Information Flow in Object-Oriented Software

Bernhard Beckert, Daniel Bruns(✉), Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich

Department of Informatics, Karlsruhe Institute of Technology (KIT),
Am Fasanengarten 5, 76131 Karlsruhe, Germany
bruns@kit.edu
http://www.key-project.org/DeduSec/

Abstract. This paper contributes to the investigation of object-sensitive information flow properties for sequential Java, i.e., properties that take into account information leakage through objects, as opposed to primitive values. We present two improvements to a popular object-sensitive non-interference property. Both reduce the burden on analysis and monitoring tools. We present a formalization of this property in a program logic – JavaDL in our case – which allows using an existing tool without requiring program modification. The third contribution is a novel fine-grained specification methodology. In our approach, arbitrary JavaDL terms (read 'side-effect-free Java expressions') may be assigned a security level – in contrast to security labels being attached to fields and variables only.

1 Introduction

The growing reliance of our daily lives on software systems of all kinds has increased the demand for software quality assurance. A particular concern is confidentiality of sensitive data: preventing information flow from secret (also called *high*) sources to publicly observable (also called *low*) sinks. Methods for specification and analysis of information flow play an important role in answering these concerns. Since the pioneering papers [11,13,14,21], research in information flow has grown considerably and diversified in numerous branches. This paper follows a language-based approach dealing with programs at the code level (instead of analyzing abstractions such as automata or process algebras). We will use a semantic definition of information flow, as e.g., introduced in [20]. For the analysis of information flow properties, we use a program logic, along the lines of [2,12], as opposed to the use of security type systems or dedicated analysis algorithms. Logical information flow analysis started out by investigating simple imperative programming languages and later also targeted object-oriented languages [1,9,28,32].

In imperative languages, sources and sinks are of primitive type. In an object oriented context, it is natural to consider sources and sinks of object type, too.

This work was supported by the German National Science Foundation (DFG) under project "Program-level Specification and Deductive Verification of Security Properties" within priority programme 1496 "Reliably Secure Software Systems – RS^3".

G. Gupta and R. Peña (Eds.): LOPSTR 2013, LNCS 8901, pp. 19–37, 2014.
DOI: 10.1007/978-3-319-14125-1_2

In this case, the usual definition of secure information flow – if a system is started in two low-equivalent states s_1, s_2 with all publicly observable values equal, then it terminates in states s_1', s_2' where all observable values are equal – is too strong. It has been replaced by object-sensitive secure information flow [3,6,9,17,18,28] that has a modified notion of low-equivalence of states: if a system is started in two states s_1, s_2 such that the observable values are related by a partial isomorphism π, then it terminates in states s_1', s_2' where all observable values are related by a partial isomorphism extending π.

A desirable outcome in developing logic-based information flow analyses is to find formalizations that allow application of existing verification tools. An instance of such reuse is [28], where the two runs of a program are encoded into a single program, allowing specifying secure information flow with JML and verifying it with the ESC/Java2 tool. The particular encoding, though, relies on ghost fields and requires instrumenting the program under investigation with ghost code. The first contribution of our paper is a formalization of object-sensitive secure information flow in Dynamic Logic that does not require any changes or additions to the investigated program. The KeY system [8] can be used to discharge the ensuing proof obligations.

To avoid loss of precision, it is reasonable to encode the partial isomorphisms of object-sensitive secure information flow explicitly in the logical formalization. This, on the other hand, holds the disadvantage that a naive encoding either increases the burden on the analysis or the burden on the user, the latter by requiring additional annotations [28]. The second contribution of this paper is an investigation into the concept of object-sensitive secure information flow itself with the aim to find alternative but equivalent formulations such that the partial isomorphisms can be restricted as much as possible. We prove (Lemma 4) that restricting the partial isomorphism π in the pre-state to be the identity still leads to an equivalent concept. We also show that additionally restricting the partial isomorphism in the post-state to newly created objects leads to a sufficient criterion for object-sensitive secure information flow (Thm. 2). Further we show that compositionality, which is considered an indispensable prerequisite for modular verification of information-flow properties and which holds for object-sensitive secure information flow only under certain conditions, holds for the sufficient criterion in general (Thm. 3). The main difference between the original property and the sufficient criterion is that the criterion admits the attacker the ability to distinguish between newly created objects and objects which already existed in the pre-state. This leads to a slightly stronger property. All three results hold the potential of significantly reducing the burden on analysis and monitoring tools. They apply under the assumption that references in Java are treated as opaque, as formalized by Postulate 1.

As a third contribution, we introduce a specification methodology where security levels (high or low) are assigned to arbitrary JavaDL terms. The set of publicly observable memory locations is thus state-dependent. This is in contrast to the typical static labeling of fields or variables only, and permits fine-grained specifications, which are especially useful for declassification.

2 Dynamic Logic for Java

In this section, we briefly review syntax and semantics of JavaDL, a Dynamic Logic for Java, as far as needed in this paper. An in-depth account can be found in [8,33]. JavaDL is an extension of classical typed first-order logic with equality (the equality symbol is denoted by $\doteq$), with which we assume the reader is familiar. The following explanations only address particularities and the modal extension.

The notion of a term in JavaDL is the same as in typed first-order logic. We assume that, among others, constant and function symbols are available for all local program variables, instance and static fields, `this`, `result` and method parameters, and operations of Java primitive data types. In addition, we make use of a special implicit program variable `heap` which stands for the current heap.

JavaDL formulas are inductively built up from atomic formulas using propositional operators and quantifiers, as usual. In addition

1. $\{a := t\}\phi$ is a JavaDL formula, where a is a term which refers to a location (a program variable, a static or dynamic field, or an array entry), t is a JavaDL term, and ϕ is a formula. $\{a := t\}$ is called an *update*.
2. For a JavaDL formula ϕ and any sequential Java program α, both $\langle\alpha\rangle\,\phi$ and $[\alpha]\,\phi$ are again JavaDL formulas.[1]

The basis JavaDL semantics is a structure $\mathcal{D}$ for typed first-order logic, called the *computation domain*. $\mathcal{D}$ provides the interpretation of all state-independent (sometimes also called *rigid*) function and predicate symbols. In our setup, program variables are the only non-rigid symbols. The universe D of $\mathcal{D}$ is divided into the interpretations $T^{\mathcal{D}}$ for the types T occurring in the language. In particular, we assume the existence of types *Any*, *Obj*, *Heap*, *Field*, *Int* with $Any^{\mathcal{D}} = D$, $Obj^{\mathcal{D}}$ = the set of all objects, $Heap^{\mathcal{D}}$ = the set of all heaps, $Field^{\mathcal{D}}$ = the set of all fields, $Int^{\mathcal{D}} = \mathbb{Z}$, $Seq^{\mathcal{D}}$ = the set of all finite nested sequences of values from D, and a subtype relation $\sqsubset$ such that the Java reference type hierarchy lies under $Obj \sqsubset Any$. Moreover, *Heap*, *Field*, and *Obj* are pairwise disjoint.

