Karlsruhe Institute of Technology

Efficient Logic-Based Information Flow Analysis Of Object-Oriented Programs

Bachelorarbeit von

Fabian Ruch

An der Fakultät für Informatik
Institut für Theoretische Informatik
Arbeitsgruppe Anwendungsorientierte Formale Verifikation

Erstgutachter:	Prof. Dr. Bernhard Beckert
Betreuender Mitarbeiter:	Daniel Bruns

Bearbeitungszeit: 1. Mai 2013 – 1. Oktober 2013

Ich möchte Daniel Bruns herzlich für den Themenvorschlag und die Einführung in das Thema dieser Arbeit sowie für seine Zeit zu regelmäßigen Treffen und Korrekturen danken.

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 1. Oktober 2013

.....

(**Fabian Ruch**)

Zusammenfassung

In dieser Arbeit wird eine dynamische Logik für die objektorientierte Programmiersprache Java Card um eine Modalität zur effizienten Informationsflussanalyse erweitert. Die Informationsflussanalyse zieht Objekte, Kontrollfluss und Aufrufkontext in Betracht, kann aber nur Aussagen für stets terminierende Programme treffen.

Die dynamische Logik JavaDL ist eine Obermenge von Prädikatenlogik, deren Formeln auch Modalitäten für Java-Programme enthalten können. Die Semantik für die Modalitäten schreibt vor, dass eine dahinter geschriebene JavaDL-Formel in jedem möglichen Endzustand des Programms gelten muss. Ein Programmzustand umfasst auch den Heap und wird durch sogenannte Updates modelliert.

Eine notwendige Bedingung für die Sicherheit von Computersystemen ist, dass einem Angreifer die Inhalte bestimmter Speicherstellen niemals zugänglich sind. Informationsflussanalyse soll verifizieren, ob ein Programm solche Regelungen einhält. Eine Formalisierung solcher Regelungen ist das Noninterference-Sicherheitsmodell, welches für einen Angreifer beobachtbare Auswirkungen von Geheimnissen auf öffentliche Ausgaben, die Rückschlüsse auf Geheimnisse zulassen, ausschließt.

Das Noninterference-Sicherheitsmodell kann in JavaDL formuliert werden und der JavaDL-Kalkül kann die Modellinstanz für eine gültige Programmspezifikation auch beweisen. Die direkte Übersetzung von Noninterference in eine JavaDL-Formel hat aber zufolge, dass das spezifizierte Programm während eines Beweises durch den Kalkül zweimal ausgeführt wird. Dieser teilweise unnötige Zusatzaufwand soll durch die in dieser Arbeit definierte Modalität und die dafür verfügbaren Kalkülregeln verringert werden.

Die Regeln des erweiterten Sequenzenkalküls für JavaDL und die neue Modalität nutzen aus, dass in manchen Fällen die beiden Ausführungen des spezifizierten Programms synchron ausgewertet werden können.

Abstract

In this thesis, the JavaDL logic is amended by a new modality for the verification of termination-insensitive, object-sensitive, flow-sensitive noninterference of memory locations in object-oriented programs, that is local variables and heap object fields. Noninterference is a security model applied in information flow analysis and the amended JavaDL logic improves the information flow analysis of Java Card programs using the JavaDL sequent calculus.

JavaDL formulae are a superset of first-order logic formulae and the logic additionally contains dynamic logic modalities as well as a concept called updates for the transition and evaluation of Java Card program states. Noninterference of memory locations is a property of programs and the respective pairs of start and end states of their executions. The satisfaction of the property implies that a particular set of memory locations does not interfere with the memory locations not included in this set. A set of memory locations does not interfere with another set if the evaluations of the included memory locations in the end state of any execution are independent of the evaluations of the memory locations included in the other set.

Noninterference in Java Card programs has already been formulated and verified in JavaDL by a concept called self-composition which compares the end states of two independent executions of the same program with respect to a particular set of memory locations. This investigation is realisable with the means provided by JavaDL, in particular the dynamic logic modalities. However, in many cases the double execution poses unnecessary proof overhead and the new modality defined in this thesis aims at reducing this overhead. The new calculus reaches this goal by partially interpreting the verified program on a single execution path. As soon as this execution path branches the calculus must return to self-composition or the analysis loses precision.

Contents

1. Introduction

One attribute of computer security is confidentiality, which excludes one party from the knowledge of some other party's information. Computer systems are designed with the requirement to be secure while providing a public user interface and working with confidential information at the same time. For example, consider a messaging system that offers functionality like editing, sending and receiving messages and keeps the conversations private within a closed circuit of users. So we may specify for the correctness of the system that users outside such a closed circuit must not be able learn information like the contents of the circuit's conversations or not even who is communicating. Furthermore, we certainly specify that answering a message never reveals the communication partner's complete identity or access tokens to the system, although the messaging system itself may have access to all of this information. Also, a service provider may place terminals in public areas which implement additional mechanisms so that any user can access the messaging service from there. The difference between the home computer scenario and the public terminals scenario is that in the former the interface is a network protocol and in the latter it is a graphical and hardware user interface. Moreover, a public terminal has access to potentially more identities and messages than a home computer.

Confidentiality manifests itself in security policies which define the aforementioned information and included parties. These manifestations can be formalised using the noninterference security model [GM82]. In this model an attacker of the computer system is challenged to find out about confidential data just by interacting with the software, that is no direct memory manipulation and no program code manipulation is allowed.

The subject of this thesis is now the formal verification of security policies which are formalised using the noninterference security model for computer systems which are written in the object-oriented programming language Java Card [GJJ96][Sun03].

There are a few approaches to the verification of noninterference on the programming language level, which we will shortly discuss in the remainder of this thesis and compare to the logic-based approach this thesis is based on. Firstly, there are security-type systems. Using programming languages information is stored to and read from memory locations, which divide computer memory. On the level of memory locations information is usually typed and type system-based approaches take a similar path and assign security levels to memory locations. Since information in programs flows by assigning values from and to memory locations, the type system-based verification checks that all assignments happen only to less classified locations [SM03]. This is overapproximated as we will

see. Secondly, there are program dependency graph-based approaches, which pair identifier occurrences with identifier definitions in a graph data structure. The information flow analysis checks that no final evaluation of public memory locations depends on secret memory locations [HKN06], which is achieved by calculating so called program slices for the public memory locations based on the program dependency graph. In contrast to the security-type systems this approach is flow-sensitive and thus less approximating while being carried out equally automated and fast. However, none of these approaches is as precise as the logic-based approaches, which make up the last group of information flow analysis tools for programs. Besides the JavaDL-based approach discussed in this thesis, there is a Hoare-like logic for information flow analysis in object-oriented programs defined by Amtoft et al. [ABB06]. Because of the Hoare calculus similarities, their approach is compositional like the type system-based approaches but also object- and flow-sensitive.

The goal of this thesis is to improve the efficiency of the JavaDL [Bec00] logic approach to noninterference verification by adding a new modality to it that specifically handles the verification of noninterference security policies for Java Card programs. Noninterference can be expressed as a proof obligation in JavaDL without further syntax or semantics changes [SS11] but proofs require two independent program executions of the verified program. Without the new modality the JavaDL approach exploits that its calculus can evaluate final execution states even when the initial states were not completely specified. Adding the assertions that on both executions the same user input was received and that no confidential information leaked produces a first-order logic formula that implies noninterference. This formalisation of noninterference is called self-composition []. The improved JavaDL approach calculates statement by statement which information can leak to the user and compares it in the final state to the information that is allowed to leak to the user.

Consider the Java Card program in Listing 1.1. The noninterference formalisation in JavaDL of the security policy that the program variables `high` and `low` do not interfere during an execution of method `m` contains two diamond modalities and a proof must execute both program occurrences symbolically in order to establish and compare the end states with respect to `low`. This is unnecessary for this example and a proof could consider both conditional branches separately since the initial states assign the same value to `low` and thus execute the same branch for a fixed `low` assignment. An attacker can only compare executions that execute the same conditional branch and the proof complexity can be reduced from comparing four execution paths to only two.

```
1  class C {
2    static int high, low;
3    static void m() {
4      if (low == 0) {
5        low = high + 0;
6      } else {
7        low = high + 23;
8      }
9      low -= high;
10   }
11 }
```

Listing 1.1: Secure program where the branch decision does not depend on secret information

The novelty in this thesis is that the new JavaDL modality mentions the verified program only once. The logic proposed by Amtoft et al. already provides these features but outside the JavaDL context of modelling states and the heap. Moreover, Amtoft et al. enforces a different state equivalence relation that cannot take into consideration which references

are actually visible to an attacker. Essentially, this means that our assumed attacker is not able to observe object creations directly but only the references assigned to local program variables and heap object fields it has access to.

The structure of this thesis is as follows. In the first part we define the new calculus and in the second part we describe how to apply it to information flow analysis. The first part begins with the definition of state equivalence, which is the formal basis for noninterference, in Section 4.1. After that, we define the new modality syntax and semantics in Section 4.2, and Section 4.3 continues with the definition of rules that are added to the JavaDL sequent calculus for the handling of the modality. Combined, the flow modality formulae and rules constitute the JavaDL* as an extension of JavaDL. The second part of the thesis consists of the application of JavaDL* in information flow analysis (Section 5.1) and some example proofs of information flow properties (Section 5.2).

2. Related Work

In this chapter, we map out the research field of programming language level approaches to information flow analysis in computer systems. All approaches discussed here imply a noninterference property for secure programs. The particular noninterference properties differ in the definitions of the indistinguishability relation according to which the approaches compare states, this includes the kinds of evaluations that are public and manipulatable by an attacker. We begin in Section 2.1 with the description of security-type systems and continue in Section 2.2 with program dependency graph-based approaches. Both approaches are approximate and we end this chapter in Section 2.3 with the description of a Hoare-like calculus that poses a logic-based information flow analysis very similar to the JavaDL extension proposed in this thesis.

```
1 class C /*@ low @*/ {
2   private /*@ high @*/ int secret;
3   public /*@ low @*/ void m(/*@low@*/ int input) {
4     /*@ high @*/ int tmp = this.secret;
5     input = tmp;
6     input = 0;
7     return input;
8   }
9 }
```

Listing 2.1: A secure program that is considered insecure by flow-insensitive analyses because of direct information flow

2.1. Security-Type Systems

Security-type systems are a well studied approach to information flow analysis that ensures noninterference between public and secret memory locations for programs that are typed according to the security-typing rules [SM03]. The noninterference property of typed programs results from the fact that a type proof rules out the possibility of an assignment leaking information explicitly or implicitly. Explicit information flows are caused by assignments and implicit flows are caused by statements that decide the program control flow, like conditional statements. It may be impossible to type a program due to an assignment of a public memory location whose execution is determined by the evaluation of a secret condition or which assigns a value that is determined by the evaluation of a secret memory location.

Type system-based approaches annotate the programming language types using security levels. There are usually two security levels called “high” and “low”, but arbitrarily many security classifications in between are in principle possible. Every variable, field, method and class must be declared using both the usual data type and a security level.

As it is the case with the language type system itself, the security-type of a program is supposed to be checked statically. Moreover, the typing rules for the annotated types are still compositional and can efficiently be implemented because only the types of the sub-expressions and sub-programs are needed in order to decide whether an expression or a program is well-typed.

However, there is a fundamental deficit of security-type systems. Assignments that reference memory locations classified by a higher security level than the assigned memory location can never be typed and always rejected by a security-type system analysis. The same applies to assignments that occur in a branch of a conditional statement that evaluates an expression that was assigned a higher security level than the one assigned to the assignment. The strict and local rejection of such statements is not exact since the information leakage can be reverted by successive statements.

The flow-insensitivity of security-type system analyses is illustrated using the following two false negative results. Both example programs cannot be typed because of insecure assignments, although the assignments occur in a control structure context that erases the information leakage before it can be observed by an attacker and the programs are in fact secure.

In the first example (Listing 2.1), line 5 explicitly assigns the value of `tmp` and thus the secret evaluation of `this.secret` to the public variable `input`, which is returned by calls of method `m`. However, line 6 erases all leaked information from the return value before it is finally returned to the caller of the method.

```
1  class C /*@ low @*/ {
2    private /*@ high @*/ boolean secret;
3    public /*@ low @*/ void m(/*@low@*/ int input) {
4      /*@ high @*/ int tmp = this.secret;
5      if (tmp) {
6        input = 1;
7      } else {
8        input = 2;
9      }
10     input = 0;
11     return input;
12   }
13 }
```

Listing 2.2: A secure program that is rejected as being insecure by flow-insensitive analyses because of indirect information flow

The second example is shown in Listing 2.2. Here, the code lines 6 and 8 assign different constant values to the public variable `input`. This would reveal the value of the conditional `tmp` and thus the secret evaluation of `this.secret` if the assignments were not both superseded by the constant assignment in line 10. Thus, before the method returns with the evaluation of `input` the public memory location is assigned the constant value `0` and the method code is in fact secure.

In general, in order for a program to be typed according to a security-type system information can never be temporarily declassified, as it was in the two example programs above, and accumulates on the highest security level.

The examples above do not exploit any object-orientation features and could have been implemented without the class definition. Banerjee and Neumann define a security-type system for a sequential Java-like programming language and prove that it enforces noninterference for typed programs [BN02]. However, they omit loops in favour of recursion and forbid exceptional control flow. Myers even allows exceptional control flow in *JFlow*, which is an extension of the Java programming language by a security-type system [Mye99]. However, an implication of noninterference for typed *JFlow* programs is not proved.

2.2. Program Dependency Graph-Based Approaches

Hammer et al. define the first flow-, context-, and object-sensitive static information flow analysis, which supports unstructured control flow, like exceptions in Java, as well [HKN06]. The analysis is based on the identification of the statements that may determine the evaluation of the attacker observable memory locations in the final program states. Those statements are called slices and are identified using a program dependency graph (PDG) of the analysed program.

The nodes of a PDG are labelled with either statements or expressions and there are two types of edges connecting nodes. The first type of edges connects two statement nodes and means that the first statement assigns a value to a memory location whose evaluation influences the evaluation of an expression the second statement refers to. And, the second edge type is used to connect two statements if the value of the variable assigned in the first statement determines whether the second statement is executed at all.

In the PDG-based approach by Hammer et al., slices are calculated for the public memory locations. The inclusion of secret memory locations in the slice of a public memory location is a necessary condition for the existence of illicit information flow [HKN06]. Therefore, Hammer et al. prove the natural theorem stating that secret-free slices for all public memory locations imply the satisfaction of a noninterference property [HKN06].

The fundamental program analysis tool exploited by this approach is program slicing. The PDG, which slicing applies to, is an abstraction of the analysed program code. There may be more than one program code resulting in the same PDG and PDGs do not encode the exact control flow. Incorporating solely PDGs, the approach cannot perform path condition falsification or functional equality establishment and the information flow analysis conducted is therefore still approximate. The complexity of this approach lies in the precise calculation of the slices and thus the precise PDG generation. In order to improve precision, the PDG-based approach also builds heavily on points-to analysis and dead code erasure [HKN06].

Hammer et al. enable declassification of information by inserting special nodes into the PDG that have a lower output security level than input security level [HKN06]. Otherwise, nodes implicitly have an output security level (the declared security level) that is above the input security level. When a declassification node intercepts a dependency edge connecting a secret source and a public sink, the information flow from the source to the sink is no longer considered illegal by the analysis.

The merits of the PDG-based approach described above over security-type systems include the precise analysis of program fragments like the example shown in Listing 2.3. An execution of that program fragment does not cause observable information flow from secret memory locations (`confidential`) to public outputs (`public`). Therefore, the program fragment is secure despite the assignments of two distinct constants (lines 2 and 4) by the branches of the conditional statement and the branch decision dependence on confidential information.

```
1 if (confidential == 1) {
2   public = 42;
3 } else {
4   public = 17;
5 }
6 public = 0;
```

Listing 2.3: A Program fragment where the final `public` value does not interfere with the initial `confidential` value

2.3. Hoare-Like Logic-Based Approaches

During the work on this thesis, the first approach to add a modality to JavaDL that enables information flow without explicitly executing the analysed program twice was based on the logic proposed by Amtoft et al. [ABB06], which enables structured flow-, context- and object-sensitive analysis.

Amtoft et al. develop a logic for information flow in object-oriented programs with a Hoare-like syntax [ABB06]. Hoare logic was originally defined by Hoare as a first attempt to formally prove functional properties of computer programs [Hoa69]. Therefore, Hoare defined a new formula syntax $P\{Q\}R$, now known as Hoare-triple, where P and R are first-order logic assertions and Q is a program fragment. The semantics of the Hoare-triples can be informally interpreted as R being true in any final execution state of Q when the program fragment was started in an initial execution state satisfying P.

In the logic defined by Amtoft et al., every formula also consists of a precondition, a program fragment and a postcondition but additionally includes a set of modified memory locations. Both pre- and postconditions contain agreement assertions for state equivalence and points-to assertions for handling aliasing in addition to the first-order logic user assertions for functional propositions. Agreement assertions express which memory locations pre- and post-states agree on and points-to assertions express which memory locations refer to the same heap objects.

