \to \mathcal{V}$ such that $rs_1 \sim_{\beta}^{rt_0} rs_2$ and*

$$([(\langle 1, m_1, rs_1\rangle, mon_0, mds_0), \ldots, (\langle 1, m_n, rs_1\rangle, mon_0, mds_0)], h_1)_P \rightsquigarrow^k$$
$$([(cs_1, mon_1, mds_1), \ldots, (cs_n, mon_n, mds_n), \ldots, (cs_{n+i}, mon_{n+i}, mds_{n+i})], h_1')_P$$

for some $k, i \in \mathbb{N}$ then there exist call stacks $cs_1', \ldots cs_{n+i}'$, a heap h_2' and $\beta' \supseteq \beta$ such that

$$([(\langle 1, m_1, rs_2\rangle, mon_0, mds_0), \ldots, (\langle 1, m_n, rs_2\rangle, mon_0, mds_0)], h_1)_P \rightsquigarrow^k$$
$$([(cs_1', mon_1, mds_1), \ldots, (cs_n', mon_n, mds_n), \ldots, (cs_{n+i}', mon_{n+i}, mds_{n+i})], h_2')_P$$

and β' is safe for h_1' and h_2', and $h_2 \sim_{\beta'}^{mds} h_2'$ for all $mds \in \{mds_1, \ldots, mds_n, \ldots, mds_{n+i}\}$

If the program starts with heaps and register sets that are indistinguishable for a low observer the heaps remain indistinguishable modulo no-read assumptions after k steps with respect to a safe bijection β' that captures the creation of objects. This needs to hold true, even if new threads are created.

Our next goal is to define what it means for a multithreaded program to use modes in a sound way. As in [MSS11] we split this definition up into two parts. First we ensure that "globally" every assumption has corresponding guarantees for all concurrently executed threads (*globally sound*). Then we ensure that a thread does not violate the guarantees it provides (*locally sound*). If a global configuration is globally sound and all contained local configurations are locally sound, we say that this global configuration ensures a sound use of modes.

The first step to define global soundness is to define when a tuple of mode states provides compatible modes (i.e. every assumption is matched by corresponding guarantees).

Definition 3.14. *A mode state tuple $(mds_1, \ldots, mds_n)$ has compatible modes if for all $i \in \{1, \ldots, n\}$ and for all $f \in \mathcal{F}$ the following holds*

1. $f \in mds_i(asm\text{-}noread) \implies \forall j \neq i : f \in mds_j(guar\text{-}noread)$

2. $f \in mds_i(asm\text{-}nowrite) \implies \forall j \neq i : f \in mds_j(guar\text{-}nowrite)$

If some mode states makes an assumption about some field f then this assumption must be matched by a corresponding guarantee in all other mode states. However the mode state itself need not provide such a guarantee.

As we want to ensure that mode states of global configurations are consistent during the entire execution, we define in the following all mode state tuples that are reachable form a global configuration.

Definition 3.15. *The set of mode state tuples that are reachable from a global configuration $gc \in Conf$ is defined as*

$$\{(mds_1, \ldots, mds_n) : \exists cs_1, \ldots, cs_n \in \mathcal{CS}, mon_1, \ldots, mon_n \in (\mathcal{L} \to \mathbb{N}), h \in \mathcal{H} \,.\\ gc \rightsquigarrow^* ((cs_1, mon_1, mds_1), \ldots, (cs_n, mon_n, mds_n), h)\}$$

We start in the global configuration gc and do arbitrary many steps ($\rightsquigarrow^*$ denotes the transitive closure of $\rightsquigarrow$). The mode states of every possible execution step are reachable from this global configuration. This includes the mode states of newly created threads.

We say that a global configuration ensures a *globally sound use of modes* if all mode state tuples that are reachable have compatible modes.

Now we define what it means for a local configuration to comply with the given guarantees. Therefore we define what it means that an a local configuration does not read respectively write to a field.

Definition 3.16. *A local configuration c does not read a field f if whenever $c = (cs, mon, mds, h) \overset{\alpha}{\rightarrow} (cs', mon', mds', h')$, then one of the following conditions holds:*

1. $\forall v \in \mathcal{V} : \forall l \in dom(h) : \exists \gamma \in E : f \in dom(\pi_2(h(l)))$
   $\implies (cs, mon, mds, h(l)[f \mapsto v]) \overset{\gamma}{\rightarrow} (cs', mon', mds', h'(l)[f \mapsto v])$
2. $\forall v \in \mathcal{V} : \forall l \in dom(h) : \exists \gamma \in E : f \in dom(\pi_2(h(l)))$
   $\implies (cs, mon, mds, h(l)[f \mapsto v]) \overset{\gamma}{\rightarrow} (cs', mon', mds', h')$

This definition approximates what it means to *no read* a field. Either the field can be left unmodified or can be modified, where the rest of the heap and the register sets stay the same. Thus the value of the field cannot be written into an other field or register. Since our notion of modes is not object sensitive, we have to ensure this for all occurrences of the field.

An local configuration does not modify a field f, if the values of the field f in all instances that contain f remain the same.

Definition 3.17. *An local configuration c does not modify a field f if whenever $c = (cs, mon, mds, h) \overset{\alpha}{\rightarrow} (cs', mon', mds', h')$ and then for all $l \in \mathcal{L}$*

$$(l \in dom(h) \wedge f \in dom(\pi_2(h(l))) \implies \pi_2(h(l))(f) = \pi_2(h'(l))(f)$$

If we can do an execution step from the local configuration, then the values of all fields f at all locations l that contain that field must be equal. We do not have to ensure that the location and the field is contained in the resulting heap, because objects cannot be destroyed and fields not be uninitialized.

Now we define all local configurations that are potentially reachable from an local configurations. This allows us to ensure that, no matter how the environment changes the heap, the reachable local configurations always adhere to their guarantees.

In the definition of the reachable local configurations we exploit, like in [MSS11], the no-write assumptions to reduce the number of possibly reachable local configurations. Of course this approximation is only sound if the environment is globally sound with respect to these assumptions.

Definition 3.18. *The set $lReach(lc)$ of local configurations that are potentially reachable form the local configuration lc is inductively defined as follows:*

1. $lc \in lReach(lc)$
2. $\forall lc' \in lReach(lc) : \forall lc'' : lc' \overset{\alpha}{\rightarrow} lc'' \Rightarrow lc'' \in lReach(lc)$
3. $\forall (cs', mon', mds', h') \in lReach(lc) :$
   $\forall h'' \in \mathcal{H} : (\forall f \in mds'(asm\text{-}nowrite) :$
   $(\forall l \in dom(h') : \pi_2(h'(l))(f) = \pi_2(h''(l))(f)) \wedge$
   $(\forall l \in dom(h'') : l \notin dom(h') \implies f \notin dom(\pi_2(h''(l)))))$
   $\implies (cs', mon', mds', h'') \in lReach(lc)$

Local configurations that are reachable from a local configuration lc are of course the configuration lc itself (condition (1)) and all configurations that result after one step in a configuration that is already reachable (condition (2)). Condition (3) covers the heap modifications of the environment. Here we take advantage out of the no-write assumptions. Namely, all configurations that are reachable are also reachable with arbitrary heap modifications, except for the fields with no-write assumptions. These modifications include fields modifications and object creation.

Now we can combine these notions and define what it means that a local configuration does not violate the guarantees it provides.

Definition 3.19. *A local configuration lc ensures a locally sound use of modes if for all $lc' = (cs', mon', mds', h') \in lReach(lc)$ and all $f \in \mathcal{F}$ the following conditions hold:*

1. $f \in \pi_{mds}(lc')(guar\text{-}noread) \Rightarrow lc'$ *does not read* f
2. $f \in \pi_{mds}(lc')(guar\text{-}nowrite) \Rightarrow lc'$ *does not modify* f

where $\pi_{mds}((cs, mon, mds, h)) = mds$.

A local configuration ensures a locally sound use of modes, if the configuration itself and all reachable local configurations adhere to the guarantees they provide. I.e. do not read fields with no-read guarantees and do not write fields with no-write guarantees.

Now we combine local and global soundness to define sound use of modes for a global configuration.

Definition 3.20. *The global configuration $((cs_1, mon_1, mds_1), \ldots, (cs_n, mon_n, mds_n), h)$ ensures a sound use of modes if it ensures a globally sound use of modes and if each local configuration (cs_i, mon_i, mds_i, h) ensures a locally sound use of modes.*

A global configurations uses modes in a sound way, if it ensures a globally sound use of modes (i.e. all assumptions are matched by corresponding guarantees) and all threads ensure a locally sound use of modes (i.e. they do not violate guarantees that they provide).

Now we can claim the compositionality of the security property.

Claim 1 (Compositionality)**.** *Let $(P, m_1), \ldots, (P, m_k)$ be secure programs such that $([(\langle 1, m_1, rs\rangle, mon_0, mds_0), \ldots, (\langle 1, m_k, rs\rangle, mon_0, mds_0)], h)$ ensures a sound use of modes for every heap h and register set rs. Then the multithreaded program $(P, m_1, \ldots, m_k)$ is secure.*

Chapter 4

Instances of the Transition System

4.1 Sequential Subset of Dalvik

In this section we define the first instantiation of the ATS. This instantiation models a sequential subset of the DVM.

Instructions for basic register operations, branching, operations on integers, method invocations and operations on objects are included. Assumptions and guarantees on fields can be acquired and released on the execution of every instruction. This is realized with annotations at instructions. The possible annotations are defined in the following.

Definition 4.1. *The set of annotations for acquiring and releasing modes is define as* $Ann = \{acq(m, f), rel(m, f) : m \in Mod, f \in \mathcal{F}\}$.

An annotations consists of a type, a mode and a field identifier. The type of the annotations is either *acq* for acquiring a mode for a field identifier or *rel* for releasing a mode for a field identifier.

