Information Flow Analysis Based on Program Simplification

Ran Ji and Reiner Hähnle

Department of Computer Science
Technische Universität Darmstadt, Germany
{ran,haehnle}@cs.tu-darmstadt.de

Abstract. Deductive verification is a popular approach to language-based information flow analysis, however, the existing methods need non-standard verification setups that hamper the prospects for automation. We propose a uniform framework, wherein information flow analysis is realized by deductive verification of a single, unmodified program with lightweight postconditions and invariants. We perform symbolic execution-based verification, during which sound program transformation generates a simplified program being bisimilar to the target program with respect to low variables. The process maintains a sound used variable set that indicates whether the resulting program is secure.

1 Introduction

Language-based information flow security analysis is an important and popular research problem [1]. Here we consider *static checking* of *security policies*, whose baseline is *non-interference* [2,3]: a variation of `High` (confidential) input does not cause a variation of `Low` (public) output. Equivalently, the values of `Low` output does not depend on the `High` input.

Despite considerable effort, a fully satisfying solution to static checking of security policies has been exclusive. *Security type systems* [4,5] track the confidentiality level (`High`/`Low`) of information contained in variables and program context, and over-approximate information flows occurring in (an over-approximation of) the possible control flow paths. Together with the value-insensitivity of type-based analyses, this results in a loss of precision in many situations.

Deductive verification has been suggested in [6] and offers high precision, but comes at the price of expert user interaction with a verification system. The reason for this is that information flow is a *relational property*. Checking it requires to compare different runs of a program with each other. Known approaches are to analyze the same program twice in a sequential manner (termed *self-composition* in [7]), or in a parallel manner [8], or to use additional quantification over the input variables [6]. In either case, precise postconditions and accordingly strong invariants are required, rendering automation problematic. Preprocessing of a program into a *product program* [9] was suggested as an improvement, but still complicates the target program in many cases.

In this paper, we propose a uniform framework, wherein information flow analysis is realized by deductive verification of a single, unmodified program with lightweight postconditions and invariants. In a first phase we perform symbolic execution-based verification, interleaved with partial evaluation (constant propagation, dead code elimination, etc.) [10]. In a second phase we perform bottom-up traversal of the symbolic execution tree to synthesize a program that is *weakly bisimilar* to the original with respect to a set of *observable* locations (i.e., the `Low` variables). This builds on earlier work on sound program transformation [11]. During synthesis we maintain a *used variable set* that may affect the values of the observable locations. Whenever no `High` variables occur in the used variable set, we can conclude that the non-interference policy is enforced. Otherwise, deductive verification [6,7,9] can be used on the *simplified* program, which still is a vast improvement, because all unused variables have been removed from it. We show that our approach is more precise than security type systems. At the same time, it is easier to automate, because only lightweight invariants and postconditions are needed.

The paper is organized as follows: Sect. 2 defines the programming language and program logic; Sect. 3 presents the sequent calculus rules used for symbolic execution; Sect. 4 introduces a bisimulation modality and extended suquent calculus rules used for program generation; Sect. 5 shows the information flow security enforcement; Sect. 6 draws the conclusion and discusses related work and future work.

2 Language and Logic

2.1 Programming Language

To keep the formalism manageable we work with a non-trivial subset of Java called PL that supports classes, objects, attributes, method polymorphism (but not method overloading). Generic types, exceptions, multi-threading, floating points, and garbage collection are not supported. The types of PL are the types derived from class declarations, the type `int` of mathematical integers ($\mathbb{Z}$), and the standard Boolean type `boolean`.

A PL program `p` is a non-empty set of class declarations with at least one class of name `Object`. The class hierarchy is a tree with class `Object` as root. A class $Cl := (cname, scname_{opt}, fld, mtd)$ consists of (i) a classname *cname* unique in `p`, (ii) the name of its superclass *scname* (only omitted for $cname =$ `Object`), and (iii) a list of field *fld* and method *mtd* declarations. The syntax coincides with that of Java. The only features lacking from Java are constructors and initialization blocks. We agree on the following conventions: if not stated otherwise, any sequence of statements is viewed as if it were the body of a static, void method declared in a class `Default` with no fields.

In a PL program `p`, a complex statement can be decomposed into a sequence of simpler statements without changing the meaning of `p`. For example, statement `y = z ++;` can be decomposed into `int t = z; z = z + 1; y = t;`, where `t` is a *fresh* variable, not used anywhere else. These *simple statements* have at most one source of side effect each, which can be a non-terminating expression (such as a null pointer access), a method call, or an assignment to a location. They are the essential to compute variable dependencies and simplify symbolic states during symbolic execution.

2.2 Program Logic

Our program logic is *dynamic logic* (DL) [12]. We consider deterministic programs, hence, a program `p` executed in state s *either* terminates and reaches exactly *one* final state *or* it does not terminate and no final state reached. A dynamic logic for PL-programs is called PL-DL. The signature of the program logic depends on a *context* PL *program* $\mathcal{C}$.