Following the tradition of Hoare and similar to type systems, the information flow calculus by Amtoft et al. is compositional. This characteristic is reflected in three essential rules. Firstly, there is a rule Seq which breaks the initial program code down to single statements and the further calculus rules reason locally about single statements and how they ensure the type of assertions mentioned above. Secondly and thirdly, there are rules Frame and Conseq. The first one allows symmetric addition of propositions to the pre- and postconditions if they only refer to locations left unmodified by the statement the rule is applied to. The second one strengthens the precondition and weakens the postcondition of a formula. Thus, the rules allow to formulate the calculus rules using assertions that mention only those sub-assertions which are affected by the type of statements the rule applies to. The existence of such rules is essentially made possible by the modifies clause and an implication relation on assertions.

The main distinction between the two logic-based approaches, the Hoare-like approach and the JavaDL approach, is the state equivalence relation applied by the noninterference criterion they both enforce. Amtoft et al. assume an attacker who is always able to observe new object allocations. In contrast, the JavaDL approach applies a state equivalence that models an attacker who can only distinguish states by new object allocations if it can observe a memory location that is assigned the heap reference of the new object before and after the allocation. Moreover, the JavaDL approach improves on the applicability of efficient logic-based information analysis since it can handle unstructured control flow and secret conditionals as well.

3. Assumptions and Limitations

A formalisation of computer security is naturally achieved by defining the assumed attacker of the assessed class of computer systems. Relative to that attacker model a specific computer system is then secure or insecure. In this thesis, we are concerned with the data confidentiality aspect of computer security and the verification that software systems do not leak confidential data. For this purpose, we assume an attacker of the analysed software system who can partially evaluate and modify the start and end states of arbitrarily many executions of the software system's program code. The analysed program code is known to the attacker and must be written in the programming language Java Card.

Since Java Card is an object-oriented programming language, the program states define not only the assignments of all occurring program variables but also the heap. The heap can be seen as a special program variable that is accessed using indices in the form of object references and field identifiers. The partial access we grant an attacker denotes that there are variables and heap locations which are not explicitly revealed to the attacker. Therefore, if two states assign the same values to the accessible memory locations the inaccessible ones can be assigned different values in both states and it is unknown from only the partial access to the states whether this or the opposite is true. Still, an attacker may be able to infer the evaluations of the memory locations not included in the partial access window on the state from the observation of the state and the knowledge of the program code.

It is a crucial restriction on the attacker model that the attacker cannot evaluate arbitrary fields of an object just because an accessible memory location evaluates to the reference of this object. This comes from the fact that the attacker is granted only the knowledge of the state evaluations of specific variable and field identifiers. In particular, since the machine executing the program code is assumed to choose object locations on the heap for newly allocated instances in an unspecified manner, an attacker can only observe which reference type variables and fields refer to the same object or which point to a different object in the final state of an execution than in the initial one. The attacker never knows the complete object unless access is granted to all its fields.

The information flow analysis calculus defined in this thesis has four major limitations. Firstly, the analysis is not termination-sensitive, secondly, the calculus cannot efficiently handle secret dependent control flow and functional end state equivalence without loss of precision, thirdly, it is not enabled for modular proofs and, lastly, it does not take declassification of confidential data into consideration. However, termination-insensitivity seems to be the only limitation future work will not be able to compensate for.

Any analysis carried out using the calculus defined in this thesis is termination-insensitive because infinite loops are not and cannot in general be identified by the calculus. Thus, if the analysed program code contains the possibility for an infinite loop an attacker may be able to manipulate the start state in a way that reveals, firstly, whether an infinite loop is triggered and, secondly, what the constraints on the confidential data are so that an infinite loop can be triggered. In that case, the knowledge of the constraints on the confidential data leaks information.

If a secure program code contains control flow decisions determined by confidential data and its security can only be verified by functionally evaluating the attacker observable parts of the end states, the calculus defined in this thesis must resort to the less efficient self-composition analysis. That is the case if there are attacker observable memory locations whose evaluations are determined by different execution paths and explicit proofs of functional propositions are necessary in order to establish that no confidential data is leaked through these public memory locations. On the contrary, if the evaluations always result from execution paths and assignments not influenced by confidential data the absence of illicit information flow can be inferred using the calculus without knowing the concrete values.

One major feature of logic-based program analysis, which is also essential for applications of our logic-based information flow calculus in software engineering, is the reuse of verification results after they have been proved. Because of reasons similar to why software engineers design modules and mathematicians prove lemmas, the calculus defined in this thesis must provide means for proving and reusing information flow method contracts. As of the completion of this thesis, the calculus lacks this feature and methods must always be inlined and proved secure in each context.

Another restriction that limits the applicability of the calculus defined in this thesis is that memory locations are either public or secret. However, in software engineering it is often necessary to assume only some meta information about memory locations to be public, while their concrete values must still not be leaked. A recurring example requiring such declassification features for its specification and verification is the analysis of a password checker. Our calculus will fail to do so because it must declare mapping from usernames to passwords secret in order to verify the specification that the implementation must not leak passwords. However, specifying the mapping secret is not fine grained enough to account for the perfectly fine leakage of whether a user exists and whether the password file exists. No correct implementation of the password checker example can prevent such leakage and, thus, no secure implementation can be verified using our calculus.

4. Flow Modality Calculus

In this chapter, we present the syntax and semantics of the JavaDL extension JavaDL*. The extended set of formulae includes the new flow modality, which is also defined in this chapter. In order to define the semantics of the flow modality, we first need to define the noninterference state equivalence. The definition of the flow modality syntax requires the definition of a new term type for the specification of lists of public memory locations. Furthermore, we define a set of sequent calculus rules for the derivation of the JavaDL* specific formulae at the end of the chapter.

```
1 class C {
2   Object r;
3
4   void m() {
5     this.r = new Object();
6   }
7 }
```

Listing 4.1: Java Card class definition containing a method implementation that instantiates a new object and assigns its heap address to a reference type field

4.1. An Object-Sensitive State Equivalence

Information flow analysis applying a noninterference security model [GM82] is defined in terms of an equivalence relation $\equiv_L$ which models the indistinguishability of states from the attacker's perspective on the initial and final program states. An attacker is allowed to observe the states by learning about the values assigned by each state to a limited set of low program variables and object fields. Within the noninterference security model, we examine two independent executions of the analysed program and check whether their final states are related with respect to $\equiv_L$ based on the assumption that their initial states are related. In particular, the noninterference criterion requires programs to transform equivalent initial states s_1 and s_2 ($s_1 \equiv_L s_2$) into equivalent states s_1' and s_2' ($s_1' \equiv_L s_2'$) respectively, but only if both executions terminate.

The equivalence relation applied in the instances of the noninterference security model for Java Card programs due to Scheben and Schmitt [SS11] defines two states equivalent if and only if all low program variables and object fields are assigned the same value in each state, regardless of whether they are of primitive or reference type. In that case, $\equiv_L$ is formally defined by Definition 4.1.1.

Definition 4.1.1 (JavaDL noninterference state equivalence)**.** Given a JavaDL signature $\Sigma = (\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}^{Unique}, \mathcal{P}, \alpha, Prg)$, a JavaDL Kripke structure $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$, a set of program variables $V \subseteq \mathcal{PV}$, a set of heap locations $L \subseteq allLocs$ and two states $s_1 \in \mathcal{S}$ and $s_2 \in \mathcal{S}$, s_1 and s_2 are related with respect to $\equiv_{\langle V,L \rangle}$ (in symbols $s_1 \equiv_{\langle V,L \rangle} s_2$) if and only if both of the following conditions hold.

- $s_1(\mathtt{v}) = s_2(\mathtt{v})$ for all $\mathtt{v} \in V$
- $I(select)(s_1(\mathtt{heap}), o, f) = I(select)(s_2(\mathtt{heap}), o, f)$ for all $\langle o, f \rangle \in L$

■

Considering the example program in Listing 4.1, the definition of the state indistinguishability above (Definition 4.1.1) may be too restrictive if reference type variables or fields are included in V or L.

Supposing that `r` is contained in L, this method body almost never transforms $\equiv_{\langle V,L \rangle}$-equivalent initial states into $\equiv_{\langle V,L \rangle}$-equivalent final states. The reason for that is that the Java Card virtual machine executing the method implementation generates the reference values assigned by line 5 (`this.r = new Object();`) in an under-specified manner and the values are likely to differ between executions. However, obviously there are no secrets leaked to an attacker since all declared memory locations are public and there are no specified secrets that could be leaked. We conclude that the application of $\equiv_{\langle V,L \rangle}$ in the noninterference security model gives false negative results and that its definition does not suffice to model our assumed attacker precisely.

```
1  class C {
2    Object r;
3    boolean secret;
4
5    void m() {
6      if (this.secret) {
7        this.r = new Object();
8      }
9    }
10 }
```

Listing 4.2: Java Card class definition containing a method implementation that only instantiates a new object and assigns its heap address to a reference type field if another field evaluates to true

A refinement of the state indistinguishability relation from above (Definition 4.1.1) could be to require equality of the state assigned values only with respect to a bijective map from the low object space of the first state to the low object space of the second state. The existence of such a map is the first prerequisite in the state equivalence enforced by the information flow calculus for object-oriented programs due to Amtoft et al. [ABB06].

Considering another program example (Listing 4.2), a second prerequisite similar to the one defined by Amtoft et al. is in fact necessary.

In that example, the reference value is only generated if `secret`, which we do not include in the specification of low memory locations for this analysis, evaluates to true. An attacker may execute the implementation of method `m` in class `C` and compare the final evaluation of `r` to its initial evaluation. As a result, the attacker learns about the evaluation of `secret` because the virtual machine will assign a new reference value to `r` if it executes line 7 (`this.r = new Object();`). The attacker will conclude `secret == true` if the evaluation of `r` changes after the initial state and `secret == false` otherwise. Nevertheless, there is a bijective map between the two low object spaces in the noninterference security model which maps the reference value assigned to `r` in the first final state to the reference value assigned to it in the second final state. The map is trivially bijective since each low object space consists solely of the object referenced by `r`.

This leads to the following definition of the noninterference state equivalence relation $\equiv^*_{\langle V,L\rangle}$ given by Scheben and Schmitt et al. [SSB+13, Definition 2, Definition 3]. It introduces a bijective map between object spaces and requires the values of reference type variables and fields in each state to be only related with respect to this map instead of being identical. Additionally, the map that relates the final states must not alter the mapping of objects already included in the initial object space.

Applying this definition in the noninterference security model corresponding to the second example program (Listing 4.2), the final states of two arbitrary executions may be no more related. In particular, if the evaluation of `secret` in the initial states differ then the map that maps the reference value of `r` in the first final state to its value in the second final state will associate two distinct objects with a single object in the first domain and will not be a function anymore.

Definition 4.1.2 (Agreement of states)**.** For a JavaDL signature $\Sigma = (\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}^{Unique}, \mathcal{P}, \alpha, Prg)$, a JavaDL Kripke structure $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$, a finite set of program variables $V \subseteq \mathcal{PV}$, a finite set of heap locations $L \subseteq allLocs$ and two states $s_1 \in \mathcal{S}$ and $s_2 \in \mathcal{S}$ define $s_1 \equiv^*_{\langle V,L\rangle} s_2$ if and only if there is an injective type-preserving and array-length-preserving function η from the set of objects referenced by variables $\mathtt{v} \in V$

and locations $\langle o, f\rangle \in L$ in s_1 to the set of objects referenced by V and L in s_2, so that all of the following conditions hold.

- $s_1(\mathtt{v}) = s_2(\mathtt{v})$ if $\mathtt{v} \in V$ and $\alpha(\mathtt{v}) \not\sqsubseteq \mathit{Object}$
- $\eta(s_1(\mathtt{v})) = s_2(\mathtt{v})$ if $\mathtt{v} \in V$ and $\alpha(\mathtt{v}) \sqsubseteq \mathit{Object}$
- $I(\mathit{select}_A)(s_1(\mathtt{heap}), o, f) = I(\mathit{select}_A)(s_2(\mathtt{heap}), \eta(o), f)$ if $\langle o, f\rangle \in L$ and $\alpha(f) = A \not\sqsubseteq \mathit{Object}$
- $\eta(I(\mathit{select}_A)(s_1(\mathtt{heap}), o, f)) = I(\mathit{select}_A)(s_2(\mathtt{heap}), \eta(o), f)$ if $\langle o, f\rangle \in L$ and $\alpha(f) = A \sqsubseteq \mathit{Object}$

Instead of $s_1 \equiv^*_{\langle V,L\rangle} s_2$, we also say that s_1 and s_2 agree on $\langle V, L\rangle$. Furthermore, we call the agreement function partial isomorphism. As Scheben and Schmitt et al. observe, we state here that the partial isomorphism is uniquely defined by the choice of memory locations and program states. ■

Since the bijection introduced by Definition 4.1.2 substitutes references with references in both the initially and finally attacker accessible terms so that the object types and the evaluations of declassified fields continue to coincide, we base our noninterference analysis on the assumption that this state equivalence relation models the attacker abilities both correctly and completely.

4.2. Flow Modality Definition

In this section, we define a new modality for the JavaDL logic specifically designed for information flow analysis using the JavaDL sequent calculus. The flow modality $[\![\cdot \mid \cdot \mid \cdot]\!]$ asserts that all executions of a given Java Card program transform low equivalent states into again low equivalent states. Two states are low equivalent if they agree on a given set of low memory locations and we use the agreement formalisation given by Definition 4.1.2.

Since low equivalence of states is always defined in terms of a set of low memory locations, the following definition gives rise to a special syntax (*RefSet* expression) used for specifying those sets as JavaDL terms within the flow modality.

Definition 4.2.1 (*RefSet* expression)**.** For a fixed JavaDL signature $\Sigma = (\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}^{Unique}, \mathcal{P}, \alpha, Prg)$ a term of type *RefSet* is a pair $\langle V, L \rangle$, where

- $V \subseteq \mathcal{PV}$ is a finite set of primitive or reference type program variables
- L is a term of type *LocSet* that always evaluates to a finite set of heap locations

Given a JavaDL Kripke structure $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$, the objects referenced by a *RefSet* expression $R = \langle V, L \rangle$ in a state $s \in \mathcal{S}$ are given by the set $Obj^s(R)$ of all $s(\mathtt{v})$, if $\mathtt{v} \in V$ is a reference type program variable, and $I(select_A)(s(\mathtt{heap}), o, f)$, if $\langle o, f \rangle \in val_{\mathcal{K},s,\beta}(L)$ and f is a reference type field of type $\delta(f) = A$. ■

From time to time, we need to talk about which program variables or heap locations are contained in a *RefSet* expression. Especially in premises of calculus rules for the flow modality, we are required to express such propositions in formal syntax. Thus, the following two definitions introduce a special semantics for the well-known predicate symbols $\in$ and $\subseteq$.

Definition 4.2.2 (*RefSet* expression element-of relation)**.** For a fixed JavaDL signature $\Sigma = (\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}^{Unique}, \mathcal{P}, \alpha, Prg)$, a *RefSet* expression $R = \langle V, L \rangle$, a program variable $\mathtt{v} \in \mathcal{PV}$, an *Object* term o and a *Field* term f, $\mathtt{v} \in R$ and $\langle o, f \rangle \in R$ are formulae.

$\mathtt{v} \in R$ and $\langle o, f \rangle \in R$ evaluate to true in a JavaDL Kripke structure $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$, a state $s \in \mathcal{S}$ and a logic variable environment β if and only if $\mathtt{v} \in V$ and $\langle o, f \rangle \in val_{\mathcal{K},s,\beta}(L)$ for the sets V and $val_{\mathcal{K},s,\beta}(L)$ respectively. ■

Definition 4.2.3 (*RefSet* expressions subset relation)**.** For two *RefSet* expressions R_1 and R_2 is $R_1 \subseteq R_2$ a formula.

$R_1 \subseteq R_2$ evaluates to true in a JavaDL Kripke structure $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$, a state $s \in \mathcal{S}$ and a logic variable environment β if and only if for all $\mathtt{v}$ and $\langle o, f \rangle$ so that $\mathtt{v}, \langle o, f \rangle \in R_1$ evaluates to true the formulae $\mathtt{v} \in R_2$ and $\langle o, f \rangle \in R_2$ evaluate to true. ■

The following definition of legal Java Card program fragments, which are used to specify the programs within the flow modality, is equal to the definition of legal program fragments in [Wei11, Definition 5.2].

Definition 4.2.4 (Java Card legal program fragment)**.** For a JavaDL signature $\Sigma = (\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}^{Unique}, \mathcal{P}, \alpha, Prg)$ a legal program fragment π is a sequence of Java Card statements so that extending Prg by the following definition of type `C` yields a legal Java Card program according to [GJJ96], with the exception that π may reference members and types that are not visible from within `C` and, furthermore, π may contain `method-frame` statements. Hereby, $a_1, \ldots, a_n \in \mathcal{PV}$ are program variables and $T_1, \ldots, T_n \in \mathcal{T}$ are types. Furthermore, for a legal program fragment π_b and two program variables `v`, `o`, the statement `method-frame(result=v,this=o) {`π_b`}` is called a method frame. Within a method frame `this` references have the same evaluation as `o` and `return` statements assign the return value to `v` before exiting the method frame.

```
1  class C {
2    static void m(T_1 a_1, ..., T_n a_n) {
3      π
4    }
5  }
```

■

Using the *RefSet* expression and legal program fragment definitions, we are now able to define the flow modality syntax.