Definition 4.2. *The set of sequential instructions is defined as* $\mathcal{I}_{\text{seq}} = I \cup \{//a//i : i \in I, a \in Ann\}$, *where*

$$\begin{aligned} I = \{&\texttt{nop},\ \texttt{move}\ r_1\ r_2,\ \texttt{moveresult}\ r,\ \texttt{returnvoid},\ \texttt{return}\ r,\ \texttt{const}\ r\ z,\\ &\texttt{instanceof}\ r_1\ r_2\ c,\ \texttt{newinstance}\ r_1\ c,\ \texttt{goto}\ j,\ \texttt{compare}\ r_1\ r_2\ r_3,\\ &\texttt{ifeq}\ r_1\ r_2\ j,\ \texttt{iget}\ r_1\ r_2\ f,\ \texttt{iput}\ r_1\ r_2\ f,\ \texttt{invoke}\ r_1 \ldots r_n\ m,\ \texttt{neg}\ r_1\ r_2,\\ &\texttt{add}\ r_1\ r_2\ r_3 : z \in \mathbb{Z}, j \in \mathcal{PP}, f \in \mathcal{F}, m \in \mathcal{M}_{\text{id}}, r \in \mathcal{R}, (r_1, \ldots, r_n) \in \mathcal{R}^n\} \end{aligned}$$

The set of instructions contains the instructions with and without annotations. As byte-code languages are unstructured we do not put additional structure into the set of instructions.

In the following we define the operational semantics of the instructions in $\mathcal{I}_{\text{seq}}$. It is important that the annotations do not have effects on the number of execution steps. Therefore we define two rules for generating elements of the local transition relation. The rule Ranno removes the annotation from the instruction, modifies the mode state accordingly to the annotation and executes the instruction according to the auxiliary judgment $(cs, mon, h)_M \overset{\alpha}{\rightharpoonup} (cs', mon', h')_M$. This judgment models the modifications of local configurations without mode states. The rule Rins is responsible for instructions without annotations. It does not modify the mode state and executes the instruction according to the same auxiliary judgment as Ranno. Thus there is no timing difference if an instruction is annotated or not.

Rann

$$\frac{\begin{array}{c} M(m)(i) = //ann//ins \\ M' = M(m)[i \mapsto ins] \qquad ((i, m, rs) :: cs, mon, h) \overset{\alpha}{\rightharpoonup} (cs', mon', h') \end{array}}{((i, m, rs) :: cs, mon, mds, h)_{C,F,M} \overset{\alpha}{\rightharpoonup} (cs', mon', \mathit{update\text{-}mds}(mds, ann), h')_{C,F,M}}$$

This rule is responsible for modifying the mode state according to the annotation on an instruction. The function $update - mds$ is defined as in [MSS11].

$$\mathit{update\text{-}mds}(mds, ann) := \begin{cases} mds[m \mapsto mds(m) \cup \{f\}] & ann = acq(m, f) \\ mds[m \mapsto mds(m) \setminus \{f\}] & ann = rel(m, f) \end{cases}$$

To supply the auxiliary judgment with the plain instruction the set of methods is temporarily modified.

Rins

$$\frac{M(m)(i) = ins \qquad ((i, m, rs) :: cs, mon, h) \overset{\alpha}{\rightharpoonup} (cs', mon', h')}{((i, m, rs) :: cs, mon, mds, h)_{C,F,M} \overset{\alpha}{\rightharpoonup} (cs', mon', mds, h')_{C,F,M}}$$

The rule Rins just passes the values to the auxiliary judgment as described before.

Now we define the rules for the auxiliary judgment. All rules are as close as possible to the actual semantics of the DVM. All deviations are described at the instruction.

Rnop

$$\frac{M(m)(i) = \mathtt{nop}}{((i, m, rs) :: cs, mon, h)_M \overset{\varepsilon}{\rightharpoonup} ((i+1, m, rs) :: cs, mon, h)_M}$$

The `nop` instruction moves the program pointer to the next instruction and leaves the rest of the configuration unchanged. Therefore is does not emit an event. The security property captures all effects of this instructions adequately, because it enforces equal timing behavior with condition (3).

Rmove

$$\frac{M(m)(i) = \mathtt{move}\ r_1\ r_2 \qquad r_2 \in dom(rs) \qquad rs' = rs[r_1 \mapsto rs(r_2)]}{((i, m, rs) :: cs, mon, h)_M \overset{r_2 \triangleright r_1}{\rightharpoonup} ((i+1, m, rs') :: cs, mon, h)_M}$$

The `move` instruction copies the value stored in register r_2 into the register r_1 in the top most register set. To reflect this change in the security property, the event $r_2 \triangleright r_1$ is emitted. Due to this event the security domain of register identifier r_1 is changed to the security domain of the register identifier r_2 (due to the *update* function). This is adequate because r_1 is not directly accessible for a *low* observer and does now contain the value of r_2. The security domain of the register is updated to track which values are "under the control" (i.e. public) of the attacker, i.e. have a *low* origin.

$$\textsc{Rmoveres}\quad \frac{M(m)(i) = \texttt{moveresult}\ r \qquad v_{res} \in dom(rs) \qquad rs' = rs[r \mapsto rs(v_{res})]}{((i, m, rs) :: cs, mon, h)_M \overset{v_{res} \triangleright r}{\rightharpoonup} ((i+1, m, rs') :: cs, mon, h)_M}$$

The moveresult instruction is very similar to the move instruction. The only difference is, that it copies the value stored in the register v_{res} into the register r. In the DVM this instruction can only be executed if it follows directly after an invoke instruction, to keep it simple we do not enforce this. Thus we could access return values twice, which should not cause any problems. As the rule for move is only applicable if at the register identifier is actually stored a value, we cannot access v_{res} before the first method call.

$$\textsc{Rreturnvoid}\quad \frac{M(m)(i) = \texttt{returnvoid}}{((i, m, rs) :: cs, mon, h)_M \overset{\triangledown}{\rightharpoonup} (cs, mon, h)_M}$$

The returnvoid instruction finishes a method call without providing a return value to the caller. Thus it removes the top element of the call stack to resume with the execution of the caller. This effect is reflected in the security property by observing the $\triangledown$ event, that is emitted by returnvoid. On this event the *update* function removes the top element of the stack of register domain assignments. This corresponds to the removal of the top element of the call stack.

$$\textsc{Rreturn}\quad \frac{M(m)(i) = \texttt{return}\ r \qquad r \in dom(rs) \qquad rs'' = rs'[v_{res} \mapsto rs(r)]}{((i, m, rs) :: (i', m', rs') :: cs, mon, h)_M \overset{\triangledown r}{\rightharpoonup} ((i', m', rs'') :: cs, mon, h)_M}$$

The return instruction is similar to the returnvoid instruction. The only difference is, that the v_{res} register of the caller is updated with the value of register r of the called method. This difference is captured by modifying the security domain of the v_{res} register of the caller by the $\triangledown r$ event.

$$\textsc{Rconst}\quad \frac{M(m)(i) = \texttt{const}\ r\ z \qquad rs' = rs[r \mapsto z]}{((i, m, rs) :: cs, mon, h)_M \overset{\triangleright r}{\rightharpoonup} ((i+1, m, rs') :: cs, mon, h)_M}$$

The const instruction copies an integer into the register r. This effect is captured by the event $\triangleright r$. The *update* function assigns on this event *low* to the register identifier r, because the value z is a constant and therefore visible in the source code, to which the attacker has access.

RINSTANCEOF-TRUE

$$\frac{\begin{array}{c}M(m)(i) = \texttt{instanceof}\ r_1\ r_2\ c \\ r_2 \in dom(rs) \qquad rs(r_2) \in dom(h) \qquad \pi_1(h(rs(r_2))) = c \qquad rs' = rs[r_1 \to 1]\end{array}}{((i, m, rs) :: cs, mon, h)_M \overset{r_2 \triangleright r}{\twoheadrightarrow} ((i + 1, m, rs') :: cs, mon, h)_M}$$

RINSTANCEOF-FALSE

$$\frac{\begin{array}{c}M(m)(i) = \texttt{instanceof}\ r_1\ r_2\ c \\ rs_2 \in dom(rs) \qquad rs(r_2) \in dom(h) \qquad \pi_1(h(rs(r_2))) \neq c \qquad rs' = rs[r_1 \to 0]\end{array}}{((i, m, rs) :: cs, mon, h)_M \overset{r_2 \triangleright r}{\twoheadrightarrow} ((i + 1, m, rs') :: cs, mon, h)_M}$$

The instruction instanceof copies the number 1 into the register r_1 if the object at location $rs(r_2)$ is an instance of class c, otherwise it copies the number 0 into register r_1. If the object at $rs(r_2)$ is a invisible object, its existence shall not be observable for a low observer. Thus we ensure with the event $r_2 \triangleright r$ that if r_2 has a high security domain r will also have a high security domain. Notice that if the object at $rs(r_2)$ is invisible, then the security domain of the register identifier r_2 is high. Otherwise a low observer could distinguish those register sets as the locations of invisible objects cannot be associated with a safe bijection β.

RNEWINSTANCE

$$\frac{\begin{array}{c}M(m)(i) = \texttt{newinstance}\ r\ c \\ l \notin dom(h) \qquad h' = h[l \mapsto (c, \emptyset)] \qquad rs' = rs[r \mapsto l]\end{array}}{((i, m, rs) :: cs, mon, h)_M \overset{c \triangleright r}{\twoheadrightarrow} ((i + 1, m, rs') :: cs, mon, h')_M}$$

A new instance of a class c is created by the newinstance instruction. The object is saved at an arbitrary free location on the heap. The new object consists of a class identifier and an empty function. This captures the behavior of the new-instance instruction in the DVM, which creates an empty object without initializing the fields. The location of the new object is copied to the register identifier r. Thus the instruction has an effect on both the current register set and the heap. The event that is send, describes a flow from c to r. This is due to the fact that the instantiation of high objects would result in distinguishable register sets if the location is saved in a low register, since those locations cannot be associated with a safe bijection β.