A *state* s is a function mapping all program variables to properly typed values in $\mathcal{D}$. By $\mathcal{D} + s$ we denote the first-order structure that interprets all, rigid and non-rigid, symbols. In most cases $\mathcal{D}$ will be implicitly understood and we write s instead of $\mathcal{D} + s$. For any state s and term t without logical variables, the evaluation t^s is as usual. If t contains logical variables, a variable assignment β is needed to evaluate the term to $t^{s,\beta}$. In the following, we will omit β whenever it is not essential. The (current) heap in a state s is completely determined by $\texttt{heap}^s$: the value $(t.f)^s$ of a field access expression $t.f$ is obtained by $select(\texttt{heap}^s, t^s, c_f)$. Here c_f is a constant symbol representing the Java field f,

[1] The definition is in fact more liberal in that α need not be a compilable program. Precisely which program sequences are allowed is explained in [8, Sect. 3.2.4]. We will nevertheless use the term 'program' synonymously.

$$\forall Int\ i((0 \leq i\ \wedge\ i < maxvalue)\ \rightarrow \\ \{a := i\}\,\langle\alpha\rangle\,(0 \leq \mathtt{r} \wedge \mathtt{r} * \mathtt{r} \leq i \wedge (\mathtt{r}+1) * (\mathtt{r}+1) > i)) \tag{1}$$

$$\forall Heap\ h, h'\ \forall Int\ i, i'(\quad (select(h, \mathtt{this}, f) \doteq select(h', \mathtt{this}, f) \wedge \\ \{\mathtt{heap} := h\}\,\langle \mathtt{m();}\rangle\, i \doteq \mathtt{r}\ \wedge\ \{\mathtt{heap} := h'\}\,\langle \mathtt{m();}\rangle\, i' \doteq \mathtt{r}) \rightarrow i \doteq i') \tag{2}$$

Fig. 1. Two examples of JavaDL formulas

select and its counterpart *store* are state-independent functions from the theory of arrays, see [24,29].

The recursive definition of the relation $s \models \phi$ (formula ϕ is true in state s) follows the usual pattern. Only the three modal operators need explanation. For a JavaDL formula ϕ and state s, we define:

1. $s \models \{a := t\}\phi$ iff $s' \models \phi$, where s' coincides with s except for $s'(a) = t^s$.
2. $s \models \langle\alpha\rangle\,\phi$ iff $s' \models \phi$ for some s' such that α started in s terminates in s'.
3. $s \models [\alpha]\,\phi$ iff $s' \models \phi$ for all s' such that α started in s terminates in s'.

If program α does not terminate when started in state s, then $s \models [\alpha]\,\phi$ is trivially true for all formulas ϕ, including $\phi \equiv \mathit{false}$.

Let us look at the examples of Fig. 1. Formula (1) expresses that program α computes the positive integer square root for any positive input a (`result` is abbreviated by `r`). Formula (2) states that the return value of method `m()` only depends on the field `this`.f. Logical variables cannot occur in programs and program variables may not be quantified over. As these examples demonstrate, updates can be used as an interface between both types of variables.

We adopt the *constant domain* approach (see for instance [8,33]), i.e., all potential objects are contained in D from the start. The generation of a new object of type T in state s is effected by changing the value of $o.created$ from ff to tt, where $o = {nextToCreate_T}^{\mathcal{D}}(s)$, with ${nextToCreate_T}^{\mathcal{D}}$ being the function which selects the next new object of type T to create depending on the state. In this paper computation domains $\mathcal{D}_1$, $\mathcal{D}_2$ will at most differ in the interpretation ${nextToCreate_T}^{\mathcal{D}_i}$. Only functions ${nextToCreate_T}^{\mathcal{D}}$ with ${created}^{\mathcal{D}+s}({nextToCreate_T}^{\mathcal{D}}(s)) = ff$ and ${exactInstance_T}^{\mathcal{D}}({nextToCreate_T}^{\mathcal{D}}(s)) = tt$ are considered.

Let α be a program, $\mathcal{D}$ a computation domain and let s_1, s_2 be states. We denote "α started in $\mathcal{D} + s_1$ terminates in $\mathcal{D} + s_2$" by $\mathcal{D} + s_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D} + s_2$.

3 Information Flow in Java

In Fig. 2 we reproduce a typical example of object-sensitive information flow. If low-equivalence of states required the values of `x` and `y` to be equal, method `m1()` would be rated as insecure. However, we treat object references in Java as

```
final class C {
  static C x, y;      // low variables
  static boolean h;   // high variable
  static void m1() { if (h) {x = new C(); y = new C();}
                     else   {y = new C(); x = new C();} } }
```

Fig. 2. Secure object creation

opaque, i.e., references can only be compared by the `==` function, cf. [23]. Thus `m1()` obviously does not leak information.[2]

We describe publicly (and thus attacker-) visible parts of the program state as sets of JavaDL terms. The attacker sees the term and the corresponding evaluation in the pre- and post-state of a method as if they were printed on a screen. Further, we assume that the attacker knows the program code. This allows them to trace back the observed differences in low values in the post-state to high values in the pre-state. In summary, an attacker *can* compare observed values that are of a primitive type to each other and to literals (of that type) as by using `==`; *can* compare observed values of object reference type to each other and to `null` as by using the `==` predicate and observe their (runtime) type and the length attribute for array references; *cannot* learn more than object identity from object references (e.g., the order in which objects have been generated cannot be learned).

Formally, we call a sequence of JavaDL terms (which itself is a JavaDL term), an *observation expression*. The low locations of Fig. 2, for instance, give rise to the observation expression $\langle \texttt{C.x}, \texttt{C.y} \rangle$. Let R be an observation expression and s a state. An attacker is able to observe the tuple (R, R^s), where $R^s = \langle e_1^s, \ldots, e_k^s \rangle$ if $R = \langle e_1, \ldots, e_k \rangle$. Hence, they are able to deduce for any $1 \leq i \leq k$ that e_i^s is the value of the term e_i. Additionally, an attacker can learn the result of the comparison of any two values $e_i^s = e_j^s$ and, in case of reference values, retrieve their runtime type $type(e_i^s)$ and, for array references, their length $len(e_i^s)$.

Definition 1. *By $Obj(R^s)$ we denote the set of objects observable by R in state s, that is, $Obj(R^s) = \{o \in Obj^{\mathcal{D}} \mid \exists i(o = R^s[i])\} \cup \bigcup_{i \in \{j \mid R^s[j] \in Seq^{\mathcal{D}}\}} Obj(R^s[i])$.*

In an object-oriented setting, what is observable may depend on the state. For example, if *o.next.val* is observable, then it depends on the state what object *o.next* evaluates to. Moreover, if all locations in a linked list are observed, then the *number* of observable locations may depend on the state, since the list length does.

Observation expressions cover such cases: JavaDL includes a sequence definition operator $seq\{i\}(from, to, e)$ with the semantics $[seq\{i\}(from, to, e)]^s =$

[2] In [19] it has been demonstrated that this abstraction might be broken, e.g., by the implementation of native methods such as `Object::hashCode()`.This potential leakage can be dealt with by assigning a high security level to the output of native methods or by using the security type analysis proposed in the quoted paper.

$\langle [e^{[i\to n]}]^s,\ [e^{[i\to n+1]}]^s,\ \ldots,\ [e^{[i\to m-1]}]^s\rangle$, if $from^s = n < m = to^s$ are integers. Here $e^{[i\to n]}$ is the term obtained from e by replacing all occurrences of the variable i by the literal n. Further JavaDL contains a reachability operator $e.it(f,i)$, where e is a JavaDL term of type T, f is an attribute defined in class T and also of type T and i is an integer term. The semantics of $e.it(f,i)$ is defined by $[e.it(f,i)]^s = f.\ldots.f([e]^s)$ (k times) with $k = i^s$. The observation of all elements of a linked list can be modeled by the observation expression $seq\{i\}(0, \texttt{list.len}, \texttt{list}.it(\texttt{next}, i).\texttt{val})$.

Here and in the following we abbreviate `length` by `len`. Further, we write sequences of fixed length as $\langle\ldots\rangle$ and denote the concatenation of two sequences R_1 and R_2 by $R_1; R_2$. For uniformity of notation we will frequently write $f(e_0)$ instead of $e_0.f$.

Our approach generalizes and unifies declassification of terms [4,31]. It already proved to be useful in a recent case-study [15] which uses our approach and implementation. Here, whether information is considered secret or public depends on the internal state of the system. Therefore, the information flow specifications of [15] make use of conditional terms. Another application is information flow class invariants: if a program or library has a public interface with several methods, then often it has to be ensured that any sequence of calls to those methods is secure. For this purpose it is useful to define the knowledge of the caller by a list of terms. The program is secure, if for any method of the interface the final values of those terms depend at most on their initial values. An illustrating example can be found in the companion technical report [7].