Definition 4.2.5 (Flow modality syntax)**.** For a JavaDL signature $\Sigma = (\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}^{Unique}, \mathcal{P}, \alpha, Prg)$, a legal program fragment π, two *RefSet* expressions F and T is $[\![\pi \mid F \mid T]\!]$ a formula. ■

We are now able to define the set of JavaDL* formulae, which essentially consists of JavaDL formulae that may contain the just defined flow modality formulae.

We obtain the set of JavaDL* formulae Fma'_Σ by extending the set Fma_Σ from [Wei11, Definition 5.3].

Definition 4.2.6 (JavaDL* formula)**.** For a fixed JavaDL signature $\Sigma = (\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}^{Unique}, \mathcal{P}, \alpha, Prg)$, a JavaDL* formula is any formula contained in the set Fma'_Σ, which in turn is defined by the following grammar.

$$
\begin{aligned}
Fma'_\Sigma ::= & true \mid false \mid p(Term^1_\Sigma, \ldots, Term^n_\Sigma) \mid \neg Fma'_\Sigma \\
& \mid Fma'_\Sigma \wedge Fma'_\Sigma \mid Fma'_\Sigma \vee Fma'_\Sigma \mid Fma'_\Sigma \rightarrow Fma'_\Sigma \mid Fma'_\Sigma \leftrightarrow Fma'_\Sigma \\
& \mid \forall Ax; Fma'_\Sigma \mid \exists Ax; Fma'_\Sigma \\
& \mid [\pi] Fma'_\Sigma \mid \langle \pi \rangle Fma'_\Sigma \mid \{Upd_\Sigma\} Fma'_\Sigma \mid [\![\pi \mid F \mid T]\!]
\end{aligned}
$$

Here, π is a legal program fragment in the context of Prg, F and T are *RefSet* expressions. Furthermore, $p \in \mathcal{P}$ is a predicate symbol, $A \in \mathcal{T}$ is a type and $x \in \mathcal{V}$ is a logic variable. ■

We end this section with the definition of the semantics of the flow modality and a proof that an equivalent JavaDL formula exists. The semantics says that the memory locations in a *RefSet* expression T depend at most on the memory locations in another *RefSet* expression F. Furthermore, the semantics expresses the noninterference property of secure programs if the two *RefSet* expressions are equal.

The flow modality semantics we define here conforms to the prerequisites Scheben and Schmitt et al. prove the noninterference property compositional for [SSB+13, Theorem 3]. In addition to classic noninterference with respect to the object-sensitive state equivalence relation defined in Section 4.1, the flow modality semantics requires that objects observable in the final state are either new or were also observable in the initial state. The compositionality result by Scheben and Schmitt et al. and thus the precise flow modality semantics are essential to the statement-wise interpretation of the analysed program fragment.

Definition 4.2.7 (Flow modality semantics)**.** For a JavaDL Kripke structure $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$ and a logic variable assignment β a flow modality formula $[\![\pi \mid F \mid T]\!]$ evaluates to true in a state $s \in \mathcal{S}$, in symbols $(\mathcal{K}, s, \beta) \models [\![\pi \mid F \mid T]\!]$, if and only if for every state $t \in \mathcal{S}$ so that

- π terminates when started in both s and t
- s and t agree on F by some partial isomorphism η

the following three conditions hold for the respective unique end states $s' \in \mathcal{S}$ and $t' \in \mathcal{S}$, in symbols $\langle s, s' \rangle \in \rho(\pi)$ and $\langle t, t' \rangle \in \rho(\pi)$,

- s' and t' agree on T by some partial isomorphism η'
- η' extends η, that is objects observable in both s and s' are mapped to the same objects by η and η', in symbols

$$\forall o \in Obj^s(F) \cap Obj^{s'}(T)(\eta(o) = \eta'(o)) \tag{4.1}$$

- all objects referenced by T in s' that existed already in s are also referenced by F in s, in symbols

$$\{o \in Obj^{s'}(T) | I(select_{Boolean})(s(\texttt{heap}), o, I(created)) = tt\} \subseteq Obj^s(F)$$

Instead of $[\![\pi \mid F \mid T]\!]$, we also say that π allows information to flow at most from F to T.

The following theorem (Theorem 1) states that the the flow modality semantics can be expressed by a JavaDL formula. The proof given here is based on the corresponding results for the flow predicate semantics defined by Scheben and Schmitt et al. [SSB+13]. In fact, the flow modality formula consists of two sub-formulae. The first sub-formula expresses noninterference and the extension condition as already expressed by the aforementioned flow predicate semantics, whereas the second formula reasons only about the program execution in the state the formula is evaluated in and expresses the observability condition of the flow modality semantics, which is given last in Definition 4.2.7.

Theorem 1. *For a JavaDL signature $\Sigma = (\mathcal{T}, \preceq, \mathcal{V}, \mathcal{PV}, \mathcal{F}, \mathcal{F}^{Unique}, \mathcal{P}, \alpha, Prg)$, a legal program fragment π and RefSet expressions F, T, there is a JavaDL formula $\Phi_{\pi,F,T}$ so that the following holds for every Kripke structure $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$, logic variable assignment β and state $s \in \mathcal{S}$.*

$$(\mathcal{K}, \beta, s) \models [\![\pi \mid F \mid T]\!] \Leftrightarrow (\mathcal{K}, \beta, s) \models \Phi_{\pi,F,T}$$

Proof. From [SSB+13, Theorem 1], we obtain a JavaDL formula $\Phi'_{\pi,F,T}$ that is equivalent to the flow modality semantics without the condition that objects observable in the final state and existent in the initial state are also observable in the initial state (observability). In fact, the flow modality semantics would be equivalent to the flow predicate semantics defined in [SSB+13, Definition 4] without that condition. The formula $\Phi'_{\pi,F,T}$ has the following pattern. Since Scheben and Schmitt et al. specify the low memory locations as sequences of JavaDL terms, which they call observation expressions, we implicitly transform our finite sets of program variable identifiers and heap locations to JavaDL sequences in the following formulae.

$$\begin{aligned} \Phi'_{\pi,F,T} \equiv \forall Heap\ h_t, h_{s'}, h_{t'}\ \forall T\ o_t\ \forall T_r\ r_{s'}, r_{t'}\ \forall \ldots, v_s, v_{s'}, v_{t'}, \ldots \\ Agree_{pre}(F) \land \langle \pi \rangle \{save\ s'\} \land \{in\ t\} \langle \pi \rangle \{save\ t'\} \longrightarrow (Agree_{post}(T) \land Ext) \end{aligned} \tag{4.2}$$

Formula 4.2 is evaluated in state s and models a second state t which is low equivalent to s with respect to the *RefSet* expression F. The two diamond modalities on the left-hand side

of the implication determine the final states s' and t' of the two independent executions of the program fragment π with the initial states s and t respectively.

The update pattern $\{in\ \ldots\}$ is resolved to a succession of updates that establish the specified state by initialising the heap, the current object reference `this` and the local program variables. The unknown heap h_t initialises the special program variable `heap` in t and there is no initialiser for the heap in s since the program variable for the heap is implicitly given by the state s the formula is evaluated in. The same applies to the initial values o_t and v_t assigned to the special program variable `this` and the local variables `v` in t. The formula quantifies over the values of program variables `v` occurring in the program fragment π without being declared. The corresponding update pattern $\{save\ \ldots\}$ resolves to a succession of equalities that ensure that the evaluations of the logic variables $h_{s'}$, $h_{t'}$, $e_{s'}$, $e_{t'}$, $r_{s'}$, $r_{t'}$, $v_{s'}$, $v_{t'}$ conform to the values of the heap, the exception state, a `return` statement and the local variables in the specified final states s' and t' respectively. The special program variables `heap` and `exc` are evaluated in order to obtain the heap and exception states respectively. Moreover, the return value is chosen randomly unless the analysed program fragment is a method. In that case, $r_{s'}$ and $r_{t'}$ are ensured to conform to the evaluation of the expression argument passed to the `return` statement which was executed last. Apart from quantification, the logic variable correspondences are mainly used for reasoning about states a formula is not evaluated in and hence in the formulae inserted for the update patterns $\{in\ \ldots\}$.

The formula $Agree_{pre}(F)$ abbreviates a formula that models the agreement relation between s and t with respect to F. If $Agree_{pre}(F)$ evaluates to true in state s, then t is low equivalent to s and agrees with s on F by id. That additional constraint on the presupposition in the agreement definition that any partial isomorphism exists is proved equivalent by Scheben and Schmitt et al.. More precisely, a program transforms low equivalent states into low equivalent states if and only if it transforms states that agree on the set of low memory locations by id into low equivalent states [SSB+13, Lemma 5]. The abbreviation $Agree_{post}(T)$ stands for a formula that implies the agreement of s' and t' on T by the uniquely determined partial isomorphism. Lastly, the abbreviation Ext is resolved to a formula that implies that the partial isomorphism that maps the objects observable in s' to the objects observable in t' extends id, which is the agreement isomorphism between the object spaces observable in the initial states s and t respectively. Both $Agree_{pre}(F)$ and $Agree_{post}(T)$ evaluate the heap and the local program variables in both s and t as well as s' and t' respectively, and the substituted formulae can refer to those evaluations because of the quantified logic variables and the update patterns $\{save\ s'\}$ as well as $\{save\ t'\}$.

Based on this pattern, we now define the formula $\Phi_{\pi,F,T}$, which we will prove equivalent to the flow modality semantics afterwards.

$$\Phi_{\pi,F,T} \equiv \forall Heap\ h_t, h_{s'}, h_{t'}\ \forall T\ o_t\ \forall T_r\ r_{s'}, r_{t'}\ \forall \ldots, v_t, v_{s'}, v_{t'}, \ldots$$
$$Agree_{pre}(F) \land \langle\pi\rangle\{save\ s'\} \land \{in\ t\}\langle\pi\rangle\{save\ t'\} \longrightarrow (Agree_{post}(T) \land Ext \land Obs(F,T))$$

In the definition of $\Phi_{\pi,F,T}$, we demand the observability condition Obs in the post-state in addition to the agreement and extension conditions $Agree_{post}(T)$ and Ext respectively. Obs (Formula 4.3) is an abbreviation for the JavaDL formulation of the additional condition that occurs in the flow modality semantics but does not occur in the flow predicate semantics. Because of space limitations, we write $h_{s'}(o,f)$ and $\texttt{heap}(o,f)$ instead of $select(h_{s'}, o, f)$ and $select(\texttt{heap}, o, f)$ respectively in the definition of Obs.

$$
\begin{aligned}
&Obs(F,T) \equiv \\
&\bigwedge_{\mathtt{v}\in T} instanceof_{Object}(v_{s'}) \wedge \mathtt{heap}(v_{s'}, created) = True \longrightarrow Obs'(F, v_{s'}) \\
&\wedge \bigwedge_{\langle o,f\rangle \in T} instanceof_{Object}(h_{s'}(o,f)) \wedge \mathtt{heap}(h_{s'}(o,f), created) = True \longrightarrow Obs'(F, h_{s'}(o,f))
\end{aligned}
\tag{4.3}
$$

$$
Obs'(F,r) \equiv \bigvee_{\mathtt{v}\in F} \mathtt{v} = r \quad \vee \bigvee_{\langle o,f\rangle \in F} \mathtt{heap}(o,f) = r
$$

Firstly, we assume $(\mathcal{K},\beta,s) \models [\![\ \pi \mid F \mid T\ ]\!]$ and show $(\mathcal{K},\beta,s) \models \Phi_{\pi,F,T}$. Therefore, we instantiate the universal quantifiers with arbitrary but fixed domain values and assume the left-hand side of the implication in $\Phi_{\pi,F,T}$. Then, the presuppositions of the flow modality semantics are fulfilled, particularly that the initial states agree on F by some partial isomorphism (id in this case) and that both executions of π with initial states s and its low-equivalent companion t, which is determined by the quantifier instantiations h_t, o_t, v_t terminate. Furthermore, the flow modality semantics dictate that the final states s' and t' agree on T by another partial isomorphism that extends the agreement isomorphism of the initial states and that all objects observable in the final state are either not existent or also observable in the initial state. The final state agreement and isomorphism extension make the sub-formulae $Agree_{post}$ and Ext true.

It remains to be shown that the third implication of the flow modality semantics makes the sub-formula $Obs(F,T)$ true. Therefore, we fix an arbitrary object $\nabla \in \mathcal{D}^{Obj}$ observable through T in s' that existed in s already, that is there exists a program variable $\mathtt{v} \in T$ or a location $\langle o,f\rangle \in val_{\mathcal{K},\beta,t}(T)$ so that $t(\mathtt{v}) = \nabla$ or $I(select)(t(\mathtt{heap}), o, f) = \nabla$ respectively and $I(select)(s(\mathtt{heap}), \nabla, I(created)) = tt$. Since the set of objects observable in the final state and existent in the initial state is a subset of the objects observable in the initial state, there exists a program variable $\mathtt{v'} \in F$ or a location $\langle o', f'\rangle \in val_{\mathcal{K},\beta,s}(F)$ so that $t(\mathtt{v'}) = \nabla$ or $I(select)(s(\mathtt{heap}), o', f') = \nabla$ respectively. We conclude that for any such object ∇ the right-hand side $Obs'(F,\nabla)$ of the implication in $Obs(F,T)$ is true. Since ∇ was arbitrarily chosen, $Obs(F,T)$ evaluates to true and $\Phi_{\pi,F,T}$ evaluates to true in state s.

Secondly, we assume $(\mathcal{K},\beta,s) \models \Phi_{\pi,F,T}$ and show $(\mathcal{K},\beta,s) \models [\![\ \pi \mid F \mid T\ ]\!]$. Scheben and Schmitt et al. showed that it is equivalent to show noninterference for states that agree by id on F and to show noninterference for states that agree on F by any partial isomorphism, we actually assume a transformation of $\Phi_{\pi,F,T}$ that substitutes $Agree_{pre}(F)$ with $Agree_{type\&prim}(F) \wedge Agree_{obj}(F)$ in $\Phi_{\pi,F,T}$. The first operand ensures that the unique partial isomorphism preserves types as well as primitive type values and the second operand ensures that it is an injection between object spaces. We fix an arbitrary state t that agrees with s on F so that executions of π with either s or t as initial state terminate. The agreement assumption makes $Agree_{type\&prim}(F) \wedge Agree_{obj}(F)$ true and the termination assumptions make the diamond modality formulae true for a set of instantiations of the universal quantifiers. Since $\Phi_{\pi,F,T}$ evaluates to true in s and the left-hand side of the implication evaluates to true, the right-hand side of the implication must be true. Thus, $Agree_{post}(T)$, Ext and $Obs(F,T)$ are all true and we know that these imply that s' and t' agree on T by a partial isomorphism that extends the isomorphism by which s and t agree on F.

In order to conclude $(\mathcal{K},\beta,s) \models [\![\ \pi \mid F \mid T\ ]\!]$, we must show that the set of objects observable in s' and existent in s is a subset of $Obj^s(F)$. Therefore, we fix an arbitrary object $\nabla \in Obj^{s'}(T)$ which satisfies $I(select)(s(\mathtt{heap}), \nabla, I(created)) = tt$. Because ∇ is

observable through T in s', there is a program variable $\mathtt{v} \in T$ or a heap location $\langle o, f \rangle \in val_{\mathcal{K},\beta,s'}(T)$ that evaluates to ∇ in s' and since $Obs(F,T)$ is true, $Obs'(F,\nabla)$ is true and there exists another program variable $\mathtt{v'} \in F$ or heap location $\langle o', f' \rangle \in val_{\mathcal{K},\beta,s}(F)$ that evaluates to ∇ in s. Hence, we have assured that ∇ is observable both through T in s' and through F in s. Since ∇ was arbitrarily chosen, we proved the subset relation between the objects observable in the final state and the objects observable in the initial state. Thus, the semantics of the flow modality in state s is satisfied and $(\mathcal{K}, \beta, s) \models [\![\pi \mid F \mid T]\!]$ is indeed a true proposition.

Finally, we proved that for every flow modality formula there is a JavaDL formula so that for a given triple consisting of a Kripke structure, a logic variable assignment and a state either both are true or both are false. □

4.3. Flow Modality Rule Set

In this section, we propose a set of inference rules for the JavaDL* calculus (Definition 4.3.2) and also give soundness proofs. The set of rules proposed is an extension of the JavaDL rule set that includes rules for the flow modality (Definition 4.2.5). Most of the JavaDL* rules are listed in the appendix of this thesis. However, the rules defined in this section are core rules that illustrate the recurrent principles of the JavaDL extension. In addition, correctness proofs are given for those rules, that is it is proved that they imply the noninterference criterion (Definition 4.2.7) for the analysed program code if a proof exists for each premise.

Before we define the new rules, we recall the definitions of sequents and rules as well as the notion of a proof as they are already used in the JavaDL logic.

A JavaDL* sequent is a JavaDL sequent [Bec00, p. 14] with the exception that antecedent and succedent are finite sets of JavaDL* formulae.