Definition 1 (PL-Signature $\Sigma_\mathcal{C}$). *A* signature $\Sigma_\mathcal{C} = (\mathsf{Srt}, \preceq, \mathsf{Pred}, \mathsf{Func}, \mathsf{LgV})$ *consists of: (i) a set of names* Srt *called* sorts *containing at least one sort for each primitive type and one for each class* Cl *declared in* $\mathcal{C}$*:* $\mathsf{Srt} \supseteq \{\mathtt{int}, \mathtt{boolean}\} \cup \{Cl \mid \textit{for all classes } Cl \textit{ declared in } \mathcal{C}\}$*; (ii) a* partial subtyping order $\preceq : \mathsf{Srt} \times \mathsf{Srt}$ *that models the subtype hierarchy of* $\mathcal{C}$ *faithfully; (iii) infinite sets of* predicate symbols $\mathsf{Pred} := \{p : T_1 \times \ldots \times T_n \mid T_i \in \mathsf{Srt}\}$ *and* function symbols $\mathsf{Func} := \{f : T_1 \times \ldots \times T_n \to\ T \mid T_i, T \in \mathsf{Srt}\}$ *for each* $n \in \mathbb{N}$*. We call* $\alpha(p) = T_1 \times \ldots \times T_n$ *and* $\alpha(f) = T_1 \times \ldots \times T_n \to\ T$ *the* signature *of the predicate/function symbol.* $\mathsf{Func} := \mathsf{Func}_r \cup \mathsf{PV} \cup \mathsf{Attr}$ *is further divided into disjoint subsets:*

- *the* rigid *function symbols* Func_r*, which do not depend on the current state of program execution;*
- *the* program variables $\mathsf{PV} = \{\mathtt{i}, \mathtt{j}, \ldots\}$*, which are non-rigid constants;*
- *the* attribute *function symbols* Attr*, such that for each attribute* `a` *of type* T *declared in class* Cl *an attribute function* $\mathtt{a}@Cl : Cl \to T \in \mathsf{Attr}$ *exists. We omit the* $@C$ *from attribute names if no ambiguity arises.*

(iv) a set of logical variables $\mathsf{LgV} := \{x : T \,|\, T \in \mathsf{Srt}\}$.

We distinguish between *rigid* and *non-rigid* predicate and function symbols. Intuitively, the semantics of rigid symbols does not depend on the current state of program execution, while non-rigid symbols are state-dependent.

Terms t and formulas ϕ are defined as usual, thus their definitions are omitted here for brevity. We use *updates* u to describe state changes by means of an explicit substitution. An *elementary update* $\mathtt{i} := t$ or $t.\mathtt{a} := t$ is a pair of location and term. They are of *static single assignment (SSA)* form, with the same meaning as simple assignments. Elementary updates are composed to *parallel updates* $u_1 \| u_2$ and work like simultaneous assignments. Updates u are defined by the grammar $u ::= \mathtt{i} := t \mid t.\mathtt{a} := t \mid u \parallel u \mid \{u\}u$ (where $\mathtt{a} \in \mathsf{Attr}$) together with the usual well-typedness conditions. Updates applied on terms (formulas), written $\{u\}t$ ($\{u\}\phi$), are again terms (formulas). Terms, formulas and updates are evaluated wrt a PL-DL Kripke structure:

Definition 2 (Kripke structure). *A PL-DL Kripke structure* $\mathcal{K}_{\Sigma_{\mathrm{PL}}} = (\mathcal{D}, I, S)$ *consists of (i) a set of elements* $\mathcal{D}$ *called domain, (ii) an interpretation* I *with*

– *$I(T) = \mathcal{D}_T$, $T \in \mathsf{Srt}$ assigning each sort its non-empty domain $\mathcal{D}_T$. It adheres to the restrictions imposed by the subtype order $\preceq$; `Null` is always interpreted as a singleton set and subtype of all class types;*
– *$I(f) : \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n} \to \mathcal{D}_T$ for each rigid function symbol $f : T_1 \times \ldots \times T_n \to T \in \mathsf{Func}_r$;*
– *$I(p) \subseteq \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n}$ for each predicate symbol $p : T_1 \times \ldots \times\ T_n \in \mathsf{Pred}$;*

and (iii) a set of states S assigning meaning to non-rigid function symbols: let $s \in S$ then $s(\mathsf{a}@Cl) : \mathcal{D}_{Cl} \to \mathcal{D}_T,\ \mathsf{a}@Cl : Cl \to T \in \mathsf{Attr}$ and $s(\mathtt{i}) : \mathcal{D}_T,\ \mathtt{i} \in \mathsf{PV}$. The pair $D = (\mathcal{D}, I)$ is called a first-order structure.

A *variable assignment* $\beta : \mathsf{LgV} \to \mathcal{D}_T$ maps a logical variable $x : T$ to its domain $\mathcal{D}_T$. A term, formula or update is evaluated relative to a given first-order structure $D = (\mathcal{D}, I)$, a state $s \in S$ and a variable assignment β, while programs and expressions are evaluated relative to a D and $s \in S$. The *evaluation function val* is defined recursively. It evaluates: (i) every term $t : T$ to a value $val_{D,s,\beta}(t) \in \mathcal{D}_T$; (ii) every formula ϕ to a truth value $val_{D,s,\beta}(\phi) \in \{tt, ff\}$; (iii) every update u to a state transformer $val_{D,s,\beta}(u) \in S \to S$; (iv) every expression $e : T$ to a set of pairs of state and value $val_{D,s}(e) \subseteq 2^{S \times T}$; (v) every statement st to a set of states $val_{D,s}(st) \subseteq 2^S$.

As PL is deterministic, all sets of states or state-value pairs have at most one element. The semantics definition of terms, formulas, expressions and statements are the same as in Java. More details can also be found in [13].