4 Isomorphisms

We assume that the reader is familiar with the concept of isomorphism for typed structures [25]. In this section we collect the results needed later on for easy reference.

We will consider isomorphisms only on the computation domain $\mathcal{D}$, and the structures $\mathcal{D}+s$ (see Section 2) for different states s. If π is an isomorphism from $\mathcal{D}+s_1$ onto $\mathcal{D}+s_2$, we will say that s_2 is isomorphic to s_1 and write $s_2 = \pi(s_1)$. We will need the following (folklore) results:

Lemma 1. *Let ρ be an automorphism of $\mathcal{D}$, s a state, ϕ a formula, e an expression. Then* $s \models \phi \Leftrightarrow \rho(s) \models \phi$ *and* $e^{\rho(s)} = \rho(e^s)$.

Lemma 2. *Let $\mathcal{D}$ be a computation domain and π' be a bijection from X onto Y for finite subsets $X, Y \subseteq Obj^{\mathcal{D}}$ with*

1. *If $null \in X$ then $\pi'(null) = null$ and $null \in Y$ implies $null \in X$.*
2. *π' preserves the exact types of its arguments.*
3. *π' preserves the length of array objects.*

Then there is a computation domain $\mathcal{D}'$ and an isomorphism $\pi : \mathcal{D} \to \mathcal{D}'$ extending π'.

Definition 2 (Partial isomorphism w.r.t. R). *Let R be an observation expression and s_1, s_2 be two states.*

A partial isomorphism *with respect to R from s_1 to s_2 is a bijection $\pi : Obj(R^{s_1}) \to Obj(R^{s_2})$ such that (a) the requirements of Lemma 2 hold and (b) $\pi_{Seq}(R^{s_1}) = R^{s_2}$ where π_{Seq} is defined on sequences as $\pi_{Seq}(\langle e_1, \ldots, e_k\rangle) = \langle e'_1, \ldots, e'_k\rangle$ with $e'_i = \pi(e_i)$ if $e_i \in Obj^{\mathcal{D}}$, $e'_i = \pi_{Seq}(e_i)$ if $e_i \in Seq^{\mathcal{D}}$ and $e'_i = e_i$ else.*

It will greatly simplify notation to stipulate that every partial isomorphism π is also defined on all primitive values w with $\pi(w) = w$.

If $p \in R$ for all program variables p, every automorphism extending a partial isomorphism π with respect to R according to Lemma 2 is a total isomorphism from $\mathcal{D} + s_1$ onto $\mathcal{D} + s_2$ since $\pi(p^{s_1}) = p^{s_2}$ by requirement (b).

Not every partial isomorphism can be extended to a total isomorphism, on the other hand. If q is a program variable such that q does not appear as a subterm in R, then $\pi(q^{s_1}) = q^{s_2}$ is not required.

To clarify the role of the additional condition (b) in Def. 2 let x be a program variable of type C and f a field in C, say of type integer such that $R = \langle x, f(x)\rangle$ and let s_1, s_2 be states. In this case the condition implies $\pi((f(x))^{s_1}) = (f(x))^{s_2} = f^{s_2}(x^{s_2}) = f^{s_2}(\pi(x^{s_1}))$. This amounts to the usual requirements on isomorphisms on mathematical structures.

5 Formalizing Information Flow

As mentioned before, we treat object references as opaque. This means in particular that the behavior of a Java program cannot depend on the values of references up to comparison by `==`. Hence, if a program α is started in two isomorphic states, then α also terminates in isomorphic states (if α terminates.) Though this assumption is not always made explicit, it is widely used in the literature [1,3,26,28]. Opaqueness of references can be formalized in our setting as follows:

Postulate 1. *Let s_1, s_2 be states. Let α be a program which started in s_1 terminates in s_2, and let $\rho : \mathcal{D} \to \mathcal{D}'$ be an isomorphism from computation domain $\mathcal{D}$ onto computation domain $\mathcal{D}'$.*

Then α started in $\mathcal{D}' + \rho(s_1)$ terminates in $\rho'(s_2)$, where $\rho' : \mathcal{D} \to \mathcal{D}'$ is an isomorphism that coincides with ρ on all objects existing in state s_1, i.e. for all $o \in Obj^{\mathcal{D}}$ with $created^{s_1}(o) = tt$ we know $\rho(o) = \rho'(o)$. (See beginning of Sect. 4 for the definition of $\rho(s_i)$.)

The reason why we cannot assume $\rho = \rho'$, is that α may generate new objects and there is no reason why a new element o' generated in the run starting in state $\mathcal{D}' + \rho(s_1)$ should be the ρ-image of the new element o generated in the run of α starting in state $\mathcal{D} + s_1$.

5.1 Basic Information Flow Definition and Its Properties

We start with formalizing the basic object-sensitive non-interference property for Java. Apart from the more flexible assignment of security levels, this property does not yet exceed the state of the art in object-sensitive non-interference (cf. Sect. 1). We consider here the termination-insensitive case. Extensions taking termination into account, as well as differentiating between normal and abnormal termination, are straightforward.

Definition 3 (Agreement of states). *Let R be an observation expression. We say that two states s, s' agree on R, abbreviated by* $\mathrm{agree}(R, s, s')$*, iff there exists a partial isomorphism $\pi : Obj(R^{s_1}) \to Obj(R^{s_2})$ with respect to R. The partial isomorphism π is uniquely determined by R, s and s'. We use the notation* $\mathrm{agree}(R, s, s', \pi)$ *to indicate that* $\mathrm{agree}(R, s, s')$ *is true and π is the mapping thus defined.*

Notice that because of our tacit agreement on the values of partial isomorphisms on primitive values, $\mathrm{agree}(R, s, s')$ entails $(e_i)^s = (e_i)^{s'}$, if e_i is a term of primitive type.

We now define what it means for a program α (when started in a state s) to allow information flow only from R_1 to R_2, a fact which we denote by $\mathrm{flow}(s, \alpha, R_1, R_2)$. The intuition is that R_1 describes the low locations in the pre-state and R_2 describes the low locations in the post-state. Thus, the values of the variables and locations in R_2 in the post-state must at most depend – up to isomorphism of states – on the values of the variables and locations in R_1 in the pre-state and on nothing else.

Definition 4 (The predicate flow**).** *Let α be a program and R_1 and R_2 be two observation expressions.*

Program α allows information to flow only from R_1 to R_2 when started in s_1, denoted by $\mathrm{flow}(s_1, \alpha, R_1, R_2)$*, iff, for computation domains $\mathcal{D}$, $\mathcal{D}'$ and all states s_1', s_2, s_2' such that $\mathcal{D} + s_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D} + s_2$ and $\mathcal{D}' + s_1' \overset{\alpha}{\rightsquigarrow} \mathcal{D}' + s_2'$, we have*
if $\mathrm{agree}(R_1, s_1, s_1', \pi^1)$ *for some π^1*
then $\mathrm{agree}(R_2, s_2, s_2', \pi^2)$ *for some π^2 that is compatible with π^1*
where π^2 is said to be compatible with π^1 if
$\pi^2(o) = \pi^1(o)$ for all $o \in Obj(R_1^{s_1}) \cap Obj(R_2^{s_2})$ with $created^{s_1}(o) = tt$.

In the most common case, the *low* locations before program execution will be the same as the *low* locations after program execution, i.e., $R_1 = R_2$. But, that may not be true in all cases. To *declassify* an expression e_{decl}, one would choose $R_1 = R_2; e_{decl}$.

Consider the following extension of method `m1()` from Fig. 2 as if defined in class `C`:

```
C next;
static void m2() { if(h) {x=new C(); y=new C(); x.next=y;}
                   else {y=new C(); x=new C(); x.next=x;} }
```

Whether `m2()` leaks information or not depends on the examined observation expression. For $R = \langle C.x, C.y \rangle$ the observation will always consist of two freshly

created, distinct object references. If agree(R, s_1, s_1', π^1), the partial isomorphism π^2 defined as an extension of π^1 by $\pi_2(x^{s_2}) = x^{s_2'}$ and $\pi_2(y^{s_2}) = y^{s_2'}$ ensures that agree(R, s_2, s_2', π^2) and, therefore, flow$(s_1, \texttt{m2()}, R, R)$.