RGOTO

$$\frac{M(m)(i) = \texttt{goto}\ j \qquad j \in dom(M(m))}{((i, m, rs) :: cs, mon, h)_M \overset{\varepsilon}{\twoheadrightarrow} ((j, m, rs) :: cs, mon, h)_M}$$

The goto instruction jumps directly to the program point j and does nothing else. This is captured by the security property by condition (3), that requires the existence of a step in the other configuration that preserves the indistinguishability.

RCOMPARE

$$\frac{\begin{array}{c}M(m)(i) = \texttt{compare}\ r_1\ r_2\ r_3 \\ r_2, r_3 \in dom(rs) \qquad rs' = rs[r_1 \mapsto cmp(rs(r_2), rs(r_3))]\end{array}}{((i, m, rs) :: cs, mon, h)_M \overset{r_2 r_3 \triangleright r_1}{\twoheadrightarrow} ((i + 1, m, rs') :: cs, mon, h)_M}$$

The `compare` instruction compares the values of the registers r_2 and r_2 and writes a value, depending on the result of the comparison, into the register r_1. The comparison and the computation of that value is done with the following function

$$cmp(x,y) = \begin{cases} 1 & x > y \\ 0 & x = y \\ -1 & x < y \end{cases}$$

This behavior is similar to the behavior of the DVM, as the dvm writes a positive value in the register if the first value is greater than the second, zero if they are equal and a negative value if the first value is smaller than the second. As the value at register identifier r_1 is now influenced by two register identifiers, the new security domain for register r_1 is approximated by *update* with the least upper bound of the security domains of the register identifiers r_2 and r_3. This ensures that the security domain of register r_1 is high, if at least one of r_2 and r_3 has a high security domain.

Rifeq-true

$$\frac{\begin{array}{c} M(m)(i) = \texttt{ifeq}\, r_1\ r_2\ j \\ r_1, r_2 \in dom(rs) \qquad rs(r_1), rs(r_2) \in \mathbb{Z} \qquad rs(r_1) = rs(r_2) \end{array}}{((i,m,rs) :: cs, mon, h)_M \overset{\varepsilon}{\rightharpoonup} ((j,m,rs) :: cs, mon, h)_M}$$

Rifeq-false

$$\frac{\begin{array}{c} M(m)(i) = \texttt{ifeq}\, r_1\ r_2\ j \\ r_1, r_2 \in dom(rs) \qquad rs(r_1), rs(r_2) \in \mathbb{Z} \qquad rs(r_1) \neq rs(r_2) \end{array}}{((i,m,rs) :: cs, mon, h)_M \overset{\varepsilon}{\rightharpoonup} ((i+1,m,rs) :: cs, mon, h)_M}$$

The `ifeq` instruction jumps to j if the values at the register identifiers r_1 and r_2 are integers and equal. Otherwise the execution proceeds with the next instruction. If at least one of the registers r_1 and r_2 has a *high* security domain then according to condition (2) of the security property *high* values can differ in indistinguishable register sets and thus by condition (3) of the bisimulation the execution of both branches have to lead to indistinguishable memories.

Riget

$$\frac{\begin{array}{c} M(m)(i) = \texttt{iget}\, r_1\ r_2\ f \qquad r_2 \in dom(rs) \\ f \in dom(hs(rs(r_2))) \qquad rs(r_2) \in dom(h) \qquad rs' = rs[r_1 \mapsto \pi_2(h(rs(r_2)))(f)] \end{array}}{((i,m,rs) :: cs, mon, h)_M \overset{r_2 f \triangleright r_1}{\rightharpoonup} ((i+1,m,rs') :: cs, mon, h)_M}$$

The `iget` instruction fetches the value of a field f at the location that is saved at register identifier r_2 and copies this value to the register identifier r_1. Thus this method has only an effect on the register set. The new security domain for r_1 that is computed by *update* is the least upper bound of the security domain of the register r_2 and the field f according to the emitted event. We take the security domain of the field into account to reflect its security domain in the register domain assignment.

Riput

$$\frac{\begin{array}{c} M(m)(i) = \texttt{iput}\ r_1\ r_2\ f \\ r_1, r_2 \in dom(rs) \qquad rs(r_2) \in dom(h) \qquad c = \pi_1(h(rs(r_2))) \\ f \in fields(c) \qquad fs = \pi_2(h(rs(r_2)))[f \mapsto rs(r_1)] \qquad h' = h[rs(r_2) \mapsto (c, fs)] \end{array}}{((i, m, rs) :: cs, mon, h)_M \overset{\varepsilon}{\rightharpoonup} ((i+1, m, rs) :: cs, mon, h)_M}$$

The iput instruction moves the value of register r_1 into the field f at the location saved at the register identifier r_2. This is only possible if the field actually belongs the class of the object at $rs(r_2)$. This instruction only modifies the heap. Condition (1) of the security property ensures that no leaks into low fields of visible objects are possible.

Rinvoke

$$\frac{\begin{array}{c} M(m)(i) = \texttt{invoke}\ r_1 \dots r_n\ m' \\ r_1, \dots, r_n \in dom(rs) \qquad rs' = \{(v_i, rs(r_{i+1})) : i \in \mathbb{N}, i < n\} \end{array}}{((i, m, rs) :: cs, mon, h)_M \overset{r_1 \dots r_n}{\rightharpoonup} ((1, m', rs') :: (i+1, m, rs) :: cs, mon, h)_M}$$

The invoke instruction invokes the method m by supplying the values of the registers $r_1, \dots, r_n$ as parameters. The parameters are copied into the first locations of a new register set, that is put on top of the call stack. In the DVM the registers are copied into the last n registers (cf. [Pro07]), since we have an infinite supply of registers we use the first n registers. The program point in the stack frame of the caller is incremented by one, to ensure that after the return instruction the execution continues in the correct order. The event that is send, ensures that a new register domain assignment for the register set of the called method is created, depending on the method parameters. All other register identifiers are assigned the security domain *low*, to increase the precision of the approximations and because the values are constants (i.e. undefined).

Rneg

$$\frac{M(m)(i) = \texttt{neg}\ r_1\ r_2 \qquad r_2 \in dom(rs) \qquad rs' = rs[r_1 \mapsto -rs(r_2)]}{((i, m, rs) :: cs, mon, h)_M \overset{r_2 \triangleright r_1}{\rightharpoonup} ((i+1, m, rs') :: cs, mon, h)_M}$$

The instruction neg inverts the sign of the number saved at register identifier r_2 and saves it at register identifier r_1. Apart from that it behaves exactly like the move instruction.

Radd

$$\frac{\begin{array}{c} M(m)(i) = \texttt{add}\ r_1\ r_2\ r_3 \\ r_2, r_3 \in dom(rs) \qquad rs' = rs[r_1 \mapsto rs(r_2) + rs(r_3)] \end{array}}{((i, m, rs) :: cs, mon, h)_M \overset{r_2 r_3 \triangleright r_1}{\rightharpoonup} ((i+1, m, rs') :: cs, mon, h)_M}$$

The instruction add is shown here to present some other binary operation on integers. It adds the values at the register identifiers r_2 and r_3 and saves the result in the register r_1. The effect on the security property is the same as for the cmp instruction.

Now we define a the transition system for the sequential subset of the DVM.

Definition 4.3. $\mathrm{DVM}_{\mathrm{seq}}(conf_0, \mathrm{P}) = ATS(conf_0, \mathrm{P}, \mathbb{Z}, \mathcal{I}_{\mathrm{seq}}, \dashrightarrow)$ *is a sequential subset of the DVM for some program* P *and some initial configuration* $conf_0$ *with* $|\, conf_0 \,| = 1$.

The initial configurations and the program is left parametric to be able to execute arbitrary programs. The restriction on the initial state and the absence of rules for thread creation ensures that only sequential programs are executed. The set of values are integers and the instructions and their semantics are used as defined above.

The following lemma shows that a $\mathrm{DVM}_{\mathrm{seq}}$ program that writes data from low to high fields is secure.

Lemma 1. *The program* $P = (C, F, M)$ *starting in method* m *with* $C = \{c\}$, $F = \{l, h\}$, $\mathcal{M}_{\mathrm{id}} = \{m\}$ *and*

$$M = \left\{ m, i \mapsto \begin{cases} iget\ v_2\ v_1\ l & i = 1 \\ iput\ v_2\ v_1\ h & i = 2 \\ returnvoid & i = 3 \end{cases} \right\}$$

is secure with respect to the domain assignment $ft(l) = low$ *and* $ft(h) = high$.