Example 1 (Update semantics). Evaluating $\{\mathtt{i} := \mathtt{j} + 1\}\mathtt{i} \geq \mathtt{j}$ in a state s is identical to evaluating the formula $\mathtt{i} \geq \mathtt{j}$ in a state s' which coincides with s except for the value of $\mathtt{i}$ which is evaluated to the value of $val_{D,s,\beta}(\mathtt{j} + 1)$. Evaluation of the parallel update $\mathtt{i} := \mathtt{j} \| \mathtt{j} := \mathtt{i}$ in a state s leads to the successor state s' identical to s except that the values of $\mathtt{i}$ and $\mathtt{j}$ are swapped. The parallel update $\mathtt{i} := 3 \| \mathtt{i} := 4$ has a *conflict* as $\mathtt{i}$ is assigned different values. In such a case the last occurring assignment $\mathtt{i} := 4$ overrides all previous ones of the same location. Evaluation of $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}\phi$ in a state s results in evaluating ϕ in a state, where $\mathtt{i}$ has the value of $\mathtt{j}$, and $\mathtt{j}$ remains unchanged.

Remark 1. $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}\phi$ is the sequential application of updates $\mathtt{i} := \mathtt{j}$ and $\mathtt{j} := \mathtt{i}$ on the formula ϕ. To ease the presentation, we overload the concept of update and also call $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}$ an update. In the following, if not stated otherwise, we use the upper-case letter $\mathcal{U}$ to denote this kind of update, compared to the proper update denoted by a lower-case letter u. An update $\mathcal{U}$ may be the of form $\{u\}$ and $\{u_1\} \cdots \{u_n\}$. Furthermore, $\{u_1\} \cdots \{u_n\}$ can be simplified to the form $\{u\}$, called the *normal form* (NF) of an update.

Definition 3 (Normal form of update). *An update is in* normal form*, denoted by $\mathcal{U}^{nf}$, if it has the shape $\{u_1 \| \cdots \| u_n\}$, $n \geq 0$, where each u_i is an elementary update and there is no conflict between u_i and u_j for any $i \neq j$.*

The normal form of an update can be achieved by applying a sequence of *update simplification* steps [13]. Soundness of these rules and that they achieve normal form are proven in [14].

3 Symbolic Execution Based Program Verification

3.1 Sequent Calculus

We perform symbolic execution-based program verification following the KeY [15] approach. Symbolic execution of a PL-program is achieved by application of sequent calculus rules. Soundness of the rules ensures validity of provable PL-DL formulas in a program verification setting [15].

A *sequent* is a pair of sets of formulas $\Gamma = \{\phi_1, \ldots, \phi_n\}$ (antecedent) and $\Delta = \{\psi_1, \ldots, \psi_m\}$ (succedent) of the form $\Gamma \Longrightarrow \Delta$. Its semantics is defined by the formula $\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi$. A *sequent calculus rule* has one conclusion and zero or more premises. It is applied to a sequent s by matching its conclusion against s. The instantiated premises are then added as children of s.

Our PL-DL sequent calculus behaves as a symbolic interpreter for PL. A *sequent* for PL-DL is always of the form $\Gamma \Longrightarrow \mathcal{U}[\mathtt{p}]\phi, \Delta$. During symbolic execution performed by the sequent rules (see Fig. 1) the antecedent Γ accumulates path conditions and contains possible preconditions. The updates $\mathcal{U}$ record the current symbolic value and ϕ represents postconditions. When a program is fully executed, we obtain a set of first-order formulas (each for an execution path) which is to be proven, or disproven, by a first-order solver.

$$\mathsf{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U}\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[]\phi, \Delta} \qquad \mathsf{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{l} := \mathtt{r}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{l} = \mathtt{r};\omega]\phi, \Delta}$$

$$\mathsf{ifElse}\ \frac{\Gamma, \mathcal{U}\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{p};\omega]\phi, \Delta \qquad \Gamma, \mathcal{U}\neg\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{q};\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if\ (b)\ \{p\}\ else\ \{q\}}\ \omega]\phi, \Delta}$$

$$\mathsf{loopInvariant}\ \frac{\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}inv, \Delta & \text{(init)} \\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\mathtt{p}]inv, \Delta & \text{(preserves)} \\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\neg\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\omega]\phi, \Delta & \text{(use case)} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}}\ \omega]\phi, \Delta}$$

Fig. 1. Selected sequent calculus rules (for more details, see [13,15]).

During symbolic execution complex statements are decomposed into simple ones. First-order reasoning as well as interleaved partial evaluation [10] help to simplify the target program on-the-fly. Symbolic execution of works as follows:

1. Select an open proof goal with a $[\cdot]$ modality. If no $[\cdot]$ exists on any branch, then symbolic execution is completed. Focus on the first active statement (possibly empty) of the program in the modality.
2. If it is a complex statement, apply rules to decompose it into simple statements and goto 1., otherwise continue.
3. Apply the sequent calculus rule corresponding to the active statement.
4. Simplify the resulting updates and apply first-order simplification to the premises. This might result in some closed branches. It is possible to detect and eliminate infeasible paths in this way. Goto 1.