Definition 4.3.1 (JavaDL* sequent)**.** A JavaDL* sequent is a pair $\langle \Gamma, \Delta \rangle \in Seq'_\Sigma$ where the set of all sequents is $Seq'_\Sigma = \mathcal{P}(Fma'_\Sigma) \times \mathcal{P}(Fma'_\Sigma)$.

However, we will use the alternative notation $\Gamma \Rightarrow \Delta$ for sequents throughout the text. Moreover, we will write $\Gamma, \gamma \Rightarrow \Delta, \delta$ instead of $\Gamma \cup \{\gamma\} \Rightarrow \Delta \cup \{\delta\}$ where γ and δ are JavaDL* formulae.

The semantics of a JavaDL* sequent $\Gamma \Rightarrow \Delta$ is the same as for a JavaDL sequent and defined for a JavaDL Kripke structure $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$, a program state $s \in \mathcal{S}$ and a logic variable assignment β by the evaluation of the JavaDL* formula $\bigwedge \Gamma \longrightarrow \bigvee \Delta$ with respect to $\mathcal{K}$, s and β. ■

A JavaDL* rule is a JavaDL rule [Bec00, p. 16] except that both the premises and the conclusion are JavaDL* sequents.

Definition 4.3.2 (JavaDL* rule)**.** A JavaDL* rule is a binary relation $r \subseteq Seq'^*_\Sigma \times Seq'_\Sigma$.

The semantics of a JavaDL* rule is that if all premises are valid then the conclusion is valid. ■

A JavaDL* proof is a JavaDL proof [Bec00, p. 16] with the exception that the nodes are labelled with JavaDL* sequents and the edges are labelled with either JavaDL rules or one of the schematic rules defined either in Section 4.3, Section 4.4 or in Appendix A of this thesis.

Since the JavaDL rules and the rules defined in this thesis are sound, we do not separately define proof trees but only proofs, which are closed proof trees that only contain edges labelled with sound rules.

Definition 4.3.3 (JavaDL* proof)**.** A JavaDL* proof is a finite and directed tree where

- Inner nodes are labelled with JavaDL* sequents
- Leaf nodes are labelled with the symbol $*$
- All edges from a parent node to its child nodes are labelled with the same JavaDL* rule
- Each parent node sequent is derivable from its child node sequents using the rule the edges from the parent to the children are labelled with or the only child node is a leaf node and the rule has no premises

■

Almost all flow modality rules are very similar to the respective rules defined in [Wei11, Figure 5.9] for the JavaDL modalities box ($[\cdot]$) and diamond ($\langle\cdot\rangle$). Each rule definition applies to a specific Java Card syntax element and there is also a rule for the empty modality.

The intuition behind the rules proposed here is that they synchronously execute the program fragment within the modality in two states that assign equal values to a subset of memory locations (Definition 4.1.2), and determine the set of agreed on memory locations in the final execution states.

In the following definitions, `v` and `c` are primitive type program variables, `r` and `s` are reference type program variables, f is a field and C is a class type. Moreover, π_1, π_2 and π are Java Card program fragments (Definition 4.2.4) and μ is an update [Bec00]. There are also two Java Card syntax placeholders θ and ρ used to refer to Java Card block headers (like `try`) and the remainder program code respectively. Moreover, it is assumed that the program fragments including those in the modalities contain no method calls, only instance allocation (`alloc()`) calls, no `for`-loops, no declare and assign statements, only a single declaration per statement, no operate and assign statement (like `+=`), only simple conditional expressions and no field accesses in complex expressions. All those normalisations can be obtained by unfolding method and constructor calls as well as assigning complex conditional expressions and field accesses to temporary program variables and replacing their original occurrences with the freshly declared program variables.

We also introduce some implicit assumptions of the ensuing soundness proofs. Let $\mathcal{K} = (\mathcal{D}, \delta, I, \mathcal{S}, \rho)$ be an arbitrary but fixed JavaDL Kripke structure. Throughout this section, $s \in \mathcal{S}$ and $t \in \mathcal{S}$ denote low equivalent program states and the formulae are always evaluated in s. However, this is never written out. Concerning the concrete low equivalence, we assume the equivalent variant of the flow modality semantics where the initial states in the security model are assumed to agree by *id* on the low memory locations [SSB$^+$13, Lemma 5]. Often, we prove a stronger version of noninterference (namely $flow^{**}(\cdot,\cdot,\cdot,\cdot)$) which is defined by Scheben and Schmitt et al. [SSB$^+$13, Definition 6]. For a program fragment π, two *RefSet* expressions F, T and a set of objects O, $flow^{**}(\pi, F, T, O)$ implies the semantics of $[\![\ \pi \mid F \mid T\ ]\!]$ [SSB$^+$13, Lemma 8]. If noninterference holds in the stronger semantics, the soundness proofs get simpler because the partial isomorphism between the final states in the stronger semantics is only defined on the newly created objects. The additional fourth predicate argument not present in the flow modality gives exactly the set of objects that exist in the final state but did not exist in the initial program state.

The first three rules handle assignments, in particular assignments of local variables, object fields and array elements. The latter two types of assignments update the heap while the former type updates only the state and does not alter the heap. Each rule transforms assignments into JavaDL updates of the heap or the assigned program variables.

Definition 4.3.4 (assignLocal*)**.** The program fragments handled by the assignLocal* rule are assignments of complex expressions α to program variables `v`.

$$\frac{\Gamma \Rightarrow \Delta, vars(\alpha) \subseteq F \qquad \Gamma \Rightarrow \Delta, \{\mu\}\{\mathtt{v} := \alpha\}[\![\ \theta\rho \mid F \cup \{\mathtt{v}\} \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v}\ {=}\alpha; \rho \mid F \mid T\ ]\!]}$$

■

Proof. Firstly, we conclude $val_{\mathcal{K},s,\beta}(\alpha) = val_{\mathcal{K},t,\beta}(\alpha)$ from the premise $vars(\alpha) \subseteq F$, since for every variable `x` that occurs in α we have $\mathtt{x} \in F$ and thus $s(\mathtt{x}) = t(\mathtt{x})$. Each final

```
1 r0.f = 0;
2 r1.f = 0;
3 r = s ? r0 : r1;
4 r.f = 1;
5 v = r0.f;
```

Listing 4.3: Java Card program fragment that leaks the evaluation of `s` through `v`

execution state s' and t' is determined by the state update introduced by the operational Java Card semantics of the assignment statement:

$$s' = s[\mathtt{v} \mapsto val_{\mathcal{K},s,\beta}(\alpha)]$$
$$t' = t[\mathtt{v} \mapsto val_{\mathcal{K},t,\beta}(\alpha)]$$

Together with the equality of the evaluation of the expression α in the two initial states s and t, we conclude $flow^{**}(\mathtt{v = }\alpha\mathtt{;}, F, F \cup \{\mathtt{v}\}, \emptyset)$, which implies $[\![\ \mathtt{v = }\alpha\mathtt{;}\ |\ F\ |\ F \cup \{\mathtt{v}\}\ ]\!]$ [SSB+13, Lemma 8]. From this and the second rule premise $[\![\ \theta\rho\ |\ F \cup \{\mathtt{v}\}\ |\ T\ ]\!]$ we conclude $[\![\ \theta\,\mathtt{v =}\alpha\mathtt{;}\ \rho\ |\ F\ |\ T\ ]\!]$ by the compositionality theorem [SSB+13, Theorem 3]. □

Field assignments result in updates of the special program variable `heap` at an index, which is similar to the index-based access of arrays. In fact, JavaDL models the Java heap using an array theory [Wei11, Section 4.2.3]. This is part of the reason why the reference to a heap object ("the index") may be illegal, which is exactly the case if the reference is the special reference `null`. When `null` is dereferenced the Java Virtual Machine throws an exception without altering the heap. The rule handling field assignments must, therefore, take an alternative execution path into consideration. Since the JavaDL* calculus depends on the fact that the two executions observed in the noninterference security model execute the same statements, the assignment rule can only be applied if either both run into the exceptional state or none does. That is the first reason why for the first rule premise the reference variable must be proved among the low memory locations. The second reason comes from the possibility of an attack if the assignment updates different heap objects in each state. Consider the example program fragment shown in Listing 4.3 and the low memory location specification $R = \{\mathtt{v}\}$, there an attacker learns that `s` is assigned `true` if `v` evaluates to `1` and that `false` is assigned if the evaluation of `v` is `0`. However, if the rule for field assignments did not presuppose that `r` was a low memory location, then the calculus would simply infer that the objects `r` could only refer to heap locations with agreed on evaluations and the program fragment would be verified secure.

Definition 4.3.5 (assignField*)**.** The program fragments handled by the assignField* rule are assignments of program variables `v` to fields f.

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{r} \in F \qquad \Gamma \Rightarrow \Delta, \mathtt{v} \in F \\ \Gamma, \{\mu\}(\mathtt{r} \neq \mathtt{null}) \Rightarrow \Delta, \{\mu\}\{\mathtt{heap} := store(\mathtt{heap}, \mathtt{r}, f, \mathtt{v})\}[\![\ \theta\rho\ |\ F \cup \{\{\mu\}\langle \mathtt{r}, f\rangle\}\ |\ T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{r} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\,\mathtt{throw\ new\ NullPointerException();}\ \rho\ |\ F\ |\ T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\,\mathtt{r.f = v;}\ \rho\ |\ F\ |\ T\ ]\!]}$$

■

Proof. Again, we assume the initial states s and t to agree on F by id. Furthermore, from the first premise we can assume that `r` is included in F and, hence, we consider the two

cases separately where either both states assign `null` to `r` ($s(\texttt{r}) = \texttt{null} = t(\texttt{r})$) or none does ($s(\texttt{r}) \neq \texttt{null} \neq t(\texttt{r})$).

In the first case, we have $s(\texttt{r}) = \texttt{null} = t(\texttt{r})$, both states s and t are left unaltered by the Java semantics and the final states are determined by the program fragment that throws an exception instead of carrying out the assignment.

$$\theta\,\texttt{throw new NullPointerException();}\,\rho$$

For the altered program fragment we conclude the rule conclusion from the second premise.

$$[\![\ \theta\,\texttt{throw new NullPointerException();}\,\rho \mid F \mid T\ ]\!]$$

In the second proof case, we have $s(\texttt{r}) \neq \texttt{null} \neq t(\texttt{r})$ and also $s(\texttt{r}) = t(\texttt{r})$ from the agreement assumption. Each state is transformed into its successive state, s' and t' respectively, by the execution of the assignment statement (`r.f = v;`). The ensuing states are given by the following two equations.

$$\begin{aligned} s' &= s[\texttt{heap} \mapsto I(store)(s(\texttt{heap}), s(\texttt{r}), I(f), s(\texttt{v}))] \\ t' &= t[\texttt{heap} \mapsto I(store)(t(\texttt{heap}), t(\texttt{r}), I(f), t(\texttt{v}))] \end{aligned}$$

Since $\texttt{v} \in F$ we have $s(\texttt{v}) = t(\texttt{v})$ and we can conclude the equality of location evaluations.

$$val_{\mathcal{K},s,\beta}(select(\texttt{heap}, \texttt{r}, f)) = val_{\mathcal{K},t,\beta}(select(\texttt{heap}, \texttt{r}, f))$$

It follows that s' and t' are low equivalent with respect to $F \cup \langle s(\texttt{r}), f\rangle$. The final proof step infers the conclusion $[\![\ \theta\,\texttt{r.f = v;}\,\rho \mid F \mid T\ ]\!]$ again by application of the compositionality theorem to the propositions $[\![\ \texttt{r.f = v;} \mid F \mid F \cup \langle s(\texttt{r}), f\rangle\ ]\!]$ and $[\![\ \theta\rho \mid F \cup \langle s(\texttt{r}), f\rangle \mid T\ ]\!]$. The first proposition holds because of the low equivalence of s' and t', the second holds because of the third rule premise. □

Since array field accesses are modelled in JavaDL using a special *Field* domain value returned by the function $arr(\cdot)$ for an index expression, they could also be handled by the assignLocalField rule in JavaDL*. However, aside from a `NullPointerException`, an array field access can also trigger an `ArrayIndexOutOfBoundsException`. Moreover, there is a separate rule for handling array length expressions, which are immutable and thus not translated into field accesses but using a special function symbol `length`.

Definition 4.3.6 (assignArray*)**.** The program fragments handled by the assignArray* rule are assignments of program variables `v` to array elements `a[i]`.

$$\frac{\begin{array}{l}\Gamma \Rightarrow \Delta, \texttt{a} \in F \qquad \Gamma \Rightarrow \Delta, \texttt{i} \in F \qquad \Gamma \Rightarrow \Delta, \texttt{v} \in F \\ \Gamma, \{\mu\}(\texttt{a} = \texttt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\,\texttt{throw new NullPointerException();}\,\rho \mid F \mid T\ ]\!] \\ \Gamma, \{\mu\}(\texttt{a} \neq \texttt{null}), \{\mu\}(0 \leq \texttt{i} < \texttt{length(a)}) \\ \Rightarrow \Delta, \{\mu\}\{\texttt{heap} := store(\texttt{heap}, \texttt{a}, arr(\texttt{i}), \texttt{v})\}[\![\ \theta\rho \mid F \cup \{\{\mu\}\langle \texttt{a}, arr(\texttt{i})\rangle\} \mid T\ ]\!] \\ \Gamma, \{\mu\}(\texttt{a} \neq \texttt{null}), \{\mu\}(\texttt{i} < 0 \vee \texttt{i} \geq \texttt{length(a)}) \\ \Rightarrow \Delta, \{\mu\}[\![\ \theta\,\texttt{throw new ArrayIndexOutOfBoundsException();}\,\rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\,\texttt{a[i] = v;}\,\rho \mid F \mid T\ ]\!]}$$

■

Proof. We distinguish the cases where either no exception, a `NullPointerException` or an `ArrayIndexOutOfBoundsException` is thrown by the Java Virtual Machine. If one noninterference execution path throws an exception, both do because the first premise requires the array reference to be among the low locations and the definition of agreement preserves both `null` references and array lengths (Definition 4.1.2).

Here, we only prove the case in which no exception is thrown and the state is actually altered. The proofs of the other two cases are analogous to the proof of the first case of the field assignment rule.

The assignment statement `a[i] = v` now transforms the initial states s and t into s' and t' respectively, which are determined as follows.

$$s' = s[\texttt{heap} \mapsto I(store)(s(\texttt{heap}), s(\texttt{a}), I(arr)(s(\texttt{i})), s(\texttt{v}))]$$
$$t' = t[\texttt{heap} \mapsto I(store)(t(\texttt{heap}), t(\texttt{a}), I(arr)(t(\texttt{i})), t(\texttt{v}))]$$

The premises $\texttt{a} \in F$, $\texttt{i} \in F$ and $\texttt{v} \in F$ imply $s(\texttt{a}) = t(\texttt{a})$, $s(\texttt{i}) = t(\texttt{i})$ and $s(\texttt{v}) = t(\texttt{v})$. Moreover, since the heap is only updated at the location `a[i]` and F is at most extended by `a[i]`, it is sufficient to show $I(select)(s(\texttt{heap}), s(\texttt{a}), I(arr)(s(\texttt{i}))) = I(select)(t(\texttt{heap}), t(\texttt{a}), I(arr)(t(\texttt{i})))$ in order to conclude $flow^{**}(\texttt{a[i] = v;}, F, F \cup \langle s(\texttt{a}), arr(s(\texttt{i}))\rangle,)$. The former is already implied by the premises.

Lastly, with the help of the compositionality theorem, we conclude $[\![\, \theta\, \texttt{a[i] = v;}\, \rho \mid F \mid T \,]\!]$ from $[\![\, \texttt{a[i] = v;} \mid F \mid F \cup \langle s(\texttt{a}), arr(s(\texttt{i}))\rangle \,]\!]$ and $[\![\, \theta\rho \mid F \cup \langle s(\texttt{a}), arr(s(\texttt{i}))\rangle \mid T \,]\!]$. In that case, the presuppositions of the compositionality theorem are implied by $flow^{**}(\texttt{a[i] = v;}, F, F \cup \langle s(\texttt{a}), arr(s(\texttt{i}))\rangle, \emptyset)$ and the fourth rule premise respectively. □

Above, we presented rules that handle the different types of memory locations that can be assigned values. In the remainder, we discuss rules for object creation and control flow as well as rules for closing flow modality proofs.