But if $R' = \langle C.x, C.y, C.x.next \rangle$ is chosen, π^2 is no longer a partial isomorphism as $\pi^2(next^{s_2}(x^{s_2})) = next^{s_2'}(x^{s_2'})$ would need to hold. But if $h^{s_1} = tt$ and $h^{s_1'} = ff$, the resulting heap structures are not isomorphic: $\pi^2(next^{s_2}(x^{s_2})) = \pi^2(y^{s_2})$ and $next^{s_2'}(x^{s_2'}) = x^{s_2'} = \pi^2(x^{s_2})$ which cannot be equal as π^2 is an injection. The attacker can learn the value of h by comparing x and $x.next$: flow$(s_1, \texttt{m2()}, R', R')$ does not hold.

For later reference we state the following lemma.

Lemma 3. *If* agree(R, s, s', π) *and* ρ *is an automorphism on* $\mathcal{D}$ *then also (1)* agree$(R, s, \rho(s'), \rho \circ \pi)$ *and (2)* agree$(R, \rho(s), s', \pi \circ \rho^{-1})$.

Proof. Part 1: By assumption the mapping π given by $\pi_{Seq}(R^s) = R^{s'}$ is a partial isomorphism, where π_{Seq} is defined as in Def. 2. Since π is a partial isomorphism and ρ is an automorphism also $\rho \circ \pi$ is a partial isomorphism. Further, let $(\rho \circ \pi)_{Seq}$ be defined as $(\rho \circ \pi)_{Seq}(\langle e_1, \ldots, e_k \rangle) = \langle e_1', \ldots, e_k' \rangle$ with $e_i' = \rho \circ \pi(e_i)$ if $e_i \in Obj^{\mathcal{D}}$, $e_i' = (\rho \circ \pi)_{Seq}(e_i)$ if $e_i \in Seq^{\mathcal{D}}$ and $e_i' = e_i$ else. Then $(\rho \circ \pi)_{Seq}(R^s) = \rho \circ \pi_{Seq}(R^s)$, because ρ is an automorphism on $\mathcal{D}$, and $\rho \circ \pi_{Seq}(R^s) = \rho(R^{s'})$, because of $\pi_{Seq}(R^s) = R^{s'}$. Finally we derive by Lemma 1 $\rho(R^{s'}) = R^{\rho(s')}$ and thus we have $(\rho \circ \pi)_{Seq}(R^s) = R^{\rho(s')}$. Hence agree$(R, s, \rho(s'), \rho \circ \pi)$ holds.

Part 2: By symmetry from Part 1. □

5.2 An Optimized but Equivalent Formulation

In this section, we introduce flow*, an optimized version of the flow property from Def. 4. The property flow* restricts the partial isomorphism of the pre-state to be the identity. This simplifies the formulation of verification conditions considerably (see Theorem 1 below), also making them easier to verify. Yet, it is semantically equivalent to flow.

Definition 5 (The flow* predicate). *Let* α *be a program and* R_1 *and* R_2 *be two observation expressions.*

We say that α *allows simple information flow only from* R_1 *to* R_2 *when started in* s_1*, denoted by* flow$^*(s_1, \alpha, R_1, R_2)$*, iff, for all computation domains* $\mathcal{D}$*,* $\mathcal{D}'$ *and states* s_1', s_2, s_2' *such that* $\mathcal{D} + s_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D} + s_2$ *and* $\mathcal{D}' + s_1' \overset{\alpha}{\rightsquigarrow} \mathcal{D}' + s_2'$*, we have*
if agree(R_1, s_1, s_1', id)
then agree(R_2, s_2, s_2', π^2) *for some* π^2 *compatible with id.*
Note that agree(R_1, s_1, s_1', id) *implies in particular* $Obj(R_1^{s_1}) = Obj(R_1^{s_1'})$ *since* $\pi^1 = id$ *is a bijection from* $Obj(R_1^{s_1})$ *onto* $Obj(R_1^{s_1'})$.

Lemma 4. *For all programs* α*, any two observation expressions* R_1 *and* R_2*, and any state* s_1 flow$^*(s_1, \alpha, R_1, R_2)$ $\Leftrightarrow$ flow(s_1, α, R_1, R_2) .

Proof. $\text{flow}(s_1, \alpha, R_1, R_2) \Rightarrow \text{flow}^*(s_1, \alpha, R_1, R_2)$ is obviously true. Thus it suffices to show $\text{flow}^*(s_1, \alpha, R_1, R_2) \Rightarrow \text{flow}(s_1, \alpha, R_1, R_2)$.

To prove $\text{flow}(s_1, \alpha, R_1, R_2)$ we fix, in addition to s_1, states s_1', s_2, s_2' such that $s_1 \overset{\alpha}{\rightsquigarrow} s_2$ and $s_1' \overset{\alpha}{\rightsquigarrow} s_2'$, and assume $\text{agree}(R_1, s_1, s_1', \pi^1)$. We need to show $\text{agree}(R_2, s_2, s_2', \pi^2)$ with π^2 extending π^1.

By Lemma 2, there is an automorphism ρ on $\mathcal{D}'$ extending $(\pi^1)^{-1}$. From $\text{agree}(R_1, s_1, s_1', \pi^1)$ we conclude $\text{agree}(R_1, s_1, \rho(s_1'), \rho \circ \pi^1)$ using Lemma 3. Since ρ extends $(\pi^1)^{-1}$ we have $\text{agree}(R_1, s_1, \rho(s_1'), id)$. By Postulate 1. there is a state s_3' and a computation domain $\mathcal{D}''$ such that $\mathcal{D}'' + \rho(s_1') \overset{\alpha}{\rightsquigarrow} \mathcal{D}'' + s_3'$. This enables us to make use of the assumption $\text{flow}^*(s_1, \alpha, R_1, R_2)$ and conclude $\text{agree}(R_2, s_2, s_3', \pi^3)$. Furthermore, $\pi^3(o) = o$ for all $o \in Obj(R_1^{s_1}) \cap Obj(R_2^{s_2})$.

Again, appealing to Postulate 1. in the situation that $\rho(s_1') \overset{\alpha}{\rightsquigarrow} s_3'$ and considering the inverse automorphism ρ^{-1}, we obtain an automorphism ρ' such that $\rho^{-1}(\rho(s_1')) = s_1' \overset{\alpha}{\rightsquigarrow} \rho'(s_3')$ and ρ' coincides with ρ^{-1} on all objects in $\{o \in Obj^{\mathcal{D}} \mid created^{\rho(s_1')}(o) = tt\}$.

Again, using Lemma 3, this time for the isomorphism ρ', we obtain from $\text{agree}(R_2, s_2, s_3', \pi^3)$ also $\text{agree}(R_2, s_2, \rho'(s_3'), \rho' \circ \pi^3)$. Since α is a deterministic program and we have already defined s_2' to be the final state of α when started in s_1' in the computation domain $\mathcal{D}'$ we get $s_2' = \rho'(s_3')$ and thus $\text{agree}(R_2, s_2, s_2', \rho' \circ \pi^3)$. Because π^2 is uniquely determined by R_2, s_2 and s_2', we have $\rho' \circ \pi^3 = \pi^2$.