Example 2. We look at typical proof goals that arise during symbolic execution:

1. $\Gamma, \texttt{i} > \texttt{j} \Rightarrow \mathcal{U}[\texttt{if (i>j) \{p\} else \{q\}}\ \omega]\phi$: Applying rule ifElse and simplification eliminates the `else` branch and continues with $\texttt{p}\ \omega$.
2. $\Gamma \Rightarrow \{\texttt{i} := \texttt{c} \| \ldots\}[\texttt{j = i;}\ \omega]\phi$ where `c` is a constant: It is sound to replace the statement $\texttt{j} = \texttt{i}$ with $\texttt{j} = \texttt{c}$ and continue with symbolic execution. This is known as *constant propagation*. More techniques for *partial evaluation* can be integrated into symbolic execution [10].
3. $\Gamma \Rightarrow \{\texttt{o1.a} := \texttt{v1} \| \ldots\}[\texttt{o2.a = v2;}\ \omega]\phi$: After executing $\texttt{o2.a} = \texttt{v2}$, the *alias* is analyzed: (i) if $\texttt{o2} = \texttt{null}$ is true the program does not terminate; (ii) else, if $\texttt{o2} = \texttt{o1}$ holds, the value of `o1.a` in the update is overridden and the new update is $\{\texttt{o1.a} := \texttt{v2} \| \ldots \| \texttt{o2.a} := \texttt{v2}\}$; (iii) else the new update is $\{\texttt{o1.a} := \texttt{v1} \| \ldots \| \texttt{o2.a} := \texttt{v2}\}$. Neither of (i)–(iii) might be provable, then symbolic execution splits into these three cases.

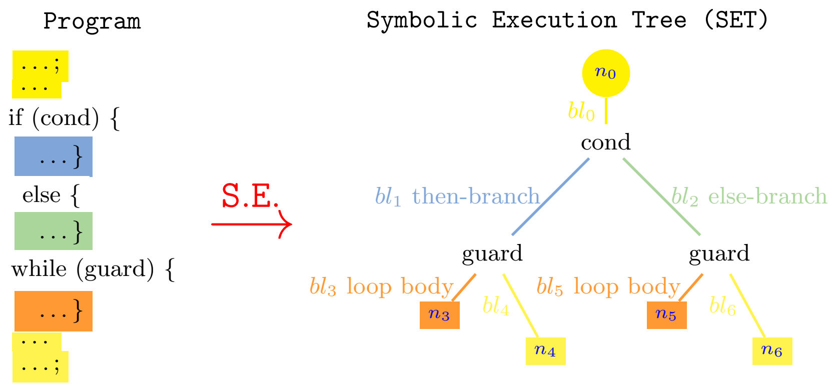

Fig. 2. Symbolic execution tree with loop invariant applied.

The result of symbolic execution for a PL program `p` following the sequent calculus rules is a *symbolic execution tree (SET)*, as illustrated in Fig. 2. Note that here we did not show the part that does not contain any PL program, e.g., the (init) branch obtained after applying the loopInvariant rule. Complete symbolic execution trees are finite trees whose root is labeled with $\Gamma \Longrightarrow [\texttt{p}]\phi, \Delta$ and no leaf has a $[\cdot]$ modality. We can assume that each inner node i is annotated by a sequent $\Gamma_i \Longrightarrow \mathcal{U}_i[\texttt{p}_\texttt{i}]\phi_i, \Delta_i$, where $\texttt{p}_\texttt{i}$ is the program to be executed. Every child node is generated by rule application from its parent. A *branching node* represents a statement whose execution causes branching, e.g., conditional, loops etc. We call a *sequential block (SB)* a maximal program fragment in an SET that is symbolically executed without branching. A sequential block bl_0 is a *child* of a sequential block bl_1 if bl_0 starts and bl_1 ends with the same branching node. The *descendant* relation is the transitive closure of the child relation. A *generalized sequential block (GSB)* is a sequential block together with all its descendants. GSBs always end with leaf nodes.

In the SET shown in Fig. 2, there are 7 sequential blocks $bl_0,\ldots,bl_6$, and bl_3 is the child of bl_1, and the descendant of bl_0. We have GSBs $\{bl_1, bl_3, bl_4\}$ and $\{bl_2, bl_5, bl_6\}$. For convenience, we refer to a GSB with the father sequential block. For instance, GSB $\{bl_1, bl_3, bl_4\}$ is denoted as GSB(bl_1). An SET is a GSB itself, which is GSB(bl_0) in Fig. 2.

4 Program Transformation

The structure of an SET makes it possible to generate a program by bottom-up traversal. The resulting program transformation is the core concept behind our information flow analysis. The idea is to apply sequent calculus rules reversely to generate a simplified program step-by-step. This requires to extend the sequent rules by means for program generation. Obviously, the generated program should behave exactly as the original one, at least for the *observable locations*.

4.1 Weak Bisimulation Relation of Programs

Definition 4 (Location sets, observation equivalence). *A* location set *is a set containing program variables* `x` *and attribute expressions* $o.\mathtt{a}$ *(*$\mathtt{a} \in \mathsf{Attr}$ *and* o *being a term of the appropriate sort). Let loc be the set of all program locations, given two states* s_1, s_2 *and a location set* $obs \subseteq loc$*. A relation* $\approx: loc \times S \times S$ *is an* observation equivalence *if and only if for all* $ol \in obs$, $val_{D,s_1,\beta}(ol) = val_{D,s_2,\beta}(ol)$ *holds. It is written as* $s_1 \approx_{obs} s_2$*. We call obs* observable locations*.*

A *transition relation* $\longrightarrow: \Pi \times S \times S$ relates two states s, s' by a program `p` iff `p` starts in state s and terminates in state s', written $s \xrightarrow{\mathtt{p}} s'$. We have: $s \xrightarrow{\mathtt{p}} s'$, where $s' = val_{D,s}(\mathtt{p})$. If `p` does not terminate, we write $s \xrightarrow{\mathtt{p}}$.