Definition 4.3.7 (createObject*)**.** The program fragments handled by the createObject* rule are assignments of new `C` type object references to program variables `v`.

$$\frac{\begin{array}{l}\Gamma, o \neq \texttt{null}, exactInstance_C(o),\\ \{\mu\}(wellformed(\texttt{heap}) \rightarrow select_{Boolean}(\texttt{heap}, o, created) = False)\\ \Rightarrow \Delta, \{\mu\}\{\texttt{heap} := create(\texttt{heap}, o)\}\{\texttt{v} := o\}[\![\, \theta\rho \mid F \cup \{\texttt{v}\} \mid T \,]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\, \theta\, \texttt{v = C.alloc();}\, \rho \mid F \mid T \,]\!]}$$

■

Proof. Firstly, we prove $flow^{**}(\texttt{r = C.alloc();}, F, F \cup \{\texttt{r}\}, \{\texttt{r}\})$. Therefore, we assume the agreement of the initial states s and t on F by id. We state that the objects o_s and o_t referenced by `v` in the successive states s' and t' did not exist in s and t. Furthermore, s' and t' are already determined by the assignment of the heap references o_s and o_t respectively to `v`.

$$s' = s[\texttt{v} \mapsto o_s]$$
$$t' = t[\texttt{v} \mapsto o_t]$$

The objects $s'(\mathtt{v})$ and $t'(\mathtt{v})$ have the same type `C` so that we are able to define the partial isomorphism η from the object space referenced by $\{\mathtt{v}\}$ in s' to the object space referenced by $\{\mathtt{v}\}$ in t' through the mapping $\eta(s'(\mathtt{v})) = t'(\mathtt{v})$. If the fresh objects in both states share even the same reference value, then s' and t' agree on $F \cup \{\mathtt{v}\}$ by id. Indeed, id is a partial isomorphism in our case, since the new object could not be referenced from F in the initial states and we assumed id to be a partial isomorphism between the object spaces referenced by F in s and t.

Secondly and lastly, since we know that $flow^{**}(\mathtt{r = C.alloc();}, F, F \cup \{\mathtt{v}\}, \{\mathtt{v}\})$ holds in s we also know that $[\![\ \mathtt{r = C.alloc();}\ |\ F\ |\ F \cup \{\mathtt{v}\}\ ]\!]$ holds in s and together with the second premise $[\![\ \theta\rho\ |\ F \cup \{\mathtt{v}\}\ |\ T\ ]\!]$ we can conclude the rule conclusion $[\![\ \theta\, \mathtt{r = C.alloc();}\, \rho\ |\ F\ |\ T\ ]\!]$ by application of the compositionality theorem. □

The next two rules handle two essential programming language constructs for execution path branching and recursion. The JavaDL* rules for conditionals and loops are responsible for ensuring that no illicit implicit information flow occurs.

Our assignment rules are never able to establish that an assigned value does not leak information although secret memory locations occur syntactically. Similarly, our control flow rules are never able to establish that the execution of an alternative branch does not leak information although the conditional evaluates a secret memory location. However, if the condition is a low program variable, then the branch choice is determined solely by the attacker and it will never leak confidential information. In that case, a security analysis can verify noninterference for the possible branches independently by an application of the CONDITIONAL* rule.

Definition 4.3.8 (CONDITIONAL*)**.** The program fragments handled by the CONDITIONAL* rule are conditionals.

$$\frac{\Gamma \Rightarrow \Delta, \mathtt{c} \in F \qquad \Gamma, \{\mu\}(\mathtt{c} = True) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \pi_t\, \rho\ |\ F\ |\ T\ ]\!] \quad \Gamma, \{\mu\}(\mathtt{c} = False) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \pi_e\, \rho\ |\ F\ |\ T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{if\ (c)\ \{}\pi_t\mathtt{\}\ else\ \{}\pi_e\mathtt{\}}\, \rho\ |\ F\ |\ T\ ]\!]}$$

■

Proof. We fix two arbitrary initial states s and t that agree on F by id. Since the evaluation of the condition `c` is the same in both states, as implied by the first rule premise ($\mathtt{c} \in F$), both the execution of the conditional started in s and the one started in t choose the same branch. As a consequence, we can show the agreement of the final states s' and t' on T independently for each branch.

If $s(\mathtt{c}) = tt = t(\mathtt{c})$ holds and the `then` branch is chosen, then the final states s' and t' are both determined by the execution of the program fragment $\theta\, \pi_t\, \rho$. From $flow(\theta\, \pi_t\, \rho, F, T)$ we obtain a partial isomorphism η' by which s' and t' agree on T and that is compatible with id in s. The latter is the case since the initial states are not altered by the condition evaluation and the branch execution begins in s and t respectively. The satisfaction of the flow predicate is implied by $[\![\ \theta\, \pi_t\, \rho\ |\ F\ |\ T\ ]\!]$, which we infer from the first rule premise and the fact that $\mathtt{c} = True$ holds in s. Moreover, the first premise, by definition, also implies the observability condition $\{o \in Obj^{s'}(T) | I(select_{Boolean})(s(\mathtt{heap}), o, I(created)) = tt\} \subseteq Obj^{s}(F)$ and we finally conclude $[\![\ \theta\, \mathtt{if\ (c)\ \{}\pi_t\mathtt{\}\ else\ \{}\pi_e\mathtt{\}}\, \rho\ |\ F\ |\ T\ ]\!]$ for the case where the `then` branch of the conditional is executed.

In the second case we have to consider, $s(\mathtt{c}) = ff = t(\mathtt{c})$ holds and the final states s' and t' are both determined by the execution of the `else` branch ($\theta\, \pi_e\, \rho$). Once more, the third rule premise implies the conclusion for this case ($[\![\ \theta\, \mathtt{if\ (c)\ \{}\pi_t\mathtt{\}\ else\ \{}\pi_e\mathtt{\}}\, \rho\ |\ F\ |\ T\ ]\!]$).

In any case and for any state s, the final states agree on T by some partial isomorphism that extends id, by which the initial states are assumed to agree on F, and the observability condition is satisfied so that the CONDITIONAL* rule is sound. □

When we are concerned with implicit information flow, the loop semantics does not differ much from the conditional semantics. That is also the reason why loops are often iteratively transformed into a succession of conditionals in order to reason about them. However, the number of loop iterations is not constant in general and, therefore, each loop iteration must usually ensure the condition we want to assume for the possible states directly after the execution of the loop statement. In our analysis such a condition is the set of low memory locations. Since the same condition term is evaluated after each loop iteration, a noninterference proof for a loop statement in the JavaDL* calculus requires that the loop condition is not only a low program variable in the initial state but also after each loop iteration. If that is indeed the case and noninterference can be proved for each execution of the loop body, then the LOOP* rule can be applied.

Definition 4.3.9 (LOOP*)**.** The program fragments handled by the LOOP* rule are loops.

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \{\mu\}(I \subseteq F) \\ \Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{c} \in I) \qquad \Gamma \Rightarrow \Delta, \{\mu\}Inv \qquad \Gamma, \{\mu\}\mathcal{V}(\mathtt{c} = True \wedge Inv) \Rightarrow \Delta, \{\mu\}\mathcal{V}[\pi]Inv \\ \Gamma, \{\mu\}\mathcal{V}(\mathtt{c} = True \wedge Inv) \Rightarrow \Delta, \{\mu\}\mathcal{V}[\![\ \pi \mid I \mid I\ ]\!] \\ \Gamma, \{\mu\}\mathcal{V}(\mathtt{c} = False \wedge Inv) \Rightarrow \Delta, \{\mu\}\mathcal{V}[\![\ \theta\rho \mid I \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{while\ (c)\ \{}\pi\mathtt{\}}\, \rho \mid F \mid T\ ]\!]}$$

Here, $\mathcal{V}$ is an update that erases all information about the program variables and heap locations that may be altered by an execution of the loop body π. Furthermore, π must not contain `throw`, `break`, `continue` and `return` statements. ■

Proof. For a fixed state s we can assume

$$s \models I \subseteq F \quad (4.4)$$
$$s \models \mathtt{c} \in I \quad (4.5)$$
$$s \models Inv \quad (4.6)$$
$$s_{\mathcal{V}} \models \mathtt{c} = True \wedge Inv \longrightarrow [\pi]Inv \quad (4.7)$$
$$s_{\mathcal{V}} \models \mathtt{c} = True \wedge Inv \longrightarrow [\![\ \pi \mid I \mid I\ ]\!] \quad (4.8)$$
$$s_{\mathcal{V}} \models \mathtt{c} = False \wedge Inv \longrightarrow [\![\ \theta\rho \mid I \mid T\ ]\!] \quad (4.9)$$

since we can assume Γ in order to prove the rule conclusion. Here, $s_{\mathcal{V}}$ is obtained by assigning random and thus unknown well-typed values to variables and heap locations that may be modified by statements occurring in the loop body π, which may be a too weak assumption. We prove

$$s \models [\![\ \theta\, \mathtt{while\ (c)\ \{}\pi\mathtt{\}}\, \rho \mid F \mid T\ ]\!] \quad (4.10)$$

Therefore, we fix another arbitrary state t which is low equivalent to s with respect to F. As clarified in the assumptions of this thesis, we assume the existence of final states s' and t' so that there must be finite numbers of loop iterations n_s and n_t respectively.

First of all, we prove that the numbers of loop iterations are equal ($n_s = n_t$) as well as that the program states after the completion of all iterations (s_{n_s} and t_{n_t}) are low equivalent, by a simple induction over n_s. Therefore, consider the case $n_s = 0$ first, then $s(\mathtt{c}) = ff$ holds and because of $\mathtt{c} \in I \subseteq F$ (4.5 and 4.4) we also have $t(\mathtt{c}) = ff$. Moreover, $n_t = 0 = n_s$ holds as well as $s_0 = s$ and $t_0 = t$, which we assumed to be low equivalent. In order to prove the induction step, we assume for $n_s = i$ that $n_t = n_s$ holds as well as that s_{n_s} and t_{n_s} are low equivalent. If $n_s = i + 1$ holds, then $s_i(\mathtt{c}) = tt$ and $s_{i+1}(\mathtt{c}) = ff$ also hold as well as $t_i(\mathtt{c}) = tt$ because of the low equivalence of s_i and t_i. Moreover, because of the fifth rule premise (4.8) and the compositionality theorem, we know that s_{i+1} and t_{i+1} are again low equivalent and that $t_{i+1}(\mathtt{c}) = s_{i+1}(\mathtt{c}) = ff$ must hold. From the fact that the loop condition evaluates to false in both s_{i+1} and t_{i+1} for the first time, we conclude $n_s = n_t = i + 1$ in addition to the low equivalence of s_{i+1} and t_{i+1}. By induction we proved that the states after the loop completion are low equivalent with respect to I. However, we are only able to repeatedly apply rule premise 4.8 because we assume that the loop body π does not contain `throw`, `break`, `continue` and `return` statements.

Second of all, we prove s' and t' low equivalent with respect to T. From the previous result we obtain the low equivalence of s_{n_s} and t_{n_s} with respect to I and the equal number of loop iterations ($n_s = n_t$). Thus, the executions of the program fragment started in s and t reside in low equivalent states with respect to I directly after the execution of the last loop iteration. The last rule premise (4.9) ensures that when the rest of the program is executed beginning in low equivalent states with respect to I, the executions end in low equivalent states with respect to T. Again, an application of the compositionality theorem to $s \models [\![\ \pi^{n_s} \mid I \mid I\ ]\!]$ and $s_{n_s} \models [\![\ \theta\rho \mid I \mid T\ ]\!]$ gives us $s \models [\![\ \theta\, \mathtt{while\ (c)\ \{}\pi\mathtt{\}}\, \rho \mid I \mid T\ ]\!]$. The previous prove step used that loop body is executed n_s times and that the final states after the executions of π^{n_s} and `while (c) {`π`}` coincide. However, the from *RefSet* expression does not yet match F and the soundness proof is not completed until a last application of the compositionality theorem in order to conclude our proof goal (4.10) from $s \models [\![\ \mid F \mid I\ ]\!]$ and $s \models [\![\ \theta\, \mathtt{while\ (c)\ \{}\pi\mathtt{\}}\, \rho \mid I \mid T\ ]\!]$. The former is trivially true and the latter has just been proven.

When we applied the rule premises 4.8 and 4.9 during the proof above, we neither gave a proof for Inv nor argued why the program states satisfied the constraints dictated by $s_{\mathcal{V}}$. That Inv holds before and after each loop iteration is proved as for the usual JavaDL loop invariant rule using the rule premises 4.6 and 4.7. Lastly, the premises can be applied in each proof step because, by definition of $\mathcal{V}$, only the memory locations contained in $\mathcal{V}$ could have been altered since execution had left the initial program state s, and $s_{\mathcal{V}}$ does not make any assumptions about the assignments of those memory locations. □

For the handling of unstructured control flow JavaDL provides another special program variable, which is called `exc`. Again, in order to simplify the rule definitions the calculus requires that `throw` statements occur in the analysed program code only with program variable arguments. If the program variable that refers to the exception object is among the low memory locations, the JavaDL* calculus handles the unstructured control flow, just like the JavaDL calculus, by removing statements that are not executed from the surrounding program code and choosing the matching `catch` block. The proof idea is the same as for the CONDITIONAL* rule.

Definition 4.3.10 (THROW*)**.** The program fragments handled by the THROW* rule are `throw` statements.

In the schematic rule, we denote by ρ_1 the longest substring that does not contain a `catch` or `finally` block on the same code level. That means that ρ_2 possibly does not contain any further statements.

$$\frac{\Gamma \Rightarrow \Delta, \mathtt{r} \in F \qquad \Gamma \Rightarrow \Delta, \{\mu\}\{\mathtt{exc} := \mathtt{r}\}[\![\, \theta\, \rho_2 \mid F \cup \{\mathtt{exc}\} \mid T \,]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\, \theta\, \mathtt{throw\ r;}\ \rho_1\rho_2 \mid F \mid T \,]\!]}$$

■

Proof. Let s be an arbitrary program state the flow modality is evaluated in and t another arbitrary state that agrees with s on F by id.

We conclude $[\![\, \theta\, \mathtt{throw\ r;}\ \rho_1\rho_2 \mid F \mid T \,]\!]$ from $[\![\, \mathtt{throw\ r;} \mid F \mid F \cup \{\mathtt{exc}\} \,]\!]$ and $[\![\, \theta\, \rho_2 \mid F \cup \{\mathtt{exc}\} \mid T \,]\!]$ by the compositionality theorem. Executing $\theta\rho_2$ in the final program state after the execution of `throw r` indeed establishes the final state of an execution of $\theta\, \mathtt{throw\ r;}\ \rho_1\rho_2$ since in an exceptional state the Java Virtual Machine continues with the execution of a `catch` block or terminates the program execution.

While $[\![\, \theta\, \rho_2 \mid F \cup \{\mathtt{exc}\} \mid T \,]\!]$ is obtained directly from the second rule premise, in order to obtain $[\![\, \mathtt{throw\ r;} \mid F \mid F \cup \{\mathtt{exc}\} \,]\!]$ we show $flow^{**}(\mathtt{throw\ r;}, F, F \cup \{\mathtt{exc}\}, \emptyset)$ by arguing that `exc` is a low memory location. The latter is sufficient because the Java semantics only advance the program counter without altering the states s and t, in particular no objects are created. Eventually, the first rule premise proves immediately that `exc` is a low memory location with respect to the partial isomorphism id. □

Definition 4.3.11 (CATCH*)**.** The program fragments handled by the CATCH* rule are `try catch` statements.

In the schematic rule, we denote by ρ_1 all remaining `catch` blocks that belong to the `try catch` statement the rule is applied to. Consequently, $\rho_1^{\mathtt{try}}$ is equal to π_f if ρ_1 is empty and otherwise equals $\mathtt{try\ \{\ \}}\ \rho_1\ \mathtt{finally\ \{}\pi_f\mathtt{\}}$.

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{exc} \in F \\ \Gamma, \{\mu\} instanceof_E(\mathtt{exc}) \Rightarrow \Delta, \{\mu\}\{\mathtt{e} := \mathtt{exc}\}\{\mathtt{exc} := \mathtt{null}\}[\![\, \theta\, \pi_f\ \ \pi_c\, \rho_2 \mid F \mid T \,]\!] \\ \Gamma, \{\mu\}\neg instanceof_E(\mathtt{exc}) \Rightarrow \Delta, \{\mu\}[\![\, \theta\rho_1^{\mathtt{try}}\rho_2 \mid F \mid T \,]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\, \theta\, \mathtt{try\ \{\ \}\ catch\ (E\ e)\ \{}\pi_c\mathtt{\}}\ \rho_1\ \mathtt{finally\ \{}\pi_f\mathtt{\}}\rho_2 \mid F \mid T \,]\!]}$$

■

Proof. Let s be an arbitrary but fixed state. We show the rule conclusion for this state.

$$s \models [\![\, \theta\, \mathtt{try\ \{\ \}\ catch\ (E\ e)\ \{}\pi_c\mathtt{\}}\ \rho_1\ \mathtt{finally\ \{}\pi_f\mathtt{\}}\rho_2 \mid F \mid T \,]\!]$$

Therefore, let t be an arbitrary but fixed state that agrees with s on F by id.

The first rule premise ($\mathtt{exc} \in F$) ensures that either both s and t are in an exceptional state or none is. As a result, we distinguish those two cases in order to prove the agreement of the final states s' and t' after the execution of $\theta\, \mathtt{try\ \{\ \}\ catch\ (E\ e)\ \{}\pi_c\mathtt{\}}\ \rho_1\ \mathtt{finally\ \{}\pi_f\mathtt{\}}\rho_2$ on T.