Finally, we show that $\rho' \circ \pi^3$ extends π^1, i.e., for every $o \in Obj(R_1^{s_1}) \cap Obj(R_2^{s_2})$ with $created^{s_1}(o) = tt$ we need to show $\rho' \circ \pi^3(o) = \pi^1(o)$. Since $\pi^3(o) = o$ for $o \in Obj(R_1^{s_1}) \cap Obj(R_2^{s_2})$ it suffices to show $\pi^1(o) = \rho'(o)$. By the definition of isomorphic states we obtain from $created^{s_1}(o) = tt$ also $created^{\rho(s_1)}(o) = tt$. Thus $\rho'(o) = \rho^{-1}(o)$ and by choice of ρ further $\rho^{-1}(o) = \pi^1(o)$, as desired. □

6 Verification Conditions

The ultimate goal is to prove information flow properties $\text{flow}(s_1, \alpha, R_1, R_2)$ for particular observations R_i and a program α. To this end, specialized proof rules for the flow predicate could be introduced. We pursue this approach in another paper. Here, we will show how to derive verification conditions directly from the definition. We will show how $\text{flow}(s_1, \alpha, R_1, R_2)$ can be expressed by a JavaDL formula – to be discharged by a standard JavaDL calculus. This exposition should also convey the idea how to obtain verification conditions with methodologies other than Dynamic Logic.

Theorem 1. *Let α be a program, and let R_1, R_2 be observation expressions. There is a* JavaDL *formula ϕ_{α,R_1,R_2} making use of self-composition such that*

$$\mathcal{D} + s_1 \models \phi_{\alpha,R_1,R_2} \quad \textit{iff } \text{flow}(s_1, \alpha, R_1, R_2)$$
for all computation domains $\mathcal{D}$.

We will explain here the construction of ϕ_{α,R_1,R_2} only. The complete proof of Thm. 1 can be found in the companion technical report [7].

The property to be formalized requires quantification over states. A state s is determined by the value of the heap h^s in s and the values of the (finitely many) program variables a^s in s. We can directly quantify over heaps h and refer to the value of a field f of type C for the object o referenced by the term e as $selectC(h, e, f)$. We cannot directly quantify over program variables, as opposed to quantifying over the values of program variables, which is perfectly possible. Thus we use quantifiers $\forall x$, $\exists x$ over the type domain of the variable and assign x to a via an update $a := x$. There are four states involved, the two pre-states s_1, s_1' and the post-states s_2, s_2'. Correspondingly, there will be, for every program variable v, four universally quantifier variables v, v_1', v_2, v_2' of appropriate type representing the values of v in states s_1, s_1', s_2, s_2'. There are some program variables that make only sense in pre-states, e.g., `this`, and variables that make only sense in post-state, e.g., `result`. There will be only two logical variables that supply values to them instead of four. This leads to the following schematic form of ϕ_{α,R_1,R_2}:

$$\begin{aligned}&\phi_{\alpha,R_1,R_2} \quad \equiv \forall Heap\ h_1', h_2, h_2' \forall To' \forall T_r r, r' \forall \ldots v_1', v_2, v_2' \ldots \\ &\ (Agree_{pre} \wedge \langle\alpha\rangle\, \mathrm{sv}\{s_2\} \wedge \{\text{in } s_1'\}\, \langle\alpha\rangle\, \mathrm{sv}\{s_2'\} \rightarrow \{\text{in } s_2\}\{\text{in } s_2'\}(Agree_{post} \wedge Ext))\end{aligned}$$

To maintain readability we have used suggestive abbreviations: (1) $\{\text{in } s_1'\}\,\langle\alpha\rangle$ signals that an update $\{\texttt{heap} := h_1' \ \|\ \texttt{this} := o' \ \|\ \ldots a_i := v_1' \ldots\}$ is placed before the modal operator. The a_i cover all relevant parameters and local variables. (2) The construct $\mathrm{sv}\{s_2\}$ abbreviates a conjunction of equations $h_2 \doteq \texttt{heap}$, $r \doteq \texttt{result}$, $\ldots$, $v_2 \doteq a_i$, $\ldots$. (3) Analogously, $\mathrm{sv}\{s_2'\}$ stands for the primed version $h_2' \doteq \texttt{heap}$, $r' \doteq \texttt{result}$, $\ldots$, $v_2' \doteq a_i$, $\ldots$. (4) The shorthand $\{\text{in } s_2\}\{\text{in } s_2'\}E$ in front of a formula is resolved by (a) prefixing every occurrence of a heap-dependent term e with the update $\{\texttt{heap} := h_2\}$ and (b) every primed term e' with $\{\texttt{heap} := h_2'\}$. (5) The same applies to $\{\text{in } s_1'\}E$. Note that there is no $\{\text{in } s_1\}$, and no quantified variables o, v since the whole formula ϕ_{α,R_1,R_2} is evaluated in state s_1.

Furthermore we use the notation $(R_i^1)'$, R_i^2, $(R_i^2)'$ for the expressions obtained from R_i by replacing each state dependent designator v by v_1', v_2, v_2' respectively. Technically, these substitutions are effected by prefixing R_i with an appropriate update. For short we use $R[i]$ instead of $seqGet_{Any}(r, i)$, $t \sqsubseteq A$ for $instance_A(t)$, and $eInst_A$ for $exactInstance_A$.

We now supply the definitions of the abbreviations used above:

$$\begin{aligned}Agree_{pre} &\equiv R_1 \doteq (R_1^1)' \\ Agree_{post} &\equiv Agree_{type\&prim}(R_2^2, (R_2^2)') \wedge Agree_{obj}(R_2^2, R_2^2, (R_2^2)', (R_2^2)') \\ Ext &\equiv Agree_{obj}(R_1, R_2^2, (R_1^1)', (R_2^2)')\end{aligned}$$

These definitions make use of the predicates $Agree_{type\&prim}$, $Agree_{obj}$ and $Agree_{obj}^2$ which are recursively defined as

$$\begin{aligned}&Agree_{type\&prim}(Seq\ X, Seq\ X') \equiv \\ &\quad X.\mathbf{len} \doteq X'.\mathbf{len} \wedge \forall i(0 \le i < X.\mathbf{len} \rightarrow \\ &\qquad\qquad \textstyle\bigwedge_{A \text{ in } \alpha}(eInst_A(X[i]) \leftrightarrow eInst_A(X'[i])) \\ &\qquad\qquad \wedge (X[i] \not\sqsubseteq Obj \wedge X[i] \not\sqsubseteq Seq \rightarrow X[i] \doteq X'[i]) \\ &\qquad\qquad \wedge (X[i] \sqsubseteq Seq \rightarrow Agree_{type\&prim}(X[i], X'[i])))\end{aligned}$$

$$
\begin{aligned}
&\mathit{Agree}_{obj}(\mathit{Seq}\ X, \mathit{Seq}\ Y, \mathit{Seq}\ X', \mathit{Seq}\ Y') \equiv \\
&\quad \forall i(0 \leq i < Y.\mathbf{len} \rightarrow\ (Y[i] \sqsubseteq \mathit{Obj} \rightarrow \mathit{Agree}^2_{obj}(X, Y[i], X', Y'[i])) \\
&\qquad\qquad \wedge\ (Y[i] \sqsubseteq \mathit{Seq} \rightarrow \mathit{Agree}_{obj}(X, Y[i], X', Y'[i]))) \\
&\mathit{Agree}^2_{obj}(\mathit{Seq}\ X, \mathit{Obj}\ y, \mathit{Seq}\ X', \mathit{Obj}\ y') \equiv \\
&\quad \forall i(0 \leq i < X.\mathbf{len} \rightarrow\ (X[i] \sqsubseteq \mathit{Obj} \rightarrow (X[i] \doteq y \leftrightarrow X'[i] \doteq y')) \\
&\qquad\qquad \wedge\ (X[i] \sqsubseteq \mathit{Seq} \rightarrow \mathit{Agree}^2_{obj}(X[i], y, X'[i], y')))
\end{aligned}
$$

In many cases these definitions are much simpler. Frequently it is the case that $R_i.\mathbf{length}$ is not state dependent, then quantification over index i reduces to a disjunction of fixed length. Also the exact type of an expression can often be checked syntactically and needs not be part of the formula. In other cases however, e.g., if R_i is a variable of type *Seq*, the full definition is necessary.