Proof. We show that a strong bisimulation modulo modes R exists such that

$$((\langle 1, m, rs_1\rangle, mon_0, mds_0, h_1)_P R_\beta^{rts_0} (\langle 1, m, rs'_1\rangle, mon_0, mds_0, h'_1))_P$$

holds for all $h_1, h'_1 \in \mathcal{H}$ such that $h_1 \sim_\beta^{mds_0} h'_1$ and for all $rs_1, rs'_1 \in \mathcal{R}$ such that $rs_1 \sim_\beta^{rt_0} rs'_1$ for some safe partial bijection β. From the definition of $\approx$ it then follows that

$$((\langle 1, m, rs_1\rangle, mon_0, mds_0, h_1)_P \approx_\beta^{rts_0} (\langle 1, m, rs'_1\rangle, mon_0, mds_0, h'_1))_P$$

Let

$$\begin{aligned}
R' := \{ & (((\langle 1, m, rs_1\rangle, mon_0, mds_0, h_1), rts_0, \beta, (\langle 1, m, rs'_1\rangle, mon_0, mds_0, h'_1)), \\
& (((\langle 1, m, rs_1\rangle, mon_0, mds_0, h_2), rts_0, \beta', (\langle 1, m, rs'_1\rangle, mon_0, mds_0, h'_2)), \\
& (((\langle 2, m, rs_2\rangle, mon_0, mds_0, h_1), rts_0, \beta, (\langle 2, m, rs'_2\rangle, mon_0, mds_0, h'_1)), \\
& (((\langle 2, m, rs_2\rangle, mon_0, mds_0, h_2), rts_0, \beta', (\langle 2, m, rs'_2\rangle, mon_0, mds_0, h'_2)), \\
& (((\langle 3, m, rs_2\rangle, mon_0, mds_0, h_1), rts_0, \beta, (\langle 3, m, rs'_2\rangle, mon_0, mds_0, h'_1)), \\
& (((\langle 3, m, rs_2\rangle, mon_0, mds_0, h_2), rts_0, \beta', (\langle 3, m, rs'_2\rangle, mon_0, mds_0, h'_2)), \\
& (((\langle\rangle, mon_0, mds, h_1), \langle\rangle, \beta, (\langle\rangle, mon_0, mds, h'_1)) \\
& (((\langle\rangle, mon_0, mds, h_2), \langle\rangle, \beta', (\langle\rangle, mon_0, mds, h'_2)) : \\
& h_1 \sim_\beta^{mds_0} h'_1, h_2 \sim_{\beta'}^{mds_0} h'_2, h_2 \supseteq h_1, h'_2 \supseteq h'_1, \beta' \supseteq \beta, \\
& rs_1 \sim_\beta^{rt_0} rs'_1, \\
& rs_2 = rs_1[v_2 \mapsto h_1(rs_1(v_1))(fl)], \\
& rs'_2 = rs'_1[v_2 \mapsto h'_1(rs'_1(v_1))(fl)]\} \\
R := R' \cup \{ & ((cs_1, mon, mds, h_1), rts, \beta^{-1}, (cs_2, mon, mds, h_2)) : \\
& ((cs_2, mon, mds, h_2), rts, \beta, (cs_1, mon, mds, h_1)) \in R'\}
\end{aligned}$$

We now show that R is closed under globally consistent field modifications. By the definition of R we have that no assumption is made for any field. Furthermore we know that all high fields in h_1 (respectively h_2) can be modified arbitrarily while remaining in R, because R relates all configurations such that $h_1 \sim_{\beta}^{mds_0} h'_1$ (respectively $h_2 \sim_{\beta'}^{mds_0} h'_2$). Thus condition (1) holds. The same argument applies for condition (2), since this condition are describes all changes to fields such that the heaps remain indistinguishable. We cover these changes, because we include all configurations with indistinguishable heaps.

The next step is to show that R is closed under globally consistent object creations. We still know that no assumptions are made in any configuration contained in R. The fist condition is satisfied, as $h_1 \sim_{\beta}^{mds_0} h'_1$ holds for all heaps with arbitrary allocations of high objects. As R does not contain distinguishable heaps condition (2) is also satisfied.

To show that R is a strong low bisimulation modulo modes, we make a case distinction on the elements or R. The symmetry condition (4) is satisfied by definition of R. Since no instruction sends the event $cs, r_1 \ldots r_n$ condition (3b) is always true.

Case $((\langle 1, m, rs_1\rangle, mon_0, mds_0, h_1) R_{\beta}^{rts_0} (\langle 1, m, rs'_1\rangle, mon_0, mds_0, h'_1))$**:** Conditions (1) and (2) are satisfied by definition of R, thus we have to show (3).

From the rules Rins and Riget of the operational semantics we obtain

$$(\langle 1, m, rs_1\rangle, mon_0, mds_0, h_1) \overset{v_1 l \triangleright v_2}{\rightharpoonup} (\langle 2, m, rs_2\rangle, mon_0, mds_0, h_1)$$

where $rs_2 = rs_1[v_2 \mapsto h_1(rs_1(v_1))(l)]$ if $v_1 \in dom(rs)$ and $rs_1(v_1) \in dom(h_1)$. Otherwise no execution step is possible and condition (3) is trivially satisfied.

If this execution step is possible, an execution step in the related configuration must be possible too. From the rules Rins and Riget of the operational semantics we obtain

$$(\langle 1, m, rs'_1\rangle, mon_0, mds_0, h'_1) \overset{v_1 l \triangleright v_2}{\rightharpoonup} (\langle 2, m, rs'_2\rangle, mon_0, mds_0, h'_1)$$

where $rs'_2 = rs'_1[v_2 \mapsto h'_1(rs'_1(v_1))(l)]$. The rule Riget is applicable because from $rs_1 \sim_{\beta}^{rt_0} rs'_1$, $v_1 \in dom(rs)$, $rs_1(v_1) \in dom(h_1)$ and the safety of β, we have $rs'_1(v_1) \in dom(h_2)$. By definition of R and $rts_0 = update(rts_0, v_1 l \triangleright v_2)$

$$((\langle 2, m, rs_2\rangle, mon_0, mds_0, h_1) R_{\beta}^{rts_0} (\langle 2, m, rs'_2\rangle, mon_0, mds_0, h'_1))$$

holds. Thus Condition (3) is satisfied.

Case $((\langle 2, m, rs_2\rangle, mon_0, mds_0, h_1) R_{\beta}^{rts_0} (\langle 2, m, rs'_2\rangle, mon_0, mds_0, h'_1))$**:** To show $\langle 2, m, rs_2\rangle \sim_{\beta}^{rts_0} \langle 2, m, rs'_2\rangle$ it suffices to show $rs_2 \sim_{\beta}^{rt_0} rs'_2$, since the call stack contains only one element and empty call stacks are by definition indistinguishable. From $h_1 \sim_{\beta}^{mds_0} h'_1$, $rs_1 \sim_{\beta}^{rt_0} rs'_1$, $ft(l) = low$ and $l \notin mds(asm\text{-}noread)$ it follows that $h_1(rs_1(v_1)(l) \sim_{\beta} h'_1(rs'_1(v_1))(l)$. From $rs_1 \sim_{\beta}^{rt_0} rs'_1$ and $h_1(rs_1(v_1)(l) \sim_{\beta} h'_1(rs'_1(v_1))(l)$ and the definition of rs_2 and rs'_2 we obtain $rs_2 \sim_{\beta}^{rt_0} rs'_2$. Thus condition (1) is satisfied.

Condition (2) is satisfied by definition of R.

Now we have to show condition (3). From the rules Rins and Riput of the operational semantics, we obtain

$$(\langle 2, m, rs_2\rangle, mon_0, mds_0, h_1) \overset{\varepsilon}{\twoheadrightarrow} (\langle 3, m, rs_2\rangle, mon_0, mds_0, h_2)$$

where $h_2 = h_1(rs_2(v_1))[h \mapsto rs_2(v_2)]$ if $v_1 \in dom(rs_2)$ and $rs_2(v_1) \in dom(h_1)$. Otherwise no execution step is possible and condition (3) is satisfied trivially.

If an execution step is possible, an execution step in the related configuration must be possible too. From the rules Rins and Riput of the operational semantics we obtain

$$(\langle 2, m, rs'_2\rangle, mon_0, mds_0, h'_1) \overset{\varepsilon}{\twoheadrightarrow} (\langle 3, m, rs'_2\rangle, mon_0, mds_0, h'_2)$$

where $h'_2 = h'_1(rs'_2(v_1))[h \mapsto rs'_2(v_2)]$. The rule Riput is applicable because from $rs_2 \sim_\beta^{rt_0} rs'_2$, $v_1 \in dom(rs_2)$, $rs_2(v_1) \in dom(h_1)$ and the safety of β, we have $rs'_2(v_1) \in dom(h_2)$. By definition of R and $rts_0 = update(rts_0, \varepsilon)$

$$((\langle 3, m, rs_2\rangle, mon_0, mds_0, h_2)R^{rt_0}(\langle 2, m, rs'_2\rangle, mon_0, mds_0, h'_2))$$

holds. Thus Condition (3) is satisfied.

Case $((\langle 3, m, rs_2\rangle, mon_0, mds_0, h_2)R_\beta^{rts_0}(\langle 3, m, rs'_2\rangle, mon_0, mds_0, h'_2))$**:** To show condition (1) we can apply same argument as in the previous case.

For condition (2) we have to show $h_2 \sim_\beta^{mds_0} h'_2$. From the definition of h_2 and h_2, from $ft(h) = high$ and from $h_1 \sim_\beta^{mds_0} h'_1$ we have $h_2 \sim_\beta^{mds_0} h'_2$.

Now we have to show condition (3). From the rules Rins and Rreturn-void of the operational semantics, we obtain

$$(\langle 3, m, rs_2\rangle, mon_0, mds_0, h_2) \overset{\varepsilon}{\twoheadrightarrow} (\langle\rangle, mon_0, mds_0, h_2)$$

From the rules Rins and Riput of the operational semantics we obtain the execution step in the related configuration

$$(\langle 2, m, rs'_2\rangle, mon_0, mds_0, h'_2) \overset{\varepsilon}{\twoheadrightarrow} (\langle\rangle, mon_0, mds_0, h'_2)$$

By definition of R and $rts_0 = update(rts_0, \varepsilon)$

$$((\langle\rangle, mon_0, mds_0, h_2)R_\beta^{rts_0}(\langle\rangle, mon_0, mds_0, h'_2))$$

holds. Thus Condition (3) is satisfied.

Case $((\langle\rangle, mon_0, mds_0, h_1)R_\beta^{\langle\rangle}(\langle\rangle, mon_0, mds_0, h'_1))$**:** Condition (1) is trivially satisfied, because empty call stacks are by definition indistinguishable.

For condition (2) applies the same argument as in the previous case.