If $s(\mathtt{exc}) = \mathtt{null} = t(\mathtt{exc})$, then no execution state is exceptional and the execution continues after the `try catch` statement with the `finally` block and ρ_2. However, this case is treated in a more general manner by the rule since JavaDL provides `exc` whose dynamic type is compared to the `catch` block condition types ($instanceof_E(\mathtt{exc})$). Those checks will never succeed, as long as `exc` is the `null` reference, and the states are not altered by those dynamic type checks. Therefore, the third rule premise ($[\![\, \theta\rho_1^{\mathtt{try}}\rho_2 \mid F \mid T \,]\!]$) implies that information flows between s and s' at most from F to T.

On the other hand, if the execution states are both exceptional ($s(\texttt{exc}) = t(\texttt{exc}) \neq \texttt{null}$), then either the same `catch` block will be executed in both states or no `catch` condition matches the type of `exc`. In the latter case, the rule conclusion is proved from the third rule premise in the same way as it was proved in the first case where there is no exception, with the only exception that the execution state is still exceptional in this case. If the `catch` condition of the `catch` block this rule is applied to matches the dynamic type of `exc`, then $instanceof_E(\texttt{exc})$ holds in both states and the second rule premise implies the noninterference criterion for the given program fragment, since the final states are determined by the execution of $\theta\, \pi_f\ \ \pi_c\, \rho_2$ and the execution states are no more exceptional ($\texttt{exc} = \texttt{null}$). However, if there is a matching `catch` condition but $\neg instanceof_E(\texttt{exc})$ holds, then the final execution states will be the same if we remove the `catch` block `catch (E e) {`π_c`}` from the `try catch` statement and the third rule premise implies the noninterference criterion for θ `try { } catch (E e) {`π_c`}` ρ_1 `finally {`π_f`}` ρ_2. □

The next rule is applied to flow modalities where the program fragment is the empty program. An application of the emptyModality* rule rewrites the flow modality to a subset predicate of the set of low memory locations determined by an attacker and the set of low memory locations observed by the attacker. Then, a proof that the subset predicate is satisfied implies that the attacker only learns information that it already knew. Usually, the contents of the first set is established through symbolic execution of a larger program and the reduction of an instance of the noninterference security model to a subset relation ends with an application of the emptyModality* rule. The JavaDL* rules for inferring the validity of the subset predicate formula are listed in the appendix of this thesis.

Definition 4.3.12 (emptyModality*)**.** The emptyModality* rule erases a flow modality from the sequent.

$$\frac{\Gamma \Rightarrow \Delta, \{\mu\}(T \subseteq F)}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \mid F \mid T\ ]\!]}$$

■

Proof. We show $flow^{**}(, F, T, \emptyset)$, which implies $[\![\ \mid F \mid T\ ]\!]$ [SSB+13, Lemma 8]. Therefore, we assume the arbitrary initial states s and t to agree on F by id. Any two states, in particular s and t, agree on the empty set (as given in the flow predicate as the set of new object instantiations) by id so that we must show their agreement on T by id without further assumptions.

Consider any program variable $\texttt{v} \in T$. Since the program fragment is empty, each start and end state of both executions $s \rightsquigarrow s'$ and $t \rightsquigarrow t'$ are the same. From the rule premise $T \subseteq F$ we obtain $\texttt{v} \in F$ and, hence, the succession of equalities $s'(\texttt{v}) = s(\texttt{v}) = t(\texttt{v}) = t'(\texttt{v})$.

The same applies to any heap location $\langle o, f\rangle \in T \subseteq F$. Observe that the heaps in s and t are not necessarily equal but they assign the same value (agreement by id) to the location $\langle o, f\rangle$ so that $select(s'(\texttt{heap}), o, f) = select(s(\texttt{heap}), o, f) = select(t(\texttt{heap}), o, f) = select(t'(\texttt{heap}), o, f)$.

Conclusively, by restricting id to the object space referenced by T in s' we obtain the intended result that id is a partial isomorphism with respect to T in the final program state s'. □

We now present the last rule in this section. The Fallback rule translates a flow modality formula into a JavaDL formula by self-composition, which is the standard theorem proving

approach to information flow analysis []. The rule is essential to the practicability of the JavaDL* calculus since it reduces a flow modality formula to a formulation of its precise semantics and, thus, compensates for the inapplicability of the syntax-oriented rules when functional reasoning is necessary. In such cases an application of the FALLBACK rule continues with a proof without losing closed proof branches for other execution paths. The latter would be the case if the calculus, instead, always had to apply self-composition to the complete program code and start from the beginning.

Definition 4.3.13 (FALLBACK)**.** The FALLBACK rule translates between a flow modality and a JavaDL formula $\Phi_{\pi,F,T}$ (Theorem 1).

$$\frac{\Gamma \Rightarrow \Delta, \{\mu\}\Phi_{\pi,F,T}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \pi \mid F \mid T\ ]\!]}$$

■

Proof. We prove the rule FALLBACK sound. Therefore, we assume $s \models \Phi_{\pi,F,T}$ for an arbitrary but fixed state s and show $s \models [\![\ \pi \mid F \mid T\ ]\!]$. From Theorem 1 we know that both propositions are even equivalent. This concludes the proof. □

4.4. Rewrite Rules

The problem of determining which expressions have the same evaluation, or at least equal values in terms of a state equivalence relation, after the termination of both program executions in the noninterference security model is reduced to the problem of determining which program variables and heap object fields are assigned equal values in the respective final program states by the JavaDL* information flow analysis.

Recall that the calculus rules for the flow modality compare states with respect to a list of program variables and heap object fields and that the rules for conditionals and loops expect a program variable as condition and that there is only one rule defined in the JavaDL* calculus that handles expressions that include more evaluated syntax than a single variable occurrence or field access. Therefore, the purpose of this section is to shortly clarify which program transformations may be necessary before the JavaDL* rules can be applied to arbitrary Java Card programs.

In the following definitions, we will employ the syntax $\pi_1 \rightsquigarrow \pi_2$ to denote the existence of a transformation rule that rewrites the program fragment π_1 anywhere in a JavaDL* formula or term by replacing it with the program fragment π_2.

Method and constructor calls Without loss of generality, it is assumed that method calls appear only in statements of the form `v = o.m(`$p_1, \ldots, p_n$`);`. If the analysed program fragment consists solely of a method call, then the return value will be assigned to the special program variable `res`. If the method being called is declared without a return type, then there will be no `return` statements after inlining the `void` method and this rewrite rule can thus be applied even if the method call is not the right-hand side of an assignment.

The method `m` is inlined by declaring program variables for the method's formal parameters and assigning them the actual parameters $p_1, \ldots, p_n$ and replacing the method call with the method frame statement `method-frame(result=v,this=o)`.

If there is more than one implementation of `m` defined in the class type hierarchy, the method call is firstly replaced with a conditional restricting the dynamic type of `o` and inlining each method implementation on the conditional branch that belongs to the respective dynamic class type. Furthermore, there must always be a case distinction whether the object reference `o` is the `null` reference or not.

Similar transformation steps are applied to constructor calls in order to inline them as well. However, inlined constructor code is additionally preceded by a `C.alloc();` call statement, where `C` is the dynamic type of the created object.

For loops `for` loops are trivially expressed as `while` loops. There is a JavaDL* rule that is applied to the latter syntax in order to handle loops in proofs.

`for (`π_i`;`α`;`π_s`) {`π`}` $\rightsquigarrow$ π_i`; while (`α`) {`π`;`π_s`}`

Switch statements Without loss of generality, it is assumed that each `case` blocks ends with a `break` statement.

`switch (`α_s`) { case `α_c`:` π_c`; break;` ρ `}`
$\rightsquigarrow$ `if (`α_s`==`α_c`) {`π_c`} else { switch (`α_s`) {`ρ`} }`

`switch (`α_s`) { default:` π_d `}` $\rightsquigarrow$ π_d

Missing finally block For every `try catch` statement that does not have a `finally` block an empty one is inserted.

Single or multiple branch conditionals Conditionals that open no alternative branch or more than one branch are normalised by the addition of an empty `else` branch or through nested conditionals.

`if (`α`) {`π_t`} ⇝ if (`α`) {`π_t`} else { }`

`if (`α_1`) {`π_t`} else if (`α_2`) {`π_{ei}`} `ρ ⇝ `if (`α_1`) {`π_t`} else { if (`α_2`) {`π_{ei}`} `ρ }

Complex conditions Expressions in conditionals and loops that do not solely consist of a program variable are assigned to a fresh program variable `c` and replaced in the statement with an occurrence of `c`.

`if (`α`) {`π_t`} else {`π_e`} ⇝ boolean c = `α`; if (c) {`π_t`} else {`π_e`}`

`while (`α`) {`π`} ⇝ boolean c = `α`; while (c) { `π`; c = `α`; }`

Complex field assignments Like complex conditions, assignments of complex expressions to fields are replaced by assignments of the complex expressions to fresh variables and assigning those to the field afterwards.

`r.f = `α`; ⇝ `$\mathcal{T}$` v = `α`; r.f = v;`

(`$\mathcal{T}$` is the static type of the expression `$\alpha$`)

Method calls and field accesses in expressions The rewrite rules described in the following remove method calls and field accesses from complex expressions completely since their evaluation opens alternative execution paths.

`v = r.f `*op* `$\alpha$`; ⇝ `$\mathcal{T}$` t = r.f; v = t `*op* `α`;`

`v = o.m(`$p_1, \ldots, p_n$`) `*op* `$\alpha$`; ⇝ `$\mathcal{T}$` t = o.m(`$p_1, \ldots, p_n$`); v = t `*op* `α`;`

(*op* stands for an operator and may be a method call or field access itself as well; `$\mathcal{T}$` is the static field type or method return type)

Multiple variable or field declarations or definitions Statements that declare more than one identifier or declare and define at the same time are split into separate statements so that a single declaration and definition calculus rule suffices.

`$\mathcal{T}$ `v_1` = `α_1`, ..., `v_n` = `α_n`; ⇝ `$\mathcal{T}$ `$v_1$` = `$\alpha_1$`; ...; `$\mathcal{T}$ `v_n` = `α_n`;`

`$\mathcal{T}$ v = `α` ⇝ `$\mathcal{T}$ v; v = `$\alpha$`;`

Ternary operator The ternary operator is translated into a conditional in a straightforward manner if its result is assigned to a program variable in the rewritten program fragment. Otherwise, expression simplifying rewrite rules must be applied first.

`v = c ? a : b; ⇝ if (c) { v = a; } else { v = b; }`

Compute and assign There are operators and assignments that implicitly assign a variable or carry out additional computation on the assigned expression.

`v = u++; ⇝ v = u; u = u + 1;`

`v = ++u; ⇝ u = u + 1; v = u;`

The same applies to the operator `-`.

`v += u; ⇝ v = v + u;`

The same applies to other operators like `-`, `%`, `/`, `*`.

5. Information Flow Analysis

This chapter describes information flow analysis of Java Card programs using the JavaDL* calculus defined in this thesis. We begin with a general introduction and end with the verification of concrete examples. The introduction refers to the common noninterference security model that specifies a single set of public memory locations and the examples illustrate that the JavaDL* calculus generalises that security model and allows the specification of distinct sets in the initial and final program states.

5.1. Flow Modality Verification

This section explains how noninterference information flow analysis is carried out using the JavaDL* calculus. An analysis begins with the security model parameter specifications and ends in a verification proof. Besides, this section depicts which attackers will never succeed in case of a positive verification result.

The analysed program source code must be given by the complete class hierarchy and its respective class definitions as well as the entry point by an initial method call. The initial method call is usually a static invocation of a method that has the well-known signature `void main(String[] args)` but will be a different one if only parts of a software system or a critical section are analysed.

Firstly, we revisit the assumed attacker model in order to clarify how analysis results can be interpreted. Secondly, we depict which parameters of the security model must be specified and what a developer specification for the analysed source code may look like. Lastly, we give a translation of the latter into a JavaDL* proof obligation.

The attacker model we assume in this thesis allows an attacker to modify and evaluate only a fixed set of program variables and heap object fields before the initial method invocation and after its termination. Because of the latter, our analysis is restricted to terminating programs and, for instance, cannot analyse programs that loop infinitely. From now on, the fixed set of memory locations accessible by an attacker is referred to as the set of "low" memory locations in contrast to the "high" memory locations, among which are all program variables and heap object fields that are not contained in the low set. The intended intuition behind these commonly used labels is that public memory locations have a low security classification and secret memory locations have a high security classification.

The analysed source code must be annotated with a specification defining the set of low memory locations users of the software system and attackers alike can influence in the ways defined by the attacker model. When a complete software system is verified and not only parts of it, the specification may include the stream objects that abstract from the input and output devices, which are written to and read from during user interaction. More importantly, the special program variable `exc` must be a low memory location if exceptional program states are exposed to the user by the Java Virtual Machine. If critical sections are analysed, then the low set usually contains fields of existing heap objects and a special program variable for the return value of the initial method call. Even more important than in the first verification scenario is the declaration of `exc` as a low memory location in these scenarios since the absence of reasonable return values is commonly signalled through the exception mechanism.

In order to verify that a program is secure concerning its information flow, the goal is to prove that an attacker cannot learn anything about the high memory locations by executing the program with freely chosen values for the low memory locations and evaluating the low memory locations in the final state. The noninterference security model formalises that goal by demanding that two independent executions of a secure program with randomly initialised memory that cannot be distinguished by examination of the low memory locations in the initial state, are indistinguishable after their termination. The satisfaction of the noninterference criterion by a program and its specification implies that an attacker cannot alter inputs in a way such that the outputs reveal insights about the secret memory locations. The reason is that the secret memory locations may be changed arbitrarily and the attacker still observes the same public results, in particular we may alter exactly the secret information an allegedly successful attacker claims to have acquired and the attacker must still assume them to be valid afterwards if they were not acquired outside the attacker model, which is an obviously false conclusion.

It remains to clarify what it means for two executions to be indistinguishable. A program execution is a succession of state transitions and a state is given by the values assigned to the program variables, which are either of primitive or reference type, and the heap object space. States are compared by comparing the respective state value of each low program variable or heap object field. Therefore, a primitive type low memory location in two indistinguishable states evaluates to the same value in both states. A reference type low memory location may point to different heap locations in both states without the states becoming distinguishable, as long as the referenced objects are indistinguishable. The latter is the case if what applies to the program variables and was just described also applies to the type's fields. An exception of this rule occurs, and two executions become distinguishable, if the initial reference value of a reference type program variable or field is only changed during one execution and left unaltered by the second execution.

In order to deductively verify a program and its specification, the low set definition is translated into a *RefSet* expression R (Definition 4.2.1) and the proof goal into the flow modality $[\![$ `r.m();` $\mid R \mid R$ $]\!]$, where `r.m();` is the initial method call. That any two executions are indistinguishable is encoded in the agreement of states (Definition 4.1.2) and the comparability of references before and after an execution (Equation 4.1).

```
1  class C {
2    /*@
3      @ \from input;
4      @ \to \result;
5      @*/
6    int m(int input) {
7      input = secret;
8      input = 0;
9      return input;
10   }
11 }
```

Listing 5.1: Example specification of a secure program that temporarily assigns a secret to a low program variable and clears it afterwards

5.2. Examples

In this section, we apply the information flow analysis described in the previous section (Section 5.1) to some rather pathological examples of Java Card programs. Each example given in this section defines a new type `class C` that contains a method `m`. The method definitions are annotated with a to be verified information flow contract that specifies the list of low memory locations before and after a method invocation. The contract syntax borrows the JML-style comments [LBR06] and uses the keywords `\from` and `\to` for defining the *RefSet* expressions F and T respectively. The initial proof obligation is to prove $[\![\texttt{c.m();} \mid F \mid T]\!]$ in arbitrary states that assign `c` an instance of the example-specific type `C` with its differing signatures and implementations of the method `m`. A proof of the flow modality formula implies a correct method implementation of the security policy that information flows at most from F to T. Each example is accompanied by a proof outline or, alternatively, an argument for why no proof should exist at all. In the proof outlines, we concentrate on the proof steps that apply rules to the flow modality or the *RefSet* expressions. In particular, we will not carry out the preceding program transformations, that may be necessary, explicitly. As mentioned earlier, among those program transformations are method binding and inlining.

The first example is shown in Listing 5.1 and we want to verify that the method `m` allows information to flow at most from the method argument `input` to the return value, although the secret variable `secret` is assigned to `input` in line 7, which is eventually returned by the method body. We could have omitted `input` from the low declarations but it seems more intuitive to declare arguments as user input. In particular, interference between `input` and any other memory location is impossible since it is never read.

For the proof of example 5.1 shown in Figure 5.1, the program variable identifiers are shortened and are only referred to by their first character. The from *RefSet* expression $\langle\{\texttt{i}\}, \emptyset\rangle$ consists of the method argument `input` and the to *RefSet* expression $\langle\{\texttt{r}\}, \emptyset\rangle$ consists of the special program variable `result` that was inserted for the return value during method inlining. The proof is carried out straightforwardly by symbolically executing the program fragment and determining that in the final state any two executions initially agreeing on the actual method parameter agree on both the value of the formal parameter variable `input` and the return value. Since the return value is included in this judgement, the method specification is fulfilled.