Since a complex statement can be decomposed into simple statements during symbolic execution, we can assume that a program consists of simple statements.

Definition 5 (Observable and internal statement/transition). *Consider states* s, s'*, a simple statement* `sSt`*, a transition relation* $\longrightarrow$*, where* $s \xrightarrow{\mathtt{sSt}} s'$*, and the observable locations obs; we call* `sSt` *an* observable statement *and* $\longrightarrow$ *an* observable transition*, if and only if there exists* $ol \in obs$*, and* $val_{D,s',\beta}(ol) \neq val_{D,s,\beta}(ol)$*. We write* $\xrightarrow{\mathtt{sSt}}_{obs}$*. Otherwise,* `sSt` *is called an* internal statement *and* $\longrightarrow$ *an* internal transition*, written* $\longrightarrow_{int}$*.*

Assume an observable transition $s \xrightarrow{\mathtt{sSt}}_{obs} s'$ changes the evaluation of some location $ol \in obs$ in state s'. The observable locations obs_1 in state s should also contain the locations ol_1 that are *read* by ol, since changes to ol_1 can lead to a change of ol in the final state s'.

Example 3. Consider the set of observable locations obs={`x`, `y`} and program fragment "`z = x + y; x = 1 + z;`". The statement `z = x + y;` becomes observable because the value of `z` is changed and it will be used later in the observable statement `x = 1 + z;`. The observable location set obs_1 should contain `z` after the execution of `z = x + y;` .

Definition 6 (Weak transition). *Given observable locations obs, the transition relation* $\Longrightarrow_{int}$ *is the reflexive, transitive closure of* $\longrightarrow_{int}$*. The transition relation* $\overset{\mathtt{sSt}}{\Longrightarrow}_{obs}$ *is the composition of the relations* $\Longrightarrow_{int}$*,* $\overset{\mathtt{sSt}}{\longrightarrow}_{obs}$ *and* $\Longrightarrow_{int}$*. The* weak transition $\overset{\widehat{\mathtt{sSt}}}{\Longrightarrow}_{obs}$ *represents either* $\overset{\mathtt{sSt}}{\Longrightarrow}_{obs}$*, if* $\mathtt{sSt}$ *observable, or* $\Longrightarrow_{int}$ *otherwise.*

Definition 7 (Weak bisimulation for states). *Given two programs* $\mathtt{p}_1, \mathtt{p}_2$ *and observable locations obs, obs′, let* $\mathtt{sSt}_1$ *be a simple statement and* s_1, s_1' *two program states of* $\mathtt{p}_1$*, and* $\mathtt{sSt}_2$ *is a simple statement and* s_2, s_2' *are two program states of* $\mathtt{p}_2$*. A relation* $\approx$ *is a* weak bisimulation *for states if and only if* $s_1 \approx_{obs} s_2$ *implies:*

- *if* $s_1 \overset{\widehat{\mathtt{sSt}_1}}{\Longrightarrow}_{obs'} s_1'$*, then* $s_2 \overset{\widehat{\mathtt{sSt}_2}}{\Longrightarrow}_{obs'} s_2'$ *and* $s_1' \approx_{obs'} s_2'$
- *if* $s_2 \overset{\widehat{\mathtt{sSt}_2}}{\Longrightarrow}_{obs'} s_2'$*, then* $s_1 \overset{\widehat{\mathtt{sSt}_1}}{\Longrightarrow}_{obs'} s_1'$ *and* $s_2' \approx_{obs'} s_1'$

where $val_{D,s_1}(\mathtt{sSt}_1) \approx_{obs'} val_{D,s_2}(\mathtt{sSt}_2)$*.*

Definition 8 (Weak bisimulation for programs). *Let* $\mathtt{p}_1, \mathtt{p}_2$ *be two programs, obs and obs′ are observable locations, and* $\approx$ *is a weak bisimulation relation for states.* $\approx$ *is a* weak bisimulation *for programs, written* $\mathtt{p}_1 \approx_{obs} \mathtt{p}_2$*, if for the sequence of state transitions:*

$$s_1 \xrightarrow{\mathtt{p}_1} s_1' \equiv s_1^0 \xrightarrow{\mathtt{sSt}_1^0} s_1^1 \xrightarrow{\mathtt{sSt}_1^1} \dots \xrightarrow{\mathtt{sSt}_1^{n-1}} s_1^n \xrightarrow{\mathtt{sSt}_1^n} s_1^{n+1}, \text{ with } s_1 = s_1^0,\ s_1' = s_1^{n+1},$$

$$s_2 \xrightarrow{\mathtt{p}_2} s_2' \equiv s_2^0 \xrightarrow{\mathtt{sSt}_2^0} s_2^1 \xrightarrow{\mathtt{sSt}_2^1} \dots \xrightarrow{\mathtt{sSt}_2^{m-1}} s_1^m \xrightarrow{\mathtt{sSt}_2^m} s_2^{m+1}, \text{ with } s_2 = s_2^0,\ s_2' = s_2^{m+1},$$

we have (i) $s_2' \approx_{obs} s_1'$*; (ii) for each state* s_1^i *there exists a state* s_2^j *such that* $s_1^i \approx_{obs'} s_2^j$ *for some obs′; (iii) for each state* s_2^j *there exists a state* s_1^i *such that* $s_2^j \approx_{obs'} s_1^i$ *for some obs′, where* $0 \le i \le n$ *and* $0 \le j \le m$*.*