Condition (3) is trivially fulfilled, because there is no rule of the operational semantics such that $(\langle\rangle, mon, mds, h) \overset{\alpha}{\twoheadrightarrow} (cs, mon, mds, h)$.

The cases in which the environment created new objects and the symmetric cases are analogous. □

Now we show that the security property classifies a program with a direct leak as insecure.

Lemma 2. *The program $P = (C, F, M)$ starting in method m with $C = \{c\}$, $F = \{l, h\}$, $\mathcal{M}_{\mathrm{id}} = \{m\}$ and*

$$M = \left\{ m, i \mapsto \begin{cases} \texttt{iget}\ v_2\ v_1\ h & i = 1 \\ \texttt{iput}\ v_2\ v_1\ l & i = 2 \\ \texttt{returnvoid} & i = 3 \end{cases} \right\}$$

is not secure with respect to domain assignment $ft(l) = low$ and $ft(h) = high$.

Proof. We show that $(P, m) \sim^{mds_0}_{low} (P, m)$ does not hold by contradiction.

Let $h_1, h'_1 \in \mathcal{H}$ such that $h_1(l_0) = (c, \{(h, 0), (l, 0)\})$ and $h'_1(l_0) = (c, \{(h, 1), (l, 0)\})$. Then $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ with $l_0 \mapsto l_0$ is safe w.r.t h_1 and h'_1 since $dom(h_1) = dom(\beta)$ and $dom(h_1) = codom(\beta)$. Further let $rs_1, rs'_1 \in (\mathcal{R} \rightharpoonup \mathcal{V})$ such that $rs_1 \sim^{rt_0}_{\beta} rs'_1$ and $rs_1(v_1) = rs'_1(v_1) = l_0$.

Assume that R is a strong low bisimulation modulo mds_0, such that

$$(\langle 1, m, rs_1 \rangle, mon_0, mds_0, h_1) R^{rt_0}_{\beta} (\langle 1, m, rs'_1 \rangle, mon_0, mds_0, h'_1)$$

Since no instruction sends the event $cs, r_1 \ldots r_n$ condition (3b) is always true.

Conditions (1) and (2) are satisfied by definition and from the semantics (rules Rins and Riget) we get

$$(\langle 1, m, rs_1 \rangle, mon_0, mds_0, h_1) \overset{v_1 h \triangleright v_2}{\twoheadrightarrow} (\langle 2, m, rs_2 \rangle, mon_0, mds_0, h_1)$$
$$(\langle 1, m, rs'_1 \rangle, mon_0, mds_0, h'_1) \overset{v_1 h \triangleright v_2}{\twoheadrightarrow} (\langle 2, m, rs'_2 \rangle, mon_0, mds_0, h'_1)$$

with $rs_2 = rs_1[v_2 \mapsto 0]$ and $rs'_2 = rs'_1[v_2 \mapsto 1]$. As the heaps do not change we have $\beta' = \beta$.

Now we have to show that

$$(\langle 2, m, rs_2 \rangle, mon_0, mds_0, h_1) R^{rts}_{\beta} (\langle 2, m, rs'_2 \rangle, mon_0, mds_0, h'_1)$$

for $rts = update(v_1 h \triangleright v_2, rts_0) = rt_0[v_2 \mapsto high] :: \langle\rangle$ holds. Since the heaps did not changed condition (1) of strong low bisimulation modulo modes is satisfied. From $rt(v_2) = high$ and the definition of register set indistinguishability we get $rs_2 \sim^{rt}_{\beta} rs'_2$ and from call stack indistinguishability we have $\langle 2, m, rs_2 \rangle \sim^{rts}_{\beta} \langle 2, m, rs'_2 \rangle$. Thus condition (2) is satisfied.

From the semantics (rules Rins and Riput) we get

$$(\langle 2, m, rs_2 \rangle, mon_0, mds_0, h_2) \twoheadrightarrow (\langle 3, m, rs_2 \rangle, mon_0, mds_0, h_3)$$
$$(\langle 2, m, rs'_2 \rangle, mon_0, mds_0, h'_2) \twoheadrightarrow (\langle 3, m, rs'_2 \rangle, mon_0, mds_0, h'_3)$$

with and $h_3(l_0) = (c, \{(h, 0), (l, 0)\})$ and $h'_3(l_0) = (c, \{(h, 1), (l, 1)\})$. Since no objects were created $\beta' = \beta$.

Now we have to show that

$$(\langle 3, m, rs_2 \rangle, mon_0, mds_0, h_3) R^{rt}_{\beta} (\langle 3, m, rs'_2 \rangle, mon_0, mds_0, h'_3)$$

From $ft(l) = low$ and the definition of value indistinguishability we get that for $l_0 \in dom(\beta)$ that $h_3(l_0)(l) = 0 \not\sim_{\beta} 1 = h'_3(l_0)(l)$ and therefore $h_3 \not\sim^{mds_0}_{\beta} h'_3$. Hence condition (1) of strong low bisimilarity modulo modes is not satisfied. This contradicts our assumption that R is a strong low bisimulation modulo modes. Therefore the program P (starting in method m) is not secure. $\square$

4.2 Static Thread Structure

In this section we introduce a multithreaded subset of the DVM with a static thread structure.

We extend the set of instructions by the instructions `enter` r and `exit` r that model the acquiring and releasing of monitors for an location saved in register r. The following rules define the semantics of these new instructions. Note that the monitor state is not set in these rules, but rather set by the global transition relation to ensure that no incompatible monitor states can occur.

$$\frac{M(m)(i) = \texttt{enter}\ r \qquad r \in \mathit{dom}(rs) \qquad rs(r) \in \mathit{dom}(h)}{((i, m, rs) :: cs, mon, h)_M \overset{\blacklozenge rs(r)}{\rightarrowtriangle} ((i + 1, m, rs) :: cs, mon, h)_M}$$

$$\frac{M(m)(i) = \texttt{exit}\ r \qquad r \in \mathit{dom}(rs) \qquad rs(r) \in \mathit{dom}(h)}{((i, m, rs) :: cs, mon, h)_M \overset{\lozenge rs(r)}{\rightarrowtriangle} ((i + 1, m, rs) :: cs, mon, h)_M}$$

If a location is saved in register r theses rules, send the corresponding enter/leave event to the global transition relation. Internal timing leaks (compare for the example in Section 3.5) are avoided by requiring equal mode states.

Now we define the transition system for the DVM subset with a static thread structure.

Definition 4.4. $\mathrm{DVM}_{\mathrm{stat}}(\mathit{conf}_0, \mathrm{P}) = \mathit{ATS}(\mathit{conf}_0, \mathrm{P}, \mathbb{Z}, \mathcal{I}_{\mathrm{stat}}, \rightarrowtriangle)$ *is a subset of the DVM with a static thread structure for some program* P *and some initial configuration* conf_0*, where*

$$\mathcal{I}_{\mathrm{stat}} = \mathcal{I}_{\mathrm{seq}} \cup \{//a//\texttt{enter}\ r, //a//\texttt{exit}\ \texttt{enter}\ r, \texttt{exit}\ r : r \in \mathcal{R}, a \in \mathit{Ann}\}$$

The transition system contains the extended set of instructions and has no restrictions on the initial state to support multiple threads at the program start.

If the following program is regarded as a multithreaded program with two threads, one starting the execution in method m_1 the other in m_2, then this program has an internal timing leak when executed under round-robin scheduler that executes m_1 first and with following field domain assignment $ft(h) = \mathit{high}$ and $ft(l) = \mathit{low}$.

Lemma 3. *The program $P = (C, F, M)$ starting in methods m_1 and m_2 with*

$$C = \{c\}$$
$$F = \{h, l\}$$
$$\mathcal{M}_{\mathrm{id}} = \{m_1, m_2\}$$

$$M = \left\{ \left(m_1, i \mapsto \begin{cases} \texttt{const}\ v_1\ 1 & i = 1 \\ \texttt{iget}\ v_2\ v_0\ h & i = 2 \\ \texttt{if}\ v_1\ v_2\ 5 & i = 3 \\ \texttt{nop} & i = 4 \\ \texttt{put}\ v_1\ v_0\ l & i = 5 \\ \texttt{returnvoid} & i = 6 \end{cases} \right), \left(m_2, i \mapsto \begin{cases} \texttt{nop} & i = 1 \\ \texttt{nop} & i = 2 \\ \texttt{const}\ v_1\ 0 & i = 3 \\ \texttt{put}\ v_1\ v_0\ l & i = 4 \\ \texttt{returnvoid} & i = 5 \end{cases} \right) \right\}$$

is insecure for the domain assignment $ft(h) = \mathit{high}$ and $ft(l) = \mathit{low}$.

Intuitively the branching in m_1 delays the assignment of 1 to the field l for one execution steps. In the case it is not delayed it is executed second last instruction, the last instruction is the assignment of 0 to the field l in method m_2, which overwrites the old value. If the assignment is delayed the situation is the other way around.

To make this argument precise, we assume that the thread executing m_1 is secure and lead this to a contradiction.

Proof. We show that $(P, m_1) \sim_{low}^{mds_0} (P, m_1)$ does not hold by contradiction.

Let $h_1, h_1' \in \mathcal{H}$ such that $h_1(l_0) = (c, \{(h, 0), (l, 0)\})$ and $h_1'(l_0) = (c, \{(h, 1), (l, 0)\})$. Then $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ with $l_0 \mapsto l_0$ is safe w.r.t h_1 and h_1' since $dom(h_1) = dom(\beta)$ and $dom(h_1) = codom(\beta)$. Further let $rs_1, rs_1' \in (\mathcal{R} \rightharpoonup \mathcal{V})$ such that $rs_1 \sim_\beta^{rt_0} rs_1'$ and $rs_1(v_0) = rs_1'(v_0) = l_0$.