The second example is shown in Listing 5.2 and the return value of the method `m` in `class C` must not interfere with any other memory location. Essentially, this means that the return value must be constant in any implementation that adheres to this security policy. The particularity in this example, however, is that line 6 declares a local variable we did

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Rightarrow True}\,\text{trueRight}}{\Rightarrow \{\texttt{i} := \texttt{s}\}\{\texttt{i} := 0\}\{\texttt{r} := \texttt{i}\}True}\,\text{applyOnRigid}}{\Rightarrow \{\texttt{i} := \texttt{s}\}\{\texttt{i} := 0\}\{\texttt{r} := \texttt{i}\}(\langle\emptyset,\emptyset\rangle \subseteq \langle\{\texttt{i},\texttt{r}\},\emptyset\rangle)}\,\text{subset}_1^*,\ \text{subset}_3^*}{\Rightarrow \{\texttt{i} := \texttt{s}\}\{\texttt{i} := 0\}\{\texttt{r} := \texttt{i}\}(\langle\{\texttt{r}\},\emptyset\rangle \subseteq \langle\{\texttt{i},\texttt{r}\},\emptyset\rangle)}\,\text{subset}_1^*,\ \text{subset}_2^*,\ \text{inUnion}^*,\ \text{inSingleton}^*}{\Rightarrow \{\texttt{i} := \texttt{s}\}\{\texttt{i} := 0\}\{\texttt{r} := \texttt{i}\}[\![\ \mid \langle\{\texttt{i},\texttt{r}\},\emptyset\rangle \mid T\ ]\!]}\,\text{emptyModality}^*}{\Rightarrow \{\texttt{i} := \texttt{s}\}\{\texttt{i} := 0\}[\![\ \texttt{r = i;} \mid \langle\{\texttt{i}\},\emptyset\rangle \mid T\ ]\!]}\,\text{assignLocal}^*}{\Rightarrow \{\texttt{i} := \texttt{s}\}[\![\ \texttt{i = 0; r = i;} \mid \langle\emptyset,\emptyset\rangle \mid T\ ]\!]}\,\text{assignLocal}^*}{\Rightarrow [\![\ \texttt{i = s; i = 0; r = i;} \mid F \mid T\ ]\!]}\,\text{assignLocal}_3^*$$

Figure 5.1.: A JavaDL* proof of the example specification shown in Listing 5.1

```
1 class C {
2   /*@
3     @ \to \result;
4     @*/
5   void m() {
6     int tmp = secret;
7     return 0;
8   }
9 }
```

Listing 5.2: Example specification of a secure program that assigns a secret to a local program variable that is not included in the specification and is merely temporary

not include in the low specification and we would not include in a specification of secret memory locations, either. This unspecified local variable is assigned an intentionally secret memory location and we want to illustrate with this example that such assignments treated naturally by the JavaDL* calculus.

A proof of the program specification 5.2 is given in Figure 5.2. The proof tree shows the steps after the method implementation is inlined. The from *RefSet* expression F turns out to be empty for this specification and the last proof steps establish that the special program variable `result` (again abbreviated by its first character) is included in the set of memory locations the final states of any two method body executions agree on (in this case, $\langle\{\texttt{result}\},\emptyset\rangle$). In fact, this is the case and the proof can be closed since the to set T consists of exactly `result`.

The third example is shown in Listing 5.3. The implementation of `m` in this example returns the actual method argument if the field `secret` evaluates to `false` and a new instance of

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Rightarrow True}\,\text{trueRight}}{\Rightarrow \{\texttt{t} := \texttt{s}\}\{\texttt{r} := 0\}True}\,\text{applyOnRigid}}{\Rightarrow \{\texttt{t} := \texttt{s}\}\{\texttt{r} := 0\}(\langle\emptyset,\emptyset\rangle \subseteq \langle\{\texttt{r}\},\emptyset\rangle)}\,\text{subset}_1^*,\ \text{subset}_3^*}{\Rightarrow \{\texttt{t} := \texttt{s}\}\{\texttt{r} := 0\}(\langle\{\texttt{r}\},\emptyset\rangle \subseteq \langle\{\texttt{r}\},\emptyset\rangle)}\,\text{subset}_1^*,\ \text{subset}_2^*,\ \text{inSingleton}^*}{\Rightarrow \{\texttt{t} := \texttt{s}\}\{\texttt{r} := 0\}[\![\ \mid \langle\{\texttt{r}\},\emptyset\rangle \mid T\ ]\!]}\,\text{emptyModality}^*}{\Rightarrow \{\texttt{t} := \texttt{s}\}[\![\ \texttt{r = 0;} \mid \emptyset \mid T\ ]\!]}\,\text{assignLocal}^*}{\Rightarrow [\![\ \texttt{int t; t = s; r = 0;} \mid F \mid T\ ]\!]}\,\text{assignLocal}_3^*$$

Figure 5.2.: A JavaDL* proof of the example specification shown in Listing 5.2

```
1  class C {
2    /*@
3      @ \from input;
4      @ \to \result;
5      @*/
6    public C m(C input) {
7      if (secret) {
8        input = new C();
9      }
10     return input;
11   }
12 }
```

Listing 5.3: Example specification of an insecure program that updates a reference type variable on only one possible execution path and the execution path is chosen depending on a secret

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Rightarrow False}}{\Rightarrow \{\mathtt{i} := v\}\{\mathtt{r} := \mathtt{i}\}False}\,\text{APPLYONRIGID}}{\Rightarrow \{\mathtt{i} := v\}\{\mathtt{r} := \mathtt{i}\}(\langle\{\mathtt{r}\},\emptyset\rangle \subseteq \langle\emptyset,\emptyset\rangle)}\,\text{SUBSET}_1^*,\ \text{SUBSET}_2^*,\ \text{INEMPTY}^*}{\Rightarrow \{\mathtt{i} := v\}\{\mathtt{r} := \mathtt{i}\}[\![\ \mid \langle\emptyset,\emptyset\rangle \mid T\ ]\!]}\,\text{EMPTYMODALITY}^*}{\Rightarrow \{\mathtt{i} := v\}[\![\ \mathtt{r = i;} \mid \langle\emptyset,\emptyset\rangle \mid T\ ]\!]}\,\text{ASSIGNLOCAL}_3^*}{\Rightarrow [\![\ \mathtt{if\ (s)\ \{\ i = new\ C();\ \}\ else\ \{\ \}\ r = i;} \mid F \mid T\ ]\!]}\,\text{CONDITIONAL}_2^*$$

Figure 5.3.: Failed proof attempt for the example specification shown in Listing 5.3

class `C` otherwise. The natural specification says that users initialise the method argument and evaluate the return value. Since the method argument and the return value are identical if and only if the secret field is `false` this implementation of `m` does not fulfil its noninterference contract.

The attempt to prove the specification given in Listing 5.3 by the rule applications listed in Figure 5.3 fails as expected. Because the field `secret` forming the conditional is not a low memory location, the second conditional rule, which removes all memory locations that may be modified on either branch from the low set, must be applied. Since there is an execution path that alters the method parameter `input`, the low set is empty after the conditional and `input` gets assigned an unknown value through the $\mathcal{V}$ update in the schematic rule. Thus, the return value does not become a low memory location when the execution of the method body is finished and the proof tree cannot be closed.

The fourth example, which is shown in Listing 5.4, is an implementation of the method `m` that does not fulfil its specification. The illegal information flow occurs in line 8. After that line of code is executed, the program variable `tmp` points to the object o for which the field f is declared accessible by users and attackers alike. As a result, an attacker learns the evaluation of `secret` by evaluating f and a sound information flow analysis of object-oriented programs must cope with explicit information flow through aliasing. If `input` was not declared a low memory location, then there would be also implicit information flow because `input` could be the `null` reference and line 8 would throw a `NullPointerException`.

Figure 5.4 shows a possible proof attempt for example 5.4 that symbolically executes the transformed method body and finds that $\langle o, f\rangle$ is not included in the set of memory locations two arbitrary executions will always agree on in their final states. In the method

```
1  class C {
2    /*@
3      @ \from input;
4      @ \to \old(input).f;
5      @*/
6    public void m(C input) {
7      C tmp = input;
8      tmp.f = secret;
9    }
10 }
```

Listing 5.4: Example specification of an insecure program that assigns a secret to a public field by accessing it on a temporary alias

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Rightarrow False}}{\Rightarrow \{\mathtt{i} := o\}\{\mathtt{t} := \mathtt{i}\}\{\mathtt{heap} := store(\mathtt{heap},\mathtt{t},f,\mathtt{s})\}False}\,\text{APPLYONRIGID}}{\Rightarrow \{\mathtt{i} := o\}\{\mathtt{t} := \mathtt{i}\}\{\mathtt{heap} := store(\mathtt{heap},\mathtt{t},f,\mathtt{s})\}(\langle o,f\rangle \in \emptyset)}\,\text{INEMPTY}^*}{\Rightarrow \{\mathtt{i} := o\}\{\mathtt{t} := \mathtt{i}\}\{\mathtt{heap} := store(\mathtt{heap},\mathtt{t},f,\mathtt{s})\}(\langle\emptyset,\{\langle o,f\rangle\}\rangle \subseteq \langle\{\mathtt{i},\mathtt{t}\},\emptyset\rangle)}\,\text{SUBSET}_1^*,\ \text{SUBSET}_2^*}{\Rightarrow \{\mathtt{i} := o\}\{\mathtt{t} := \mathtt{i}\}\{\mathtt{heap} := store(\mathtt{heap},\mathtt{t},f,\mathtt{s})\}[\![\ \mid \langle\{\mathtt{i},\mathtt{t}\},\emptyset\rangle \mid T\ ]\!]}\,\text{EMPTYMODALITY}^*}{\Rightarrow \{\mathtt{i} := o\}\{\mathtt{t} := \mathtt{i}\}[\![\ \mathtt{t.f = s;} \mid \langle\{\mathtt{i},\mathtt{t}\},\emptyset\rangle \mid T\ ]\!]}\,\text{ASSIGNFIELD}_2^*}{\Rightarrow \{\mathtt{i} := o\}[\![\ \mathtt{C\ t;\ t = i;\ t.f = s;} \mid F \mid T\ ]\!]}\,\text{ASSIGNLOCAL}^*$$

Figure 5.4.: Failed proof attempt for the example specification shown in Listing 5.4

specification, the term `\old(input).f` is used to refer to the field f of the object the method argument `input` points to when the method `m` is called. For the translation of the specification into a proof obligation, we introduce a fixed but unknown logic variable o that is assigned to the program variable `input` by an initial update. The from *RefSet* expression F consists of `input` solely, which allows us to apply the rule ASSIGNFIELD* but is also consistent with our intuition that the method argument `input` is a low memory location. The to *RefSet* expression T consists of the location $\langle o, f\rangle$.

The last example in this section specifies a secure program fragment that cannot be verified by the JavaDL* calculus. The fifth example is shown in Listing 5.5 and similar to the third example (5.3) the evaluation of the secret field `secret` decides whether the method parameter `input` gets assigned a fresh reference to an instance of class `C`. However, this time `input` is altered both if the secret field evaluates to `true` (line 8) and if the secret field evaluates to `false` (line 12). Thus, both executions observed in the noninterference security model always overwrite the actual method argument with fresh object references. It is true that those almost always differ but an attacker cannot determine which evaluation of `secret` generated which reference value. While a self-composition analysis will verify the secure behaviour of `m` in this example, the JavaDL* calculus is not able to exploit the fact that both conditionals execute the same statements and that each conditional is the negation of the other. Aside from the FALLBACK rule, CONDITIONAL$_2^*$ is the only applicable JavaDL* rule with satisfied premises. An application of the second conditional rule to the example erases all low memory locations and makes the value of `input` anonymous. As a result, the non-empty set of low memory locations in the final state T must be proved a subset of the empty set, which is not possible. The failed proof attempt is conducted in Figure 5.5.

```
1  class C {
2    /*@
3      @ \from input;
4      @ \to \result;
5      @*/
6    public C m(C input) {
7      if (secret) {
8        input = new C();
9      }
10
11     if (!secret) {
12       input = new C();
13     }
14
15     return input;
16   }
17 }
```

Listing 5.5: Example specification of a secure program that assigns fresh reference values to a low program variable on two alternative branches. The branches are selected depending on boolean secrets, which, however, are mutual negations

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Rightarrow False}}{\Rightarrow \{\mathtt{i} := v\}\{\mathtt{i} := w\}\{\mathtt{r} := \mathtt{i}\}False}\,\textsc{applyOnRigid}}{\Rightarrow \{\mathtt{i} := v\}\{\mathtt{i} := w\}\{\mathtt{r} := \mathtt{i}\}(\langle\{\mathtt{i},\mathtt{r}\},\emptyset\rangle \subseteq \langle\emptyset,\emptyset\rangle)}\,\textsc{subset}_1^*,\ \textsc{subset}_2^*,\ \textsc{inEmpty}^*}{\Rightarrow \{\mathtt{i} := v\}\{\mathtt{i} := w\}\{\mathtt{r} := \mathtt{i}\}[\![\ \mathtt{r = i;}\ |\ \langle\emptyset,\emptyset\rangle\ |\ T\ ]\!]}\,\textsc{emptyModality}^*}{\Rightarrow \{\mathtt{i} := v\}\{\mathtt{i} := w\}[\![\ \mathtt{r = i;}\ |\ \langle\emptyset,\emptyset\rangle\ |\ T\ ]\!]}\,\textsc{assignLocal}_3^*}{\Rightarrow \{\mathtt{i} := v\}[\![\ \mathtt{if\ (!s)\ \{\ i = new\ C();\ \}\ r = i;}\ |\ \langle\emptyset,\emptyset\rangle\ |\ T\ ]\!]}\,\textsc{conditional}_2^*}{\Rightarrow [\![\ \mathtt{if\ (s)\ \{\ i = new\ C();\ \}\ if\ (!s)\ \{\ i = new\ C();\ \}\ r = i;}\ |\ F\ |\ T\ ]\!]}\,\textsc{conditional}_2^*$$

Figure 5.5.: Failed proof attempt for the example specification in Listing 5.5

6. Conclusions

We extended the JavaDL calculus by inference rules that defer the application of self-composition when proving noninterference between two sets of program variables and locations in Java Card programs. Moreover, we provided semi-formal proofs for the soundness of the defined rules. In the course of this work, a new modality was added to the JavaDL syntax. The semantics of the introduced flow modality is only a sufficient condition for the noninterference property of the analysed program fragment. However, this is due to an exploitation of a compositionality result which has recently been proven by Scheben and Schmitt et al. for their information flow predicate. The compositionality of the flow modality enabled the symbolic execution of program fragments without resorting to self-composition until implicit information flow might occur. As a result, the complexity of information flow proofs in JavaDL could be reduced for many obviously secure programs.

7. Outlook

The KeY system [BHS07] includes an interactive prover for JavaDL. JavaDL formulae can directly be inserted or generated from JML annotated Java source files. As mentioned before, JML noninterference contracts as well as their translation into JavaDL formulae were defined by Scheben and Schmitt [SS11] and implemented in a development version of the KeY system. The KeY prover includes an implementation of the JavaDL sequent calculus and supports easy addition of further rules to the rule database via a domain specific language. The rules defined in this thesis are thought of as an extension to the JavaDL sequent calculus and may be implemented in the KeY system in the future. Before that, an efficient handling of *RefSet* expressions must be found. The operations on *RefSet* expressions include the addition and removal of elements in the assignment rules as well as the enumeration of the elements in the fallback rule.

The JavaDL* calculus still does not contain a loop invariant rule that allows the loop body to contain statements that lead to unstructured control flow. In particular, the forbidden statements are `throw`, `break`, `continue` and `return`. If there is one of those statements executable from the loop body, then the JavaDL* rule is no more correct since control flow will continue after the loop. An alternative loop invariant must be added to the JavaDL* calculus since this is a considerable restriction on the programming language features.

An important feature when program properties are verified on a meta-language level is the support of modular specification and proof reuse. Modular specification of noninterference properties is possible since the extension of JML by Scheben and Schmitt [SS11]. However, the reuse of proofs for specific noninterference specifications is not yet possible, neither using the standard JavaDL rules nor using the flow modality rules defined in this thesis.

The *RefSet* expressions, which are used in this thesis to denote the attacker accessible information, only support the specification of variable identifiers and heap locations. It may be desirable to specify any JavaDL term since this would simplify the *RefSet* type definition and, most of all, give rise to more fine-grained security policies. Moreover, the support of specifying the public information as a list of arbitrary JavaDL terms seems to be closely related to the declassification of meta-information about memory locations, which stands in contrast to only disclosing the assigned values. A recurrent program example that needs declassification for a realistic specification is that of a password checker. Classifying the password file object confidential is too restrictive since it is obviously necessary for a correct implementation to forward error messages like `FileNotFoundException` to the user,

which would not fulfil a specification saying that the file object does not interfere with the program output. A declassification of the information whether an error occurred or not would enable a proof of the noninterference contract under the assumption that an attacker knows the evaluation of a formula describing whether an error occurred or not. Declassification was not taken into consideration in this thesis and may be possible as soon as the flow modality can handle a generalised version of *RefSet* expressions.

Bibliography

[ABB06] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *POPL*, J. G. Morrisett and S. L. P. Jones, Eds. ACM, 2006, pp. 91–102.

[Bec00] B. Beckert, "A dynamic logic for the formal verification of java card programs," in *Java Card Workshop*, ser. Lecture Notes in Computer Science, I. Attali and T. P. Jensen, Eds., vol. 2041. Springer, 2000, pp. 6–24.

[BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. LNCS 4334. Springer-Verlag, 2007.

[BN02] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a java-like language," in *CSFW*. IEEE Computer Society, 2002, pp. 253–.

[GJJ96] J. Gosling, W. N. Joy, and G. L. S. Jr., *The Java Language Specification*. Addison-Wesley, 1996.

[GM82] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[HKN06] C. Hammer, J. Krinke, and F. Nodes, "Intransitive noninterference in dependence graphs," in *2nd International Symposium on Leveraging Application of Formal Methods, Verification and Validation (ISoLA 2006)*, Nov. 2006, pp. 119–128.

[Hoa69] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[LBR06] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: a behavioral interface specification language for java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[Mye99] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *POPL*, A. W. Appel and A. Aiken, Eds. ACM, 1999, pp. 228–241.

[SM03] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[SS11] C. Scheben and P. H. Schmitt, "Verification of information flow properties of java programs without approximations," in *FoVeOOS*, ser. Lecture Notes in Computer Science, B. Beckert, F. Damiani, and D. Gurov, Eds., vol. 7421. Springer, 2011, pp. 232–249.

[SSB+13] C. Scheben, P. Schmitt, B. Beckert, V. Klebanov, M. Ulbrich, and D. Bruns, "Information flow in object-oriented software," in *Logic-Based Program Synthesis and Transformation, LOPSTR 2013*, G. Gupta, Ed., 2013.

[Sun03] Sun Microsystems, "Java Card 2.2.1 Specification," 2003.

[Wei11] B. Weiß, "Deductive verification of object-oriented software: Dynamic frames, dynamic logic and predicate abstraction," Ph.D. dissertation, Karlsruhe Institute of Technology, 2011.

Appendix

A. JavaDL* Rules

$$\textsc{emptyModality}^* \frac{\Gamma \Rightarrow \Delta, \{\mu\}(T \subseteq F)}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \mid F \mid T \ ]\!]}$$

$$\textsc{subset}_1^* \frac{\Gamma \Rightarrow \Delta, \{\mu\}(T_V \subseteq F_V) \qquad \Gamma \Rightarrow \Delta, \{\mu\}(T_L \subseteq F_L)}{\Gamma \Rightarrow \Delta, \{\mu\}(\langle T_V, T_L \rangle \subseteq \langle F_V, F_L \rangle)}$$

$$\textsc{subset}_2^* \frac{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in s) \qquad \Gamma \Rightarrow \Delta, \{\mu\}(V \subseteq s)}{\Gamma \Rightarrow \Delta, \{\mu\}(\{\mathtt{v}\} \cup V \subseteq s)}$$

$$\textsc{subset}_3^* \frac{\Gamma \Rightarrow \Delta, \{\mu\}True}{\Gamma \Rightarrow \Delta, \{\mu\}(\emptyset \subseteq s)}$$

$$\textsc{inEmpty}^* \frac{\Gamma \Rightarrow \Delta, \{\mu\}False}{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in \emptyset)}$$

$$\textsc{inSingleton}^* \frac{\Gamma \Rightarrow \Delta, \{\mu\}True}{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in \{\mathtt{v}\})}$$

$$\textsc{inUnion}^* \frac{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in s_1 \vee \mathtt{v} \in s_2)}{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in s_1 \cup s_2)}$$

$$\textsc{inSetMinus}^* \frac{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in s_1 \wedge \mathtt{v} \notin s_2)}{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in s_1 \setminus s_2)}$$

53

$$\textsc{inIntersect}^* \quad \frac{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in s_1 \wedge \mathtt{v} \in s_2)}{\Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{v} \in s_1 \cap s_2)}$$

$$\textsc{assignLocal}^* \quad \frac{\Gamma \Rightarrow \Delta, vars(\alpha) \subseteq F \qquad \Gamma \Rightarrow \Delta, \{\mu\}\{\mathtt{v} := \alpha\}[\![\ \theta\rho \mid F \cup \{\mathtt{v}\} \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v = } \alpha\mathtt{;}\ \rho \mid F \mid T\ ]\!]}$$

$$\textsc{assignLocal}_2^* \quad \frac{\Gamma \Rightarrow \Delta, \mathtt{r} \in F \qquad \Gamma, \{\mu\}(\delta(\mathtt{r}) \preceq D) \Rightarrow \Delta, \{\mu\}\{\mathtt{v} := cast_D(\mathtt{r})\}[\![\ \theta\rho \mid F \cup \{\mathtt{v}\} \mid T\ ]\!] \qquad \Gamma, \{\mu\}(\delta(\mathtt{r}) \not\preceq D) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ ClassCastException();}\ \rho \mid F \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v = (D)r;}\ \rho \mid F \mid T\ ]\!]}$$

$$\textsc{assignLocal}_3^* \quad \frac{\Gamma \Rightarrow \Delta, \{\mu\}\{\mathtt{v} := \alpha\}[\![\ \theta\rho \mid F \setminus \{\mathtt{v}\} \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v = } \alpha\mathtt{;}\ \rho \mid F \mid T\ ]\!]}$$

$$\textsc{assignLocal}_4^* \quad \frac{\begin{array}{l}\Gamma \Rightarrow \Delta, \mathtt{r} \in F \qquad \Gamma \Rightarrow \Delta, \{\mu\}\langle \mathtt{r}, f\rangle \in F \qquad \Gamma, \{\mu\}(\mathtt{r} \neq \mathtt{null}) \Rightarrow \Delta, \{\mu\}\{\mathtt{v} := select(\mathtt{heap}, \mathtt{r}, f)\}[\![\ \theta\rho \mid F \cup \{\mathtt{v}\} \mid T\ ]\!] \\ \qquad\qquad \Gamma, \{\mu\}(\mathtt{r} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ NullPointerException();}\ \rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v = r.f;}\ \rho \mid F \mid T\ ]\!]}$$

$$\textsc{assignLocal}_5^* \quad \frac{\begin{array}{l}\Gamma \Rightarrow \Delta, \mathtt{r} \in F \qquad \Gamma, \{\mu\}(\mathtt{r} \neq \mathtt{null}) \Rightarrow \Delta, \{\mu\}\{\mathtt{v} := select(\mathtt{heap}, \mathtt{r}, f)\}[\![\ \theta\rho \mid F \setminus \{\mathtt{v}\} \mid T\ ]\!] \\ \qquad \Gamma, \{\mu\}(\mathtt{r} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ NullPointerException();}\ \rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v = r.f;}\ \rho \mid F \mid T\ ]\!]}$$

assignLocal$^*_6$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{a} \in F \qquad \Gamma \Rightarrow \Delta, \mathtt{i} \in F \\ \Gamma \Rightarrow \Delta, \{\mu\}\langle \mathtt{a}, arr(\mathtt{i})\rangle \in F \qquad \Gamma, \{\mu\}(\mathtt{a} \neq \mathtt{null}), \{\mu\}(0 \leq \mathtt{i} < \mathtt{length}(\mathtt{a})) \Rightarrow \Delta, \{\mu\}\{\mathtt{v} := select(\mathtt{heap}, \mathtt{a}, arr(\mathtt{i}))\}[\![\ \theta\rho \mid F \cup \{\mathtt{v}\} \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{a} \neq \mathtt{null}), \{\mu\}(\mathtt{i} < 0 \vee \mathtt{i} \geq \mathtt{length}(\mathtt{a})) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ ArrayIndexOutOfBoundsException();}\ \rho \mid F \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{a} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ NullPointerException();}\ \rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v = a[i];}\ \rho \mid F \mid T\ ]\!]}$$

assignLocal$^*_7$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{a} \in F \qquad \Gamma \Rightarrow \Delta, \mathtt{i} \in F \qquad \Gamma, \{\mu\}(\mathtt{a} \neq \mathtt{null}), \{\mu\}(0 \leq \mathtt{i} < \mathtt{length}(\mathtt{a})) \Rightarrow \Delta, \{\mu\}\{\mathtt{v} := select(\mathtt{heap}, \mathtt{a}, arr(\mathtt{i}))\}[\![\ \theta\rho \mid F \setminus \{\mathtt{v}\} \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{a} \neq \mathtt{null}), \{\mu\}(\mathtt{i} < 0 \vee \mathtt{i} \geq \mathtt{length}(\mathtt{a})) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ ArrayIndexOutOfBoundsException();}\ \rho \mid F \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{a} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ NullPointerException();}\ \rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v = a[i];}\ \rho \mid F \mid T\ ]\!]}$$

assignField*

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{r} \in F \qquad \Gamma \Rightarrow \Delta, \mathtt{v} \in F \qquad \Gamma, \{\mu\}(\mathtt{r} \neq \mathtt{null}) \Rightarrow \Delta, \{\mu\}\{\mathtt{heap} := store(\mathtt{heap}, \mathtt{r}, f, \mathtt{v})\}[\![\ \theta\rho \mid F \cup \{\{\mu\}\langle \mathtt{r}, f\rangle\} \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{r} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ NullPointerException();}\ \rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{r.f = v;}\ \rho \mid F \mid T\ ]\!]}$$

assignField$^*_2$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{r} \in F \qquad \Gamma, \{\mu\}(\mathtt{r} \neq \mathtt{null}) \Rightarrow \Delta, \{\mu\}\{\mathtt{heap} := store(\mathtt{heap}, \mathtt{r}, f, \mathtt{v})\}[\![\ \theta\rho \mid F \setminus \{\{\mu\}\langle \mathtt{r}, f\rangle\} \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{r} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ NullPointerException();}\ \rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{r.f = v;}\ \rho \mid F \mid T\ ]\!]}$$

assignArray*

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{a} \in F \qquad \Gamma \Rightarrow \Delta, \mathtt{i} \in F \\ \Gamma \Rightarrow \Delta, \mathtt{v} \in F \qquad \Gamma, \{\mu\}(\mathtt{a} \neq \mathtt{null}), \{\mu\}(0 \leq \mathtt{i} < \mathtt{length}(\mathtt{a})) \Rightarrow \Delta, \{\mu\}\{\mathtt{heap} := store(\mathtt{heap}, \mathtt{a}, arr(\mathtt{i}), \mathtt{v})\}[\![\ \theta\rho \mid F \cup \{\{\mu\}\langle \mathtt{a}, arr(\mathtt{i})\rangle\} \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{a} \neq \mathtt{null}), \{\mu\}(\mathtt{i} < 0 \vee \mathtt{i} \geq \mathtt{length}(\mathtt{a})) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ ArrayIndexOutOfBoundsException();}\ \rho \mid F \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{a} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ NullPointerException();}\ \rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{a[i] = v;}\ \rho \mid F \mid T\ ]\!]}$$

55

ASSIGNARRAY$_2^*$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{a} \in F \\ \Gamma \Rightarrow \Delta, \mathtt{i} \in F \qquad \Gamma, \{\mu\}(\mathtt{a} \neq \mathtt{null}), \{\mu\}(0 \leq \mathtt{i} < \mathtt{length(a)}) \Rightarrow \Delta, \{\mu\}\{\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{a}, \mathit{arr}(\mathtt{i}), \mathtt{v})\}[\![\ \theta\rho \mid F \setminus \{\{\mu\}\langle \mathtt{a}, \mathit{arr}(\mathtt{i})\rangle\} \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{a} \neq \mathtt{null}), \{\mu\}(\mathtt{i} < 0 \vee \mathtt{i} \geq \mathtt{length(a)}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ ArrayIndexOutOfBoundsException();}\ \rho \mid F \mid T\ ]\!] \\ \Gamma, \{\mu\}(\mathtt{a} = \mathtt{null}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ new\ NullPointerException();}\ \rho \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{a[i]\ =\ v;}\ \rho \mid F \mid T\ ]\!]}$$

CREATEOBJECT*

$$\frac{\begin{array}{l}\Gamma, o \neq \mathtt{null}, \mathit{exactInstance}_C(o), \{\mu\}(\mathit{wellformed}(\mathtt{heap}) \rightarrow \mathit{select}_{Boolean}(\mathtt{heap}, o, \mathit{created}) = \mathit{False}) \\ \Rightarrow \Delta, \{\mu\}\{\mathtt{heap} := \mathit{create}(\mathtt{heap}, o)\}\{\mathtt{v} := o\}[\![\ \theta\rho \mid F \cup \{\mathtt{v}\} \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{v\ =\ C.alloc();}\ \rho \mid F \mid T\ ]\!]}$$

THROW*

$$\frac{\Gamma \Rightarrow \Delta, \mathtt{r} \in F \qquad \Gamma \Rightarrow \Delta, \{\mu\}\{\mathtt{exc} := \mathtt{r}\}[\![\ \theta\, \rho_2 \mid F \cup \{\mathtt{exc}\} \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ r;}\ \rho_1\rho_2 \mid F \mid T\ ]\!]}$$

THROW$_2^*$

$$\frac{\Gamma \Rightarrow \Delta, \{\mu\}\{\mathtt{exc} := \mathtt{r}\}[\![\ \theta\, \rho_2 \mid F \setminus \{\mathtt{exc}\} \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{throw\ r;}\ \rho_1\rho_2 \mid F \mid T\ ]\!]}$$

CATCH*

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{exc} \in F \\ \Gamma, \{\mu\}\mathit{instanceof}_E(\mathtt{exc}) \Rightarrow \Delta, \{\mu\}\{\mathtt{e} := \mathtt{exc}\}\{\mathtt{exc} := \mathtt{null}\}[\![\ \theta\, \pi_f\ \ \pi_c\, \rho_2 \mid F \mid T\ ]\!] \qquad \Gamma, \{\mu\}\neg\mathit{instanceof}_E(\mathtt{exc}) \Rightarrow \Delta, \{\mu\}[\![\ \theta\rho_1^{\mathtt{try}}\rho_2 \mid F \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{try\ \{\ \}\ catch\ (E\ e)}\ \{\pi_c\}\ \rho_1\ \mathtt{finally}\ \{\pi_f\}\, \rho_2 \mid F \mid T\ ]\!]}$$

FINALLY*

$$\frac{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \pi_f\, \rho \mid F \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \mathtt{try\ \{\ \}\ finally}\ \{\pi_f\}\, \rho \mid F \mid T\ ]\!]}$$

CONDITIONAL*

$$\frac{\Gamma \Rightarrow \Delta, \mathtt{c} \in F \qquad \Gamma, \{\mu\}(\mathtt{c} = True) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \pi_t\, \rho \mid F \mid T\ ]\!] \qquad \Gamma, \{\mu\}(\mathtt{c} = False) \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \pi_e\, \rho \mid F \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \texttt{if (c) \{}\pi_t\texttt{\} else \{}\pi_e\texttt{\}}\, \rho \mid F \mid T\ ]\!]}$$

CONDITIONAL$_2^*$

$$\frac{\Gamma \Rightarrow \Delta, \{\mu\}\mathcal{V}[\![\ \theta\, \rho \mid F \setminus V \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \texttt{if (c) \{}\pi_t\texttt{\} else \{}\pi_e\texttt{\}}\, \rho \mid F \mid T\ ]\!]}$$

V is a set that contains all program variables and heap locations that may be altered by an execution of π_t or π_e. $\mathcal{V}$ is the update that results from assigning random values to the memory locations included in V and thus erasing all information about them after the conditional. `throw` and `return` statements must not occur in both π_t and π_e.

LOOP*

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \{\mu\}(I \subseteq F) \qquad \Gamma \Rightarrow \Delta, \{\mu\}(\mathtt{c} \in I) \qquad \Gamma \Rightarrow \Delta, \{\mu\}Inv \qquad \Gamma, \{\mu\}\mathcal{V}(\mathtt{c} = True \wedge Inv) \Rightarrow \Delta, \{\mu\}\mathcal{V}[\pi]Inv \\ \Gamma, \{\mu\}\mathcal{V}(\mathtt{c} = True \wedge Inv) \Rightarrow \Delta, \{\mu\}\mathcal{V}[\![\ \pi \mid I \mid I\ ]\!] \qquad \Gamma, \{\mu\}\mathcal{V}(\mathtt{c} = False \wedge Inv) \Rightarrow \Delta, \{\mu\}\mathcal{V}[\![\ \theta\rho \mid I \mid T\ ]\!]\end{array}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \texttt{while (c) \{}\pi\texttt{\}}\, \rho \mid F \mid T\ ]\!]}$$

FALLBACK

$$\frac{\Gamma \Rightarrow \Delta, \{\mu\}\Phi_{\pi,F,T}}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \pi \mid F \mid T\ ]\!]}$$

RETURN$_1^*$

$$\frac{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \texttt{v = r;}\, \rho_2 \mid F \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \texttt{method-frame(result=v,this=o) \{ return r;}\ \rho_1\ \texttt{\}}\, \rho_2 \mid F \mid T\ ]\!]}$$

RETURN$_2^*$

$$\frac{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \texttt{throw r;}\, \rho_2 \mid F \mid T\ ]\!]}{\Gamma \Rightarrow \Delta, \{\mu\}[\![\ \theta\, \texttt{method-frame(result=v,this=o) \{ throw r;}\ \rho_1\ \texttt{\}}\, \rho_2 \mid F \mid T\ ]\!]}$$

57