The weak bisimulation relation for programs defined above requires a weak transition that relates two states with at most one observable transition. This definition reflects the *structural* properties of a program and can be characterized as a *small-step semantics.* It directly implies the lemma below that relates the weak bisimulation relation of programs to a *big-step semantics.*

Lemma 1. *Let* $\mathtt{p}, \mathtt{q}$ *be programs, obs a set of observable locations. Then* $\mathtt{p} \approx_{obs} \mathtt{q}$ *if and only if* $val_{D,s}(\mathtt{p}) \approx_{obs} val_{D,s}(\mathtt{q})$ *for any first-order structure D, state s.*

4.2 The Weak Bisimulation Modality

We introduce a weak bisimulation modality which allows us to relate two programs that behave indistinguishably on the observable locations.

Definition 9 (Weak bisimulation modality—syntax). *The* bisimulation modality $[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)$ *is a modal operator providing compartments for programs* $\mathtt{p}$*,* $\mathtt{q}$ *and location sets obs and use. We extend our definition of formulas: Let* ϕ *be a* PL*-DL formula and* $\mathtt{p}, \mathtt{q}$ *two* PL *programs and* obs, use *two location sets such that* $pv(\phi) \subseteq obs$ *where* $pv(\phi)$ *is the set of all program variables occurring in* ϕ*, then* $[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi$ *is also a* PL*-DL formula.*

The intuition behind the location set $usedVar(s, \mathtt{p}, obs)$ defined below is to capture precisely those locations whose value influences the final value of an observable location $\mathtt{l} \in obs$ (or the evaluation of a formula ϕ) after executing a program $\mathtt{p}$. We approximate the set later by the set of all program variables in a program that are used before being redefined (i.e., assigned a new value).

Definition 10 (Used program variable). *A variable* $\mathtt{v} \in \mathsf{PV}$ *is called* used by program $\mathtt{p}$ *relative to a location set obs, if there exists an* $\mathtt{l} \in obs$ *such that*

$$D, s \models \forall \mathtt{v}_1.\exists \mathtt{v}_0.((\langle \mathtt{p} \rangle \mathtt{l} = \mathtt{v}_1) \to (\{\mathtt{v} := \mathtt{v}_0\}\langle \mathtt{p} \rangle \mathtt{l} \neq \mathtt{v}_1))$$

The set $usedVar(s, \mathtt{p}, obs)$ *is defined as the smallest set containing all used program variables of* $\mathtt{p}$ *with respect to obs.*

The formula defining a used variable $\mathtt{v}$ of a program $\mathtt{p}$ encodes that there is an interference with a location contained in *obs*. In Ex. 3, $\mathtt{z}$ is a used variable.

We formalize the semantics of the weak bisimulation modality:

Definition 11 (Weak bisimulation modality—semantics). *Let* $\mathtt{p}, \mathtt{q}$ PL-*programs,* D, s, β*, obs, use as above, then* $val_{D,s,\beta}([\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi) = tt$ *if and only if*

1. $val_{D,s,\beta}([\mathtt{p}]\phi) = tt$
2. $use \supseteq usedVar(s, \mathtt{q}, obs)$
3. *for all* $s' \approx_{use} s$ *we have* $val_{D,s}(\mathtt{p}) \approx_{obs} val_{D,s'}(\mathtt{q})$

Lemma 2 ([13]). *Let obs be the set of all locations observable by* ϕ *and* $\mathtt{p}, \mathtt{q}$ *be programs. If* $\mathtt{p} \approx_{obs} \mathtt{q}$ *then* $val_{D,s,\beta}([\mathtt{p}]\phi) \leftrightarrow val_{D,s,\beta}([\mathtt{q}]\phi)$ *holds for all* D*,* s*,* β*.*

The following lemma illustrates the meaning of used variable set *use*. An extended sequent for the bisimulation modality is $\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi, \Delta$.

Lemma 3 ([13]). *An extended sequent* $\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi, \Delta$ *within a sequential block bl represents a certain state* s_1*, where* $\mathtt{P}$ *is the original program executed in bl,* $\mathtt{p}$ *the program to be executed in bl at state* s_1*, and* $\mathtt{p}'$ *the program already executed in bl; likewise,* $\mathtt{Q}$ *is the program to be generated in bl,* $\mathtt{q}$ *the already generated program in bl,* $\mathtt{q}'$ *the program remaining to be generated in bl (illustrated in Fig. 3). Then use are the* dynamically observable locations *such that: (i)* $\mathtt{p} \approx_{obs} \mathtt{q}$*; (ii)* $\mathtt{P} \approx_{obs} \mathtt{Q}$*; (iii)* $\mathtt{p}' \approx_{use} \mathtt{q}'$*.*

4.3 Sequent Calculus Rules for Bisimulation Modality

We define a sequent calculus over extended sequents with the weak bisimulation modality. One example is the assignment rule:

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{l} := \mathtt{r}\}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{l} = \mathtt{r}; \omega \between \mathtt{l} = \mathtt{r}; \overline{\omega}\ ]@(obs, use - \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use \\ \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{l} = \mathtt{r}; \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

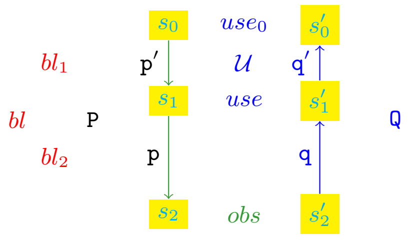

Fig. 3. Program in a sequential block.