Reconsider method `m1()` from Fig. 2 on page 23. Let $R = \langle C.x,\ C.y\rangle$. Then, $(R.\mathbf{len})^s = 2$ for all states s and the exact type of both fields x, y is always C. Thus Agree_{pre} equals $x \doteq x'_1 \wedge y \doteq y'_1$. Agree_{post} equals $x_2 \doteq y_2 \leftrightarrow x'_2 \doteq y'_2$. The complete formula $\phi_{m3(),R,R}$ is (after some simplification)

$$
\begin{aligned}
&\phi_{m3(),R,R} \equiv \\
&\forall \mathit{Heap}\ h'_1, h_2, h'_2 \forall C\ o' \forall x'_1, x_2, x'_2, y'_1, y_2, y'_2((x \doteq x'_1 \wedge y \doteq y'_1\ \wedge \\
&\langle m3()\rangle\,(x_2 \doteq x \wedge y_2 \doteq y)\ \wedge\ \{x := x'_1, y := y'_1\}\,\langle m3()\rangle\,(x'_2 \doteq x \wedge y'_2 \doteq y)) \\
&\rightarrow \\
&(x_2 \doteq y_2 \leftrightarrow x'_2 \doteq y'_2\ \wedge\ x \doteq x_2 \rightarrow x'_1 \doteq x'_2\ \wedge\ y \doteq x_2 \rightarrow y'_1 \doteq x'_2 \wedge \\
&x \doteq y_2 \rightarrow x'_1 \doteq y'_2\ \wedge\ y \doteq y_2 \rightarrow y'_1 \doteq y'_2))
\end{aligned}
$$

7 An Efficient Compositional Criterion

Though flow* from Def. 5 already simplifies the formulation of verification conditions and consequently checking for flow, we want to present another information flow property, flow**, which is still simpler to check. flow** is a criterion for flow, i.e., a sufficient but not a necessary condition. Roughly speaking, the main difference between flow and flow** is that flow** admits the attacker to distinguish between newly created objects and objects which already existed in the pre-state. This property of flow** is responsible for its compositionality (Thm. 3), which is an indispensable prerequisite for *modular* verification of information-flow properties. On the face of it, flow** takes more words to explain than the original flow property, but it is easier to prove: the partial isomorphism only differs from the identity on new objects. This reduces the effort to verify flow** considerably if only few or no new objects are created. Also, there is no obligation that one isomorphism is an extension of another.

On the other hand, an additional observation expression N_2 has to be given which exactly names the new elements of the set of objects observable in the post-state. Further it has to be proven that N_2 exactly names the new elements. However, normally it is quite an easy task to prove whether an object is newly created or not. Additionally, if a newly created object is observable in the post-state by an observation expression R_2, then there has to be a term in R_2 which

evaluates to this object. Hence N_2 is normally an explicit subexpression of R_2 and can be named easily.

Definition 6 (The predicate flow****).** *Let N_2 be an observation expression such that all terms in N_2 are of object type. Let, furthermore, α be a program, R_1, R_2 observation expressions, and s_1 a state.*

The predicate $\text{flow}^{**}(s_1, \alpha, R_1, R_2, N_2)$ *is true iff, for all computation domains $\mathcal{D}$, $\mathcal{D}'$ and states s_1', s_2, s_2' such that $\mathcal{D} + s_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D} + s_2$ and $\mathcal{D}' + s_1' \overset{\alpha}{\rightsquigarrow} \mathcal{D}' + s_2'$, we have*
if $\text{agree}(R_1, s_1, s_1', id)$
then all objects in $Obj(N_2^{s_2})$ and $Obj(N_2^{s_2'})$ are new and
$\text{agree}(N_2, s_2, s_2', \pi)$ *for a partial isomorphism π and*
if $\text{agree}(N_2, s_2, s_2', id)$ *then* $\text{agree}(R_2, s_2, s_2', id)$

Theorem 2. *Let N_2 be an observation expression such that all expressions in N_2 are of object type. Let furthermore α be a program, R_1, R_2 observation expressions, and s_1 a state.*

1. $\text{flow}^{**}(s_1, \alpha, R_1, R_2, N_2) \Rightarrow \text{flow}(s_1, \alpha, R_1, R_2)$.
2. *If for all domains $\mathcal{D}$ such that $\mathcal{D} + s_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D} + s_2$ we have $Obj(N_2^{s_2}) = \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = ff\}$ and $\{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$*
then $\text{flow}(s_1, \alpha, R_1, R_2) \Rightarrow \text{flow}^{**}(s_1, \alpha, R_1, R_2, N_2)$.

For the proof of the theorem we need the following auxiliary lemma. It states that we always can find domains $\mathcal{D}_2$, $\mathcal{D}_2'$ and therefore *nextToCreate* functions such that in two runs of a program α, which are started in R equivalent states, the same new objects are chosen for those objects which are observable by R.

Lemma 5. *Let α be a program such that $\mathcal{D} + s_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D} + s_2$, $\mathcal{D}' + s_1' \overset{\alpha}{\rightsquigarrow} \mathcal{D}' + s_2'$,* $\text{agree}(R, s_1, s_1', id)$ *and* $\text{agree}(N, s_2, s_2', \pi)$ *hold true for observation expressions R and N. In addition we assume that all objects in $Obj(N^{s_2})$ and $Obj(N^{s_2'})$ are new.*
Then there are domains $\mathcal{D}_2$, $\mathcal{D}_2'$ and isomorphisms $\rho : \mathcal{D} \to \mathcal{D}_2$, $\rho' : \mathcal{D}' \to \mathcal{D}_2'$ such that α started in $\mathcal{D}_2 + s_1$ terminates in $\mathcal{D}_2 + \rho(s_2)$, α started in $\mathcal{D}_2' + s_1'$ terminates in $\mathcal{D}_2' + \rho'(s_2')$ and $\text{agree}(N, \rho(s_2), \rho'(s_2'), id)$ *and $\rho(o) = o$, $\rho'(o') = o'$ for all o existing in state s_1 and for all o' existing in state s_1'.*

We omit the proof of Lemma 5 and go for the proof of Thm. 2 instead.

Proof (Theorem 2).

Part 1: We assume $\text{flow}^{**}(s_1, \alpha, R_1, R_2, N_2)$ and show $\text{flow}^{*}(s_1, \alpha, R_1, R_2)$. To this end we fix states s_1', s_2, s_2' and domains $\mathcal{D}$, $\mathcal{D}'$ such that $\mathcal{D} + s_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D} + s_2$, $\mathcal{D}' + s_1' \overset{\alpha}{\rightsquigarrow} \mathcal{D}' + s_2'$ and $\text{agree}(R_1, s_1, s_1', id)$. We need to show $\text{agree}(R_2, s_2, s_2', \pi)$, where the uniquely determined partial isomorphism π is compatible with id.

By assumption we obtain $\text{agree}(N_2, s_2, s_2', \sigma)$ and we know that all objects in $Obj(N_2^{s_2})$ and $Obj(N_2^{s_2'})$ are new. By Lemma 5 there are domains $\mathcal{D}_2$, $\mathcal{D}_2'$ and

isomorphisms $\rho : \mathcal{D} \to \mathcal{D}_2$, $\rho' : \mathcal{D}' \to \mathcal{D}'_2$ such that α started in $\mathcal{D}_2 + s_1$ terminates in $\mathcal{D}_2 + \rho(s_2)$, α started in $\mathcal{D}'_2 + s'_1$ terminates in $\mathcal{D}'_2 + \rho'(s'_2)$, and agree$(N_2, \rho(s_2), \rho'(s'_2), id)$. This enables us to use flow$^{**}(s_1, \alpha, R_1, R_2, N_2)$ again, now for the domains $\mathcal{D}_2$, $\mathcal{D}'_2$ in place of $\mathcal{D}$, $\mathcal{D}'$ to obtain agree$(R_2, \rho(s_2), \rho'(s'_2), id)$. Another appeal to Lemma 3 yields agree$(R_2, s_2, s'_2, \rho' \circ \rho^{-1})$. For $o \in Obj(R_1^{s_1}) \cap Obj(R_2^{s_2})$ we have $\rho' \circ \rho^{-1}(o) = o$, thus $\rho' \circ \rho^{-1}$ is compatible with id and the claim is proved.