Assume that R is a strong low bisimulation modulo modes, such that

$$(\langle 1, m_1, rs_1\rangle, mon_0, mds_0, h_1) R_\beta^{rts_0} (\langle 1, m_1, rs_1'\rangle, mon_0, mds_0, h_1')$$

Since no instruction sends the event $cs, r_1 \ldots r_n$ condition (3b) is always true.

Conditions (1) and (2) are satisfied by definition and from the semantics (rules Rins and Rconst) we get

$$(\langle 1, m_1, rs_1\rangle, mon_0, mds_0, h_1) \xrightarrow{\triangleright v_1} (\langle 2, m_1, rs_2\rangle, mon_0, mds_0, h_1)$$
$$(\langle 1, m_1, rs_1'\rangle, mon_0, mds_0, h_1') \xrightarrow{\triangleright v_1} (\langle 2, m_1, rs_2'\rangle, mon_0, mds_0, h_1')$$

with $rs_2 = rs_1[v_1 \mapsto 1]$ and $rs_2' = rs_1'[v_1 \mapsto 1]$. As the heaps do not change we have $\beta' = \beta$.

Now we have to show that

$$(\langle 2, m_1, rs_2\rangle, mon_0, mds_0, h_1) R_\beta^{rts_0} (\langle 2, m_1, rs_2'\rangle, mon_0, mds_0, h_1')$$

since the security domains of the registers stayed the same. As the heaps have not changed condition (1) is satisfied and as we wrote a constant into register v_1 we have $r_2 \sim_\beta^{rt_0} r_2'$.

From the semantics (rules Rins and Riget) we get

$$(\langle 2, m_1, rs_2\rangle, mon_0, mds_0, h_1) \xrightarrow{v_0 h \triangleright v_2} (\langle 3, m_1, rs_3\rangle, mon_0, mds_0, h_1)$$
$$(\langle 2, m_1, rs_2'\rangle, mon_0, mds_0, h_1') \xrightarrow{v_0 h \triangleright v_2} (\langle 3, m_1, rs_3'\rangle, mon_0, mds_0, h_1')$$

with $rs_3 = rs_2[v_2 \mapsto 0]$ and $rs_3' = rs_2'[v_2 \mapsto 1]$. As the heaps do not change we have $\beta' = \beta$.

Now we have to show that

$$(\langle 3, m, rs_3\rangle, mon_0, mds_0, h_2) R_\beta^{rts} (\langle 3, m, rs_3'\rangle, mon_0, mds_0, h_2')$$

for $rts = update(v_0 h \triangleright v_2, rts_0) = rt :: \langle\rangle = rt_0[v_2 \mapsto high] :: \langle\rangle$ holds. Since the heaps have not changed condition (1) is satisfied. From $rt(v_2) = high$ we get $rs_3 \sim_\beta^{rt} rs_3'$.

We can apply the rules Rins and Rifeq-false because $1 = rs_3(v_1) \neq rs_3(v_2) = 0$, which leads to

$$(\langle 3, m_1, rs_3\rangle, mon_0, mds_0, h_2) \xrightarrow{\varepsilon} (\langle 4, m_1, rs_3\rangle, mon_0, mds_0, h_2)$$

and we can apply the rules RINS and RIFEQ-TRUE) because $1 = r'_3(v_1) = r'_3(v_2) = 1$, which leads to

$$(\langle 3, m_1, rs'_3\rangle, mon_0, mds_0, h'_2) \overset{\varepsilon}{\rightharpoonup} (\langle 5, m_1, rs'_3\rangle, mon_0, mds_0, h'_2)$$

Now we have to show that

$$(\langle 4, m, rs_3\rangle, mon_0, mds_0, h_2) R^{rts}_{\beta} (\langle 5, m, rs'_3\rangle, mon_0, mds_0, h'_2)$$

holds. Because the register sets and heaps have not changed conditions (1) and (2) are satisfied satisfied. To check condition (3a), we apply the rules RINS and RIPUT, which leads to

$$(\langle 5, m_1, rs_3\rangle, mon_0, mds_0, h_3) \overset{\varepsilon}{\rightharpoonup} (\langle 6, m_1, rs_3\rangle, mon_0, mds_0, h_4)$$

As we can apply the RINS and RIPUT we have the following

$$(\langle 4, m_1, rs'_3\rangle, mon_0, mds_0, h'_3) \overset{\varepsilon}{\rightharpoonup} (\langle 5, m_1, rs'_3\rangle, mon_0, mds_0, h'_3)$$

with $h_4(l_0) = (c, \{(h, 1), (l, 1)\})$.

Now we have to show that

$$(\langle 6, m, rs_3\rangle, mon_0, mds_0, h_4) R^{rts}_{\beta} (\langle 5, m, rs'_3\rangle, mon_0, mds_0, h'_3)$$

But from $ft(l) = low$ and the definition of value indistinguishability we get that for $l_0 \in dom(\beta)$ that $h_4(l_0)(l) = 1 \not\sim_\beta 0 = h'_3(l_0)(l)$. Hence condition (2) of strong low bismiliarity is not fulfilled. This contradicts our assumption that R is a strong low bisimulation modulo modes. Hence the program is not secure. □

4.3 Dynamic Thread Creation

In this section we extend the instance of the previous section with an instruction for dynamic thread creation.

$$\frac{\begin{array}{c} M(m)(i) = \texttt{start}\ r_1 \dots r_n\ m \\ rs' = \{(v_i, rs(r_{i+1})) : 0 \leq i < n\} \qquad t = (\langle 1, m, rs'\rangle, mon_0, mds_0) \end{array}}{((i, m, rs) :: cs, mon, h)_M \overset{t;r_1\dots r_n}{\rightharpoonup} ((i+1, m, rs) :: cs, mon, h)_M}$$

The instruction `start` creates a new thread that starts the execution in method m and has a register set that is initialized with the registers $r_1, \dots, r_n$. The is reflected in the security property by the event $t; r_1 \dots r_n$ that ensures that the register sets of the new threads are still indistinguishable.

Definition 4.5. $\mathrm{DVM}_{\mathrm{dyn}}(conf_0, \mathrm{P}) = ATS(conf_0, \mathrm{P}, \mathbb{Z}, \mathcal{I}_{\mathrm{dyn}})$ *is a subset of the DVM with support for dynamic thread creation for some program* P *and some initial configuration* $conf_0$ *where*

$$\mathcal{I}_{\mathrm{dyn}} = \mathcal{I}_{\mathrm{stat}} \cup \{//a//\texttt{start}\ r_1 \dots r_n\ m, \texttt{start}\ r_1 \dots r_n\ m : r_1, \dots, r_n \in \mathcal{R}, m \in \mathcal{M}, a \in Ann\}$$

The following program has an indirect leak from field h to l, if it is started in method m_1 independent of the scheduler.

Lemma 4. *The program $P = (C, M, F)$ starting in method m_1 where*

$$C = \{c\}$$
$$F = \{h, l\}$$
$$\mathcal{M}_{\text{id}} = \{m_1, m_2, m_3\}$$

$$M = \left\{ \left(m_1, i \mapsto \begin{cases} \texttt{const}\ v_1\ 1 & i = 1 \\ \texttt{iget}\ v_2\ v_0\ h & i = 2 \\ \texttt{if}\ v_1\ v_2\ 6 & i = 3 \\ \texttt{create}\ v_0\ m_2 & i = 4 \\ \texttt{goto}\ 7 & i = 5 \\ \texttt{create}\ v_0\ m_3 & i = 6 \\ \texttt{returnvoid} & i = 7 \end{cases} \right), \right.$$
$$\left. \left(m_2, i \mapsto \begin{cases} \texttt{const}\ v_1\ 1 & i = 3 \\ \texttt{put}\ v_1\ v_0\ l & i = 4 \\ \texttt{returnvoid} & i = 5 \end{cases} \right), \left(m_3, i \mapsto \begin{cases} \texttt{const}\ v_1\ 0 & i = 3 \\ \texttt{put}\ v_1\ v_0\ l & i = 4 \\ \texttt{returnvoid} & i = 5 \end{cases} \right) \right\}$$

is not secure with respect to the domain assignment $ft(l) = low$ and $ft(h) = high$.

Method m_1 spawns a thread starting in m_2 if the field h contains 1, otherwise it spawns the thread m_3. Method m_2 writes 1 into the field l and method m_3 writes 0 into field l. Thus in principle the program has an classic implicit leak, with the indirection of thread creation.

To make this argument precise, we assume that the thread executing m_1 is secure and lead this to a contradiction.

Proof. We show that $(P, m_1) \sim_{low}^{mds_0} (P, m_1)$ does not hold by contradiction.

Let $h_1, h_1' \in \mathcal{H}$ such that $h_1(l_0) = (c, \{(h, 0), (l, 0)\})$ and $h_1'(l_0) = (c, \{(h, 1), (l, 0)\}$. Then $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ with $l_0 \mapsto l_0$ is safe w.r.t h_1 and h_1 since $dom(h_1) = dom(\beta)$ and $dom(h_1) = codom(\beta)$. Further let $rs_1, rs_1' \in (\mathcal{R} \rightharpoonup \mathcal{V})$ such that $rs_1 \sim_{\beta}^{rt_0} rs_1'$ and $rs_1(v_0) = rs_1'(v_0) = l_0$.

Assume that R is a strong low bisimulation modulo modes such that

$$(\langle 1, m_1, rs_1 \rangle, mon_0, mds_0, h_1) R_{\beta}^{rts_0} (\langle 1, m_1, rs_1' \rangle, mon_0, mds_0, h_1')$$

Conditions (1) and (2) are satisfied by definition and from the semantics (rules Rins and Rconst we get

$$(\langle 1, m_1, rs_1 \rangle, mon_0, mds_0, h_1) \overset{\triangleright v_1}{\rightharpoonup} (\langle 2, m_1, rs_2 \rangle, mon_0, mds_0, h_1)$$
$$(\langle 1, m_1, rs_1' \rangle, mon_0, mds_0, h_1') \overset{\triangleright v_1}{\rightharpoonup} (\langle 2, m_1, rs_2' \rangle, mon_0, mds_0, h_1')$$

with $rs_2 = rs_1[v_1 \mapsto 1]$ and $rs_2' = rs_1'[v_1 \mapsto 1]$. As the heaps do not change we have $\beta' = \beta$.