Here, ϖ represents the program generated before rule application. The *use* set contains all variables that may affect the values of observable locations in the final state. If $\mathtt{l}$ is among those variables, we update the *use* set by removing $\mathtt{l}$ and adding $\mathtt{r}$ which is read by the assignment. Otherwise, we generate no code.

Updates record the evaluation of the locations in an execution path. For the purpose of information flow analysis, we need a set of more precise program generation rules that also involve the updates in a sequential block.

An elementary update $\mathtt{l}_1 := \mathtt{exp}_1$ is *independent* from another elementary update $\mathtt{l}_2 := \mathtt{exp}_2$ if $\mathtt{l}_1$ does not occur in $\mathtt{exp}_2$ and $\mathtt{l}_2$ does not occur in $\mathtt{exp}_1$.

Definition 12 (SNF update). *An update is in* sequentialized normal form *(SNF), denoted by $\mathcal{U}^{snf}$, if it has the shape of a sequence of two parallel updates $\{u^a_1\|\ldots\|u^a_m\}\{u_1\|\ldots\|u_n\}$, $m \geq 0, n \geq 0$. We call $\{u_1\|\ldots\|u_n\}$ the* core *update, denoted by $\mathcal{U}^{snf_c}$, where each u_i is an* elementary update *of the form $\mathtt{l}_i := \mathtt{exp}_i$, and all u_i, u_j $(i \neq j)$ are independent and have no conflict. We call $\{u^a_1\|\ldots\|u^a_m\}$ the* auxiliary *update, denoted by $\mathcal{U}^{snf_a}$, where (i) each u^a_i is of the form $\mathtt{l}^k := \mathtt{l}$ $(k \geq 0)$; (ii) $\mathtt{l}$ is a program variable; (iii) $\mathtt{l}^k$ is a* fresh *program variable; (iv) there is no conflict between u^a_i and u^a_j for all $i \neq j$.*

An NF update with independent elementary updates is also an SNF update with only a core part. Sound rules to compute the SNF of updates and maintain SNF after rule application are in[13]. Using SNF of updates, the assignment rule becomes:

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf}[\,\omega \between \varpi\,]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}^{snf}[\,\mathtt{l}=\mathtt{r};\omega \between \mathtt{l}=\mathtt{r}_1;\varpi\,]@(obs, use-\{\mathtt{l}\}\cup\{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\,\mathtt{l}=\mathtt{r};\omega \between \varpi\,]@(obs, use)\phi, \Delta & otherwise \end{array}\right)}$$

where $\mathcal{U}_1^{snf} = \mathcal{U}_1^{snf_a}\{\ldots\|\mathtt{l} := \mathtt{r}_1\}$ is the SNF of $\mathcal{U}^{snf}\{\mathtt{l} := \mathtt{r}\}$.

Whenever the core update is empty, we use the auxAssignment rule:

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf_a}[\,\omega \between \varpi\,]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}^{snf_a}[\,\omega \between \mathtt{T}_1\ \mathtt{l}^0 = \mathtt{l};\varpi\,]@(obs, use-\{\mathtt{l}^0\}\cup\{\mathtt{l}\})\phi, \Delta & \text{if } \mathtt{l}^0 \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf_a}[\,\omega \between \varpi\,]@(obs, use)\phi, \Delta & otherwise \end{array}\right)}$$

where $\mathcal{U}^{snf_a} = \{u\}$ and $\mathcal{U}_1^{snf_a} = \{u\|\mathtt{l}^0 := \mathtt{l}\}$ being the auxiliary update

The auxiliary assignments are always generated at the start of a sequential block. Fig. 4 shows some other extended sequent calculus rules (`nop` denotes *no operation*, and $\_$ denotes the place holder of *empty*). More are in [13].

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U}^{snf}@(obs, \_)\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\,\texttt{nop} \between \texttt{nop}\,]@(obs, obs)\phi, \Delta}$$

$$\text{ifElse}\ \frac{\begin{array}{c}\Gamma, \mathcal{U}^{snf}\texttt{b} \Longrightarrow \mathcal{U}^{snf}[\,\texttt{p};\omega \between \overline{\texttt{p};\omega}\,]@(obs, use_{\texttt{p};\omega})\phi, \Delta \\ \Gamma, \mathcal{U}^{snf}\neg\texttt{b} \Longrightarrow \mathcal{U}^{snf}[\,\texttt{q};\omega \between \overline{\texttt{q};\omega}\,]@(obs, use_{\texttt{q};\omega})\phi, \Delta\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf}[\,\texttt{if (b) \{p\} else \{q\}};\omega \between \\ \qquad\qquad \texttt{if (b) }\{\overline{\texttt{p};\omega}\}\texttt{ else }\{\overline{\texttt{q};\omega}\}\,]@(obs, use_{\texttt{p};\omega} \cup use_{\texttt{q};\omega} \cup \{\texttt{b}\})\phi, \Delta\end{array}}$$

(with `b` boolean variable.)

$$\text{loopInv}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf}inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\texttt{b} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod} \\ \qquad\qquad\qquad\qquad\qquad [\,\texttt{p} \between \overline{\texttt{p}}\,]@(use_1 \cup \{\texttt{b}\}, use_2)inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\neg\texttt{b} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod}[\,\omega \between \overline{\omega}\,]@(obs, use_1)\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\,\texttt{while(b)\{p\}}\,\omega \between \texttt{while(b)}\{\overline{\texttt{p}}\}\,\overline{\omega}\,]@(obs, use_1 \cup use_2 \cup \{\texttt{b}\})\phi, \Delta}$$

Fig. 4. A collection of extended sequent calculus rules with SNF updates.