Part 2: For the reverse implication we assume flow(s_1, α, R_1, R_2).

For the proof of flow$^{**}(s_1, \alpha, R_1, R_2, N_2)$ we consider states s'_1, s_2, s'_2 and domains $\mathcal{D}, \mathcal{D}'$ such that $\mathcal{D} + s_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D} + s_2, \mathcal{D}' + s'_1 \overset{\alpha}{\rightsquigarrow} \mathcal{D}' + s'_2$ and agree(R_1, s_1, s'_1, id). From flow(s_1, α, R_1, R_2) we obtain agree(R_2, s_2, s'_2, π) for π compatible with id. By case assumption we know $Obj(N_2^{s_2}) = \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = \mathit{ff}\}$. We see that π is a partial isomorphism from $Obj(N_2^{s_2})$ onto $Obj(N_2^{s'_2})$. This already gives us agree(N_2, s_2, s'_2, π). We assume agree(N_2, s_2, s'_2, id) to verify the remaining part of flow$^{**}(s_1, \alpha, R_1, R_2, N_2)$ with the intention to show agree(R_2, s_2, s'_2, id).

By agree(R_2, s_2, s'_2, π) we already know $\pi_{Seq}(R_2^{s_2}) = R_2^{s'_2}$, where π_{Seq} is defined as $\pi_{Seq}(\langle e_1, \ldots, e_k \rangle) = \langle e'_1, \ldots, e'_k \rangle$ with $e'_i = \pi(e_i)$ if $e_i \in Obj^{\mathcal{D}}$, $e'_i = \pi_{Seq}(e_i)$ if $e_i \in Seq^{\mathcal{D}}$ and $e'_i = e_i$ else. It remains to be shown that $\pi(e_i) = e_i$ for $e_i \in Obj(R_2^{s_2})$. We distinguish two cases: (1) $created^{s_1}(e_i) = tt$ and (2) $created^{s_1}(e_i) = \mathit{ff}$.

In case (1) we obtain $e_i \in Obj(R_1^{s_1})$ by the assumption $\{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$. Hence $\pi(e_i) = e_i$ since π is compatible with id. In case (2) use assumptions agree(N_2, s_2, s'_2, id) and $Obj(N_2^{s_2}) = \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = \mathit{ff}\}$, and also arrive at $\pi(e_i) = e_i$. □

The next lemma shows that the verification condition for flow** normally is much simpler than the one for flow*.

Lemma 6. *Let α be a program, let R_1, R_2, N_2 be observation expressions.*

Then there is a JavaDL *formula $\phi_{\alpha,R_1,R_2,N_2}$ such that for all states s_1* flow$^{**}(s_1, \alpha, R_1, R_2, N_2) \quad \Leftrightarrow \quad s_1 \models \phi_{\alpha,R_1,R_2,N_2}$.

Proof. The desired formula follows a pattern similar to the one in Thm. 1.

$$\begin{aligned}\phi_{\alpha,R_1,R_2,N_2} \quad \equiv \quad & \forall Heap\ h'_1, h_2, h'_2 \forall To' \forall T_r r, r' \forall \ldots v'_1, v_2, v'_2 \ldots \\ & (R_1 \doteq (R_1^1)' \wedge \langle\alpha\rangle \,\mathrm{save}\{s_2\} \wedge \mathrm{in}\{s'_1\}\, \langle\alpha\rangle \,\mathrm{save}\{s'_2\} \\ & \rightarrow \{\mathrm{in}\ s_2\}\{\mathrm{in}\ s'_2\}(newIso \wedge (N_2^2 \doteq (N_2^2)' \rightarrow R_2^2 \doteq (R_2^2)')))\end{aligned}$$

The abbreviations used above are defined as follows:

$$\begin{aligned}newIso \equiv\ & newOn(\mathtt{heap}, N_2^2) \wedge newOn(h'_1, (N_2^2)') \wedge Agree_{type}(N_2^2, (N_2^2)') \wedge \\ & Agree_{obj}(N_2^2, N_2^2, (N_2^2)', (N_2^2)') \\ newOn(Heap\ h, Seq\ X) \equiv\ & \\ \forall i(0 \leq i < X.\mathbf{len} \rightarrow\ & (X[i] \sqsubseteq Obj \rightarrow select(h, X[i], created) \doteq FALSE) \\ & \wedge\ (X[i] \sqsubseteq Seq \rightarrow newOn(h, X[i])))\end{aligned}$$

$$Agree_{type}(Seq\ X, Seq\ X') \equiv$$

$$X.\mathbf{len} \doteq X'.\mathbf{len} \wedge \forall i(0 \leq i < X.\mathbf{len} \rightarrow \bigwedge_{A \text{ in } \alpha}(eInst_A(X[i]) \leftrightarrow eInst_A(X'[i])) \wedge (X[i] \sqsubseteq Seq \rightarrow Agree_{type}(X[i], X'[i])))$$

$Agree_{obj}$ as in Thm. 1. We skip the rest of the proof, since it greatly parallels the one given for Thm. 1. □

We now show the compositionality of flow**. To this end we need to prove that flow** implies that the set of objects, which can be observed by an attacker in the post-state, contains only objects which are newly created or which already have been observed in the pre-state.

Lemma 7. *Let* s_1, s_1', s_2, s_2' *be states such that* $s_1 \overset{\alpha}{\rightsquigarrow} s_2$ *and* $s_1' \overset{\alpha}{\rightsquigarrow} s_2'$.
$\text{flow}^{**}(s_1, \alpha, R_1, R_2, N_2)$ *implies* $\text{flow}(s_1, \alpha, R_1, R_2)$ *and* $\text{agree}(R_1, s_1, s_1') \Rightarrow \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$.

Lemma 7 in combination with Thm. 2 gives an almost complete characterization of flow**. Indeed we can show that $\text{flow}^{**}(s_1, \alpha, R_1, R_2, N_2)$ also implies $\text{agree}(R_1, s_1, s_1') \Rightarrow \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = ff\} \subseteq Obj(N_2^{s_2})$ which makes this characterization tight. This characterization shows in particular that the main difference between flow and flow** is that, roughly speaking, flow** admits the attacker to distinguish between newly created objects and objects which already existed in the pre-state. This property of flow** is responsible for its compositionality:

Theorem 3 (Compositionality of flow****).** *Let* s_1, s_1', s_2, s_2', s_3, s_3' *be states such that* $s_1 \overset{\alpha_1}{\rightsquigarrow} s_2$, $s_2 \overset{\alpha_2}{\rightsquigarrow} s_3$, $s_1' \overset{\alpha_1}{\rightsquigarrow} s_2'$ *and* $s_2' \overset{\alpha_2}{\rightsquigarrow} s_3'$. *If*

1. $\text{flow}(s_1, \alpha_1, R_1, R_2)$,
2. $\text{flow}(s_2, \alpha_2, R_2, R_3)$,
3. $\text{agree}(R_1, s_1, s_1') \Rightarrow \{o \in Obj(R_2^{s_2}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$ *and*
4. $\text{agree}(R_2, s_2, s_2') \Rightarrow \{o \in Obj(R_3^{s_3}) \mid created^{s_2}(o) = tt\} \subseteq Obj(R_2^{s_2})$

then
$\text{flow}(s_1, \alpha_1; \alpha_2, R_1, R_3)$ *and* $\text{agree}(R_1, s_1, s_1') \Rightarrow \{o \in Obj(R_3^{s_3}) \mid created^{s_1}(o) = tt\} \subseteq Obj(R_1^{s_1})$.

We omit the proofs of Lemma 7 and Thm. 3 for the sake of brevity, but they can be found in the companion technical report [7].

8 Related Work

There exists a very large body of work on language-based security. Besides the discussion below, we refer to [30] for a survey.