Now we have to show that

$$(\langle 2, m_1, rs_2 \rangle, mon_0, mds_0, h_1) R_{\beta}^{rts_0} (\langle 2, m_1, rs_2' \rangle, mon_0, mds_0, h_1')$$

since the security domains of the registers stayed the same. As the heaps did not changed we have $h_2 \sim_\beta^{mds_0} h_2'$ and as we wrote a constant into register v_1 we have $r_2 \sim_\beta^{rt_0} r_2'$.

From the semantics (rules Rins and Riget) we get

$$(\langle 2, m_1, rs_2\rangle, mon_0, mds_0, h_1) \xrightarrow{v_0 h \triangleright v_2} (\langle 3, m_1, rs_3\rangle, mon_0, mds_0, h_2)$$
$$(\langle 2, m_1, rs_2'\rangle, mon_0, mds_0, h_1') \xrightarrow{v_0 h \triangleright v_2} (\langle 3, m_1, rs_3'\rangle, mon_0, mds_0, h_2')$$

with $rs_3 = rs_2[v_2 \mapsto 0]$ and $rs_3' = rs_2'[v_2 \mapsto 1]$. As the heaps do not change we have $\beta' = \beta$.

Now we have to show that

$$(\langle 3, m, rs_3\rangle, mon_0, mds_0, h_2) R_\beta^{rts} (\langle 3, m, rs_3'\rangle, mon_0, mds_0, h_2')$$

for $rts = update(v_0 h \triangleright v_2, rts_0) = rt :: \langle\rangle = rt_0[v_2 \mapsto high] :: \langle\rangle$ holds. As the heaps have not changed condition (1) is satisfied. From $rt(v_2) = high$ we get $rs_3 \sim_\beta^{rt} rs_3'$.

We can apply the rules Rins and Rifeq-false because $1 = rs_3(v_1) \neq rs_3(v_2) = 0$, which leads to

$$(\langle 3, m_1, rs_3\rangle, mon_0, mds_0, h_2) \xrightarrow{\varepsilon} (\langle 4, m_1, rs_3\rangle, mon_0, mds_0, h_2)$$

and we can apply the rules Rins and Rifeq-true) because $1 = r_3'(v_1) = r_3'(v_2) = 1$, which leads to

$$(\langle 3, m_1, rs_3'\rangle, mon_0, mds_0, h_2') \xrightarrow{\varepsilon} (\langle 5, m_1, rs_3'\rangle, mon_0, mds_0, h_2')$$

Now we have to show that

$$(\langle 4, m, rs_3\rangle, mon_0, mds_0, h_2) R_\beta^{rts} (\langle 5, m, rs_3'\rangle, mon_0, mds_0, h_2')$$

holds. Because the register sets and heaps have not changed conditions (1) and (2) are satisfied. To check condition (3), we apply the rules Rins and Rcreate, which leads to

$$(\langle 5, m_1, rs_3\rangle, mon_0, mds_0, h_3) \xrightarrow{\langle 1, m_2, rs_4\rangle; v_0} (\langle 6, m_1, rs_3\rangle, mon_0, mds_0, h_4)$$
$$(\langle 4, m_1, rs_3'\rangle, mon_0, mds_0, h_3') \xrightarrow{\langle 1, m_3, rs_4'\rangle; v_0} (\langle 5, m_1, rs_3'\rangle, mon_0, mds_0, h_3')$$

where $rs_4 = rs_3 \restriction \{v_0\}$ and $rs_4' = rs_3' \restriction \{v_0\}$. According to condition (3b) of strong low bisimulation modulo modes, we have to show that

$$(\langle 1, m_2, rs_4\rangle, mon_0, mds_0, h_3) R_\beta^{rts_0} (\langle 1, m_3, rs_4'\rangle, mon_0, mds_0, h_3')$$

and that no assumptions were made on creation. The second requirement is trivially satisfied, since we have no assumptions at all. To show the first requirement we proceed as before.

As the heaps have not changed and the register sets are reduced to the same registers conditions (1) and (2) hold. From Rins and Rconst we get

$$(\langle 1, m_2, rs_4\rangle, mon_0, mds_0, h_3) \xrightarrow{\triangleright v_1} (\langle 2, m_2, rs_5\rangle, mon_0, mds_0, h_3)$$
$$(\langle 1, m_3, rs_4'\rangle, mon_0, mds_0, h_3') \xrightarrow{\triangleright v_1} (\langle 2, m_3, rs_5'\rangle, mon_0, mds_0, h_3')$$

with $rs_5 = rs_4[v_1 \mapsto 1]$, $rs'_5 = rs'_4[v_1 \mapsto 1]$. As the heaps do not change we have $\beta' = \beta$.

Now we have to show that

$$(\langle 2, m_1, rs_2\rangle, mon_0, mds_0, h_1) R_\beta^{rts_0} (\langle 2, m_1, rs'_2\rangle, mon_0, mds_0, h'_1)$$

since the security domains of the registers stayed the same. As the heaps did not changed we have $h_2 \sim_\beta^{mds_0} h'_2$ and as we wrote a constant into register v_1 we have $r_2 \sim_\beta^{rt_0} r'_2$. To check condition (3), we apply the rules RINS and RIPUT, which leads to

$$(\langle 2, m_2, rs_5\rangle, mon_0, mds_0, h_3) \overset{\varepsilon}{\twoheadrightarrow} (\langle 3, m_2, rs_5\rangle, mon_0, mds_0, h_4)$$
$$(\langle 2, m_3, rs'_5\rangle, mon_0, mds_0, h'_3) \overset{\varepsilon}{\twoheadrightarrow} (\langle 3, m_3, rs'_5\rangle, mon_0, mds_0, h'_4)$$

where $h_4(l_0) = (c, \{(h, 0), (l, 1)\})$ and $h_4(l_0) = (c, \{(h, 1), (l, 0)\})$.

Now we have to show that

$$(\langle 3, m, rs_5\rangle, mon_0, mds_0, h_4) R_\beta^{rts} (\langle 3, m, rs'_5\rangle, mon_0, mds_0, h'_4)$$

. But from $ft(l)low$ and the definition of value indistinguishability we get that for $l_0 \in dom(\beta)$ that $h_4(l_0)(l) = 1 \not\sim_\beta 0 = h'_3(l_0)(l)$. Hence condition (2) of strong low bismiliarity is not fulfilled. This contradicts our assumption that R is a strong low bisimulation modulo modes. Hence the program is not secure when starting in method m_1. □

Chapter 5

Summary

5.1 Conclusion

This work started with presenting information about the execution model of the DVM and by setting the assumptions and guarantees from [MSS11] into relation to this execution model.

After that we modeled a transition system that is parametric in the transition relation for the semantics of the instructions. This transition system provides the facilities for multi-threading (using a possibilistic scheduler), synchronization via monitors and dynamic thread creation. While it is designed to be instantiated with subsets of the DVM, it can be instantiated with all byte-code languages that have similar requirements on multi-threading. Therefore this transition system may be of independent interest for other analysis of multi-threaded byte-code languages.

Based on this transition system we developed a security policy that assigns security domains to every field. We analyzed the information flow in the transition system and captured the results in a security property based on [MSS11]. One major problem in this process was the treatment of registers. The chosen solution is not fully satisfactory, because besides being precise it introduces to much complexity in the security property. In addition to presenting the security property we also provide a reference property for multi-threaded programs and provide the basis for a proof of the parallel compositionality of the developed security property.

We finished the work by instantiating the transition system with subsets of the DVM and illustrating the security property on small example programs. The analysis of these programs showed, that it is not feasible – even compared to other security properties – to analyze other programs as toy examples by hand, as the relations become really huge.

5.2 Related Work

Language-based information flow control for byte-code languages is an active research area (cf. [GS05], [HP06], [BPR07], [BRRS10]). The authors of [BPR07] introduced a security property and a sound information flow type system for a large fragment of the JVM. This fragment includes object orientation, methods, exception-handling and arrays, but has no support for multiple threads. Based on this work [Man11] adapted (parts of) the type system for the DVM with was further developed and proofed sound in [Web12]. In contrast to this work they used a standard non-interference property instead of a bisimulation.

With the success of the Android operating system there are attempts (e.g. [JMF12] and [WK12]) in formalizing the semantics of the DVM. One of those is SymDroid. SymDroid [JMF12] is a symbolic executor for Dalvik programs, that is implemented in OCaml. It operates on 16 abstract instructions. Each abstract instruction represents a group of actual DVM instructions. For example all binary operations on numbers are grouped to one abstract instruction. The semantics of the abstract instructions is formalized as a small-step semantics. However SymDroid lacks support multi-threading.

[WK12] is an other approach to static analysis of the DVM. They have developed a traditional control flow analysis using abstract domains for the DVM. This analysis supports even dynamic features like reflection and Javascript interfaces. A prototype implementation of the analysis generates Prolog clauses these can then be queried for any information that the analysis specifies.

5.3 Future Work

Assumptions and guarantees are in this thesis are not object sensitive. That means a assumption or guarantee on a field needs to hold for all objects to which the field belongs to. That includes objects that are created after acquiring and before releasing a assumption or guarantee. This is in most cases too restrictive, especially when working with generic classes like e.g. collections. Those assumption and guarantees would be set for all instances of a collection, which may be not what you want, if you use collections for several purposes. Therefore refining the security property to object sensitive assumptions and guarantees would increase precision and make the property more adequate for real world programs.