Sequent rule application for the bisimulation modality is in two phases:

Phase 1. Symbolic execution of source program as usual. In addition, the observable location sets *obs* are propagated, because they contain the observable locations (by the program fragments and the post condition) to be used in the second phase. For the purpose of information flow analysis, *obs* contains the `Low` variables and the locations used in the continuation of the program, e.g., program variables used after a loop must be reflected in the observable locations of the loop body. It results in an SET as illustrated in Fig. 2.

Phase 2. Generate the simplified program and used variable set by applying the rules bottom-up. Start with a leaf node (emptyBox rule) and generate the program within its sequential block first, e.g., bl_3 in Fig. 2. These programs are combined by rules corresponding to statements containing a sequential block, such as loopInv (containing bl_3 and bl_4). One continues with the GSB containing the compound statements, e.g., GSB(bl_2), and so on, until the root. The order of processing the sequential blocks matters, for instance, the program for bl_4 must be generated before that for bl_3, because the observable locations in n_3 depend on the used variable set of bl_4, according to the loopInv rule.

Remark 2. The SNF updates used in the calculus rules are the SNF updates in the current sequential block. A program execution path may contain several sequential blocks. We do keep the SNF update for each sequential block without simplifying them further into a bigger SNF update for the entire execution path. In Fig. 2, the execution path from node n_0 to n_4 involves sequential blocks

bl_0, bl_1 and bl_4. When we generate the program in bl_4, we should formally use $\mathcal{U}_0^{snf}\mathcal{U}_2^{snf}\mathcal{U}_4^{snf}$, however, we just care about the SNF update of bl_4 when generating the program for bl_4, so in the above rules, $\mathcal{U}^{snf}$ refers to $\mathcal{U}_4^{snf}$ only.

Lemma 4 ([13]). *The extended sequent calculus rules are sound.*

5 Information Flow Security Analysis

Example 4. Let `l` be `Low` variables and `h` be `High` variables in a program. We discuss whether the standard security policy, as stated in the introduction, holds for some example programs:

(i). $\mathtt{l} = \mathtt{h};$ is *insecure* because information of `h` is leaked directly to `l`.
(ii). $\mathtt{l} = \mathtt{h}; \mathtt{l} = \mathtt{0};$ is *secure* because the final value of `l` does not depend on `h`.
(iii). $\mathtt{h} = \mathtt{0}; \mathtt{l} = \mathtt{h};$ is *secure* because the final value of `l` is always 0.
(iv). $\mathtt{l} = \mathtt{h}; \mathtt{l} = \mathtt{l} - \mathtt{h};$ is *secure* because the final value of `l` is always 0.
(v). $\mathtt{if(h > 0)}\ \{\mathtt{h} = \mathtt{l}; \mathtt{l} = \mathtt{h};\}$ is *secure* because the final value of `l` is unchanged.
(vi). $\mathtt{if(h > 0)}\ \{\mathtt{l} = \mathtt{1};\}\ \mathtt{else}\ \{\mathtt{l} = \mathtt{2};\}$ is *insecure* because partial information of `h` can be learned from the final value of `l`.
(vii). $\mathtt{if(h > 0)}\ \{\mathtt{l} = \mathtt{1};\}\ \mathtt{else}\ \{\mathtt{l} = \mathtt{2};\}\ \mathtt{l} = \mathtt{0};$ is *secure* because the final value of `l` is always 0.
(viii). $\mathtt{if(h > 0)}\ \{\mathtt{l} = \mathtt{1};\}\ \mathtt{else}\ \{\mathtt{l} = \mathtt{1};\}$ is *secure* because the final value of `l` is always 1.

In (i), the information from the value of the `High` variables flows directly to the `Low` variables (*explicit* flow). It is also possible that information flows indirectly from `High` to `Low` variables (*implicit* flow), as shown in (vi).

The approaches using security type systems (as overviewed in [1]) are *sound*, i.e., an *insecure* program can never be classified as *secure*. However, they often overapproximate and classify *secure* programs as *insecure* or *unknown*. In Ex. 4 they have trouble classifying (iii), (iv), (v) due to *value insensitivity*, as well as (vii), (viii) which requires *control-flow sensitivity*.

We introduced a sound program transformation approach in the previous sections. The choice of observable locations *obs* does not affect the soundness of the framework. We can fix *obs* as the `Low` variables, then the generated program is a *dependency flow* of `Low` variables. Along with program generation, we maintain the used variable set *use* in the extended sequent calculus rules. When program generation is finished, by Lemma 3 and Def. 10, *use* is the set of observable locations in the initial state and each variable that belongs to *use* will interfere with *obs* (`Low` variables) in the final state. For information flow security this means every input variable that belongs to *use* will interfere with `Low` output variables. According to the definition of non-interference, it suffices to guarantee that `High` variables do not occur in *use* to enforce non-interference.

Theorem 1 (Non-Interference Enforcement). *Given a* PL *program* `p`*, a set of* `High` *variables* `H` *and a set of* `Low` *variables* `L`*; after program transformation, we obtain program* `q` *and used variable set* use_0*, such that* $\mathtt{p