Security type systems are one of the most popular approaches. A prominent example in this field is the JIF system [26]. Type system approaches are efficient, but sometimes also quite imprecise. A further approach is checking the

dependence graph of a program for graph-theoretical reachability properties [16]. Though this technique is substantially different from type system approaches, it is efficient and sometimes quite imprecise, too. Further approaches use abstraction and ghost code for explicit tracking of dependencies [10]. They are quite near in spirit to flow-sensitive security type systems, but have not tackled the problem of modular verification yet. All approaches mentioned so far appear to be limited to information flow between variables and it is questionable whether they can be adopted to fine-grained specifications as the one introduced in this paper.

The most popular approach in logic based information flow analysis is stating secure information flow with the help of self-composition [5,12] and using off-the-shelf software verification systems to check for it, as we do. The approach has the appealing feature that it can be arbitrarily precise as long as the used verification system has a relatively complete calculus. An important alternative in logic based information flow analysis is the usage of specialized, approximate calculi [1]. Finally, secure information flow can be formalized in higher-order logic, and higher-order theorem provers like Coq can be used for checking secure information flow [27]. This approach seems to be very expressive, but comes at the price of more and more complex interactions with the proof system.

Focusing on *object-sensitive* secure information flow, the paper closest to ours is [1]. The authors build on *region logic*, a kind of Hoare logic with concepts from separation logic, which is comparable to JavaDL. They use the same basic definition of object-sensitive secure information flow. Instead of providing verification conditions which can be discharged with a standard calculus, as we do, they introduce a specialized, more efficient calculus to show object-sensitive secure information flow. This specialized calculus uses approximate rules which avoid explicit modeling of isomorphisms, but comes with the price of imprecision. The discerning points of our work are: (1) a further investigation of the security property, allowing the restriction of isomorphisms as far as possible and thus making the explicit, non approximate modeling of isomorphisms feasible with a minimum of additional user interaction; (2) verification conditions that are discharged with an existing tool; and (3) a more flexible specification methodology.

Contributions (1) and (3) also distinguish this work from the other approaches mentioned above, including JIF, which already presented an approximative treatment of object-sensitive secure information flow for Java [26]. JIF is a practical approach to the analysis of secure information flow which covers a broad range of language features, but it has not been formally proven to enforce non-interference. Similar to JIF, [3,6] use type systems for the verification of object-oriented secure information flow. They treat a smaller set of language features, but prove that their type systems indeed enforce non-interference. A closely related approach is [9]. Here, only the information flow analysis is based on type systems; the verification task is separated from the analysis and based on program logics. Still, points (1) and (3) as well as the overall precision are discerning points of this paper. The approach in [6], already mentioned above, and the approaches [17,18] target Java Bytecode in contrast to source code, as the other approaches do. The latter is

a type system approach, too, whereas the former uses abstract interpretation in combination with classical static analysis.

To the best of our knowledge, the only approach which models isomorphisms explicitly is the self-composition approach [28]. The drawback of that approach is that the specifier needs to track the isomorphism manually with the help of additional ghost code annotations. This increases the burden on the specifier, whereas our approach detects the isomorphism automatically.

Focusing on fine-grained information flow specifications, the approaches closest to ours are [4,31]. These approaches specify information flow between variables and fields only, but allow for the declassification of terms. Our approach generalizes and unifies these approaches. This generalization already proved to be useful in a recent case-study, see Sect. 3.

9 Conclusions and Future Work

Lemma 4 and Thm. 2 prove the relation between standard object-sensitive noninterference and our improved versions. These results lead to an approach to verify object-sensitive non-interference properties of Java programs by a direct translation into Dynamic Logic (Thm. 1 and Lemma 6). The approach has been implemented in the KeY tool and successfully tested on small examples. The implementation can be tested on our web page using Java Web Start. In particular, we have successfully treated the examples included in this paper, as well as the (somewhat more involved) examples by Naumann [28]. Application to a larger e-voting case study is currently underway.

In a future paper we plan to present a complementary specialized calculus for the flow predicate intended to further increase reasoning efficiency. As proved in Thm. 3, the flow** criterion is compositional and is expected to lead to a particularly efficient calculus. A specification interface to the Java Modeling Language (JML) [22] for information flow properties has been published in [31].

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In Proceedings POPL, pp. 91–102. ACM (2006)
2. Amtoft, T., Banerjee, A.: Information Flow Analysis in Logical Form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer, Heidelberg (2004)
3. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a Java-like language. In: Proceedings CSFW (2002)
4. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: IEEE Symposium on Security and Privacy. SP 2008, pp. 339–353. IEEE (2008)
5. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. CSFW 2004, pp. 100–115. IEEE CS, Washington, USA (2004)

6. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference Java bytecode verifier. Mathematical Structures in Comp. Sci., FirstView:1–50, 4 (2013)
7. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software : Extended version. Technical Report 2013-14, KIT (2013)
8. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software. LNCS, vol. 4334. Springer, Heidelberg (2007)
9. Beringer, L., Hofmann, M.: Secure information flow and program logics. In: CSF, pp. 233–248 (2007)
10. Bubel, R., Hähnle, R., Weiß, B.: Abstract Interpretation of Symbolic Execution with Explicit State Updates. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 247–277. Springer, Heidelberg (2009)
11. Cohen, E.S.: Information transmission in computational systems. In: SOSP, pp. 133–139 (1977)
12. Darvas, A., Hähnle, R., Sands, D.: A Theorem Proving Approach to Analysis of Secure Information Flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
13. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
14. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
15. Greiner, S., Birnstill, P., Krempel, E., Beckert, B., Beyerer, J.: Privacy preserving surveillance and the tracking paradox. In: Proceedings, Future Security Conference 2013, Berlin (2013). (To appear September 15–19, 2013)
16. Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: ISSSE, pp. 87–96. IEEE (March 2006)
17. Hansen, R.R., Probst, C.W.: Non-interference and erasure policies for Java Card bytecode. In: WITS (2006)
18. Hedin, D., Sands, D.: Timing aware information flow security for a JavaCard-like bytecode. In: BYTECODE, vol. 141:1 of ENTCS, pp. 163–182 (2005)
19. Hedin, D., Sands, D.: Noninterference in the presence of non-opaque pointers. In: CSFW, pp. 217–229. IEEE Computer Society (2006)
20. Joshi, R., Leino, K.R.M.: A semantic approach to secure information flow. Science of Computer Programming **37**(1–3), 113–138 (2000)
21. Lampson, B.W.: A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)
22. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a Java Modeling Language. In: Formal Underpinnings of Java Workshop (at OOPSLA 1998) (October 1998)
23. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman Publishing Co. Inc., Boston (1999)
24. McCarthy, J.: Towards a mathematical science of computation. Information Processing, pp. 21–28 (1962)
25. Mitchell, J.C.: Type systems for programming languages. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics, pp. 365–458 (1990)
26. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: POPL, pp. 228–241 (1999)
27. Nanevski, A., Banerjee, A., Garg, D.: Verification of information flow and access control policies with dependent types. In: SP, pp. 165–179 (2011)

28. Naumann, D.A.: From Coupling Relations to Mated Invariants for Checking Information Flow. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 279–296. Springer, Heidelberg (2006)
29. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2. Tr, U. of Iowa (2006)
30. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
31. Scheben, C., Schmitt, P.H.: Verification of Information Flow Properties of Java Programs without Approximations. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 232–249. Springer, Heidelberg (2012)
32. Sun, Q., Banerjee, A., Naumann, D.A.: Modular and Constraint-Based Information Flow Inference for an Object-Oriented Language. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 84–99. Springer, Heidelberg (2004)
33. Weiß, B.: Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction. PhD thesis, KIT (2011)

http://www.springer.com/978-3-319-14124-4

Logic-Based Program Synthesis and Transformation
23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers
Gupta, G.; Peña, R. (Eds.)
2014, XII, 237 p. 51 illus., Softcover
ISBN: 978-3-319-14124-4