The current security policy is object insensitive, too. That leads to similar restrictions as the insensitivity of the assumption and guarantees. E.g. it is not possible to have one collection that has a low security domain and an other with a high security domain. Therefore it would make sense to refine the security policy, too.

Analyzing programs by hand using the security property is lengthy, error prone and time consuming, therefore extending the security property with a sound type system would pave the way for an automated analysis. Before developing such a type system, proofing the compositionality of the security property would increase the confidence in the security property.

Appendix A

Definition Graphs

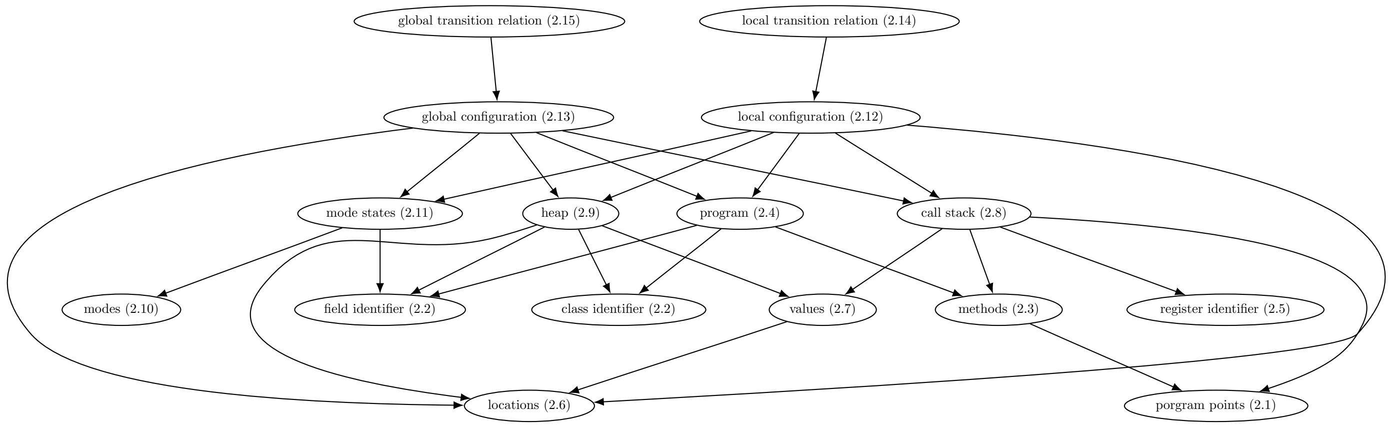
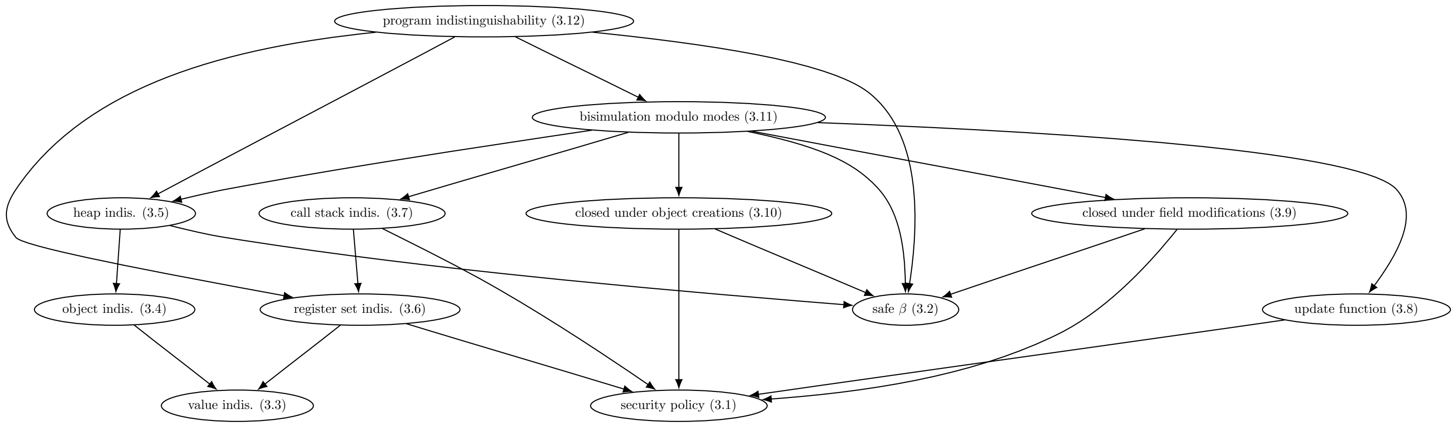
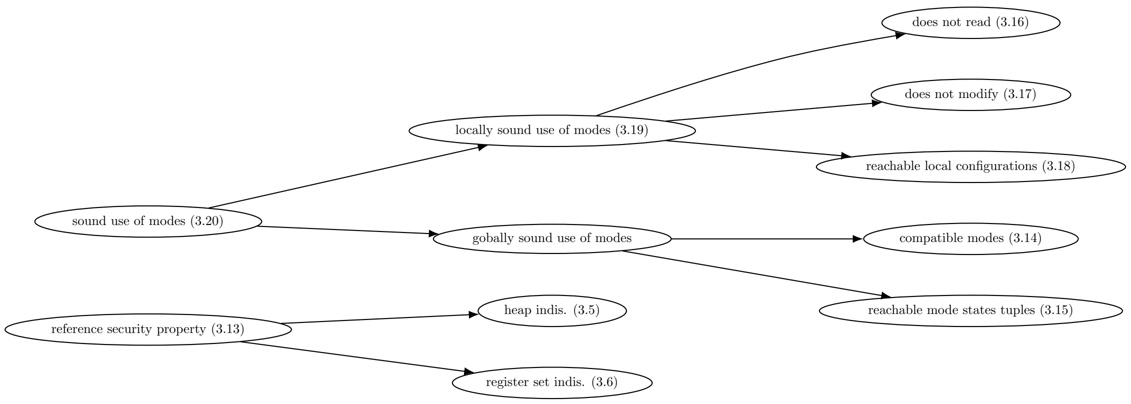


Figure A.1: A depencency graph of the definitions of Chapter 2. The number in parentheses is the number of the definition. $a \rightarrow b$ denotes that definition a uses/referes to definition b.



program indistinguishability (3.12)
bisimulation modulo modes (3.11)
heap indis. (3.5)
call stack indis. (3.7)
closed under object creations (3.10)
closed under field modifications (3.9)
object indis. (3.4)
register set indis. (3.6)
safe β (3.2)
update function (3.8)
value indis. (3.3)
security policy (3.1)

Figure A.2: A depencency graph of the definitions of Chapter 3. The number in parentheses is the number of the definition. $a \to b$ denotes that definition a uses or referes to definition b. The graph does not include definitions from Chapter 2.



does not read (3.16)
does not modify (3.17)
locally sound use of modes (3.19)
reachable local configurations (3.18)
sound use of modes (3.20)
gobally sound use of modes
compatible modes (3.14)
heap indis. (3.5)
reachable mode states tuples (3.15)
reference security property (3.13)
register set indis. (3.6)

Figure A.3: A depencency graph of the definitions of Chapter 3. The number in parentheses is the number of the definition. $a \to b$ denotes that definition a uses or referes to definition b. The graph does not include definitions from Chapter 2.

Bibliography

[BPR07] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 125–140, Berlin, Heidelberg, 2007. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1762174.1762189`.

[BRRS10] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3), 2010.

[DDSW10] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In Mike Burmester, Gene Tsudik, Spyros S. Magliveras, and Ivana Ilic, editors, *ISC*, volume 6531 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2010.

[Dev] Mailinglist Android Linux Kernel Development. *Are the DVM registers shared between threads?* Last accessed on 06.03.2013. URL: `https://groups.google.com/forum/?fromgroups=#!topic/android-kernel/Y6jGgEaJGx4`.

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

[GS05] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 346–362, Berlin, Heidelberg, 2005. Springer-Verlag.

[HP06] R. R. Hansen and C. W. Probst. Non-interference and erasure policies for java card bytecode. In *6th International Workshop on Issues in the Theory of Security (WITS '06)*, 2006. URL: `http://www2.imm.dtu.dk/pubdb/p.php?4742`.

[JMF12] Jinseong Jeon, Kristopher K. Micinski, and Jeffery S. Forster. SymDroid: Symbolic Execution for Dalvik Bytecode. In *Draft*, 2012.

[Man11] Christopher Mann. A Static Framework for Privacy Analysis of Android Applications (Bachelor Thesis), 2011.

[MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and Guarantees for Compositional Noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF)*, pages 218–232, Cernay-la-Ville, France, 2011. IEEE Computer Society.

[Pro] The Android Open Source Project. *Starting an Activity.* Last accessed on 06.03.2013. URL: `http://developer.android.com/training/basics/activity-lifecycle/starting.html`.

[Pro07] The Android Open Source Project. *Bytecode for the Dalvik VM*, 2007. Last accessed on April 8, 2013. URL: `http://source.android.com/tech/dalvik/dalvik-bytecode.html`.

[PvdM12] Christy Pettey and Rob van der Meulen. Gartner Says Worldwide Sales of Mobile Phones Declined 3 Percent in Third Quarter of 2012; Smartphone Sales Increased 47 Percent, November 2012. Last accessed on 15.03.2013. URL: `http://www.gartner.com/it/page.jsp?id=2237315`.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic Noninterference for Multi-Threaded Programs. In *CSFW*, pages 200–214, 2000.

[Web12] Alexandra Weber. A Sound Theory of Analysis for a Certifying App Store (Bachelor Thesis), 2012.

[WK12] Erik Ramsgaard Wognsen and Henrik Søndberg Karlsen. Static Analysis of Dalvik Bytecode and Reflection in Android, 2012.

[ZM03] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *In Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, 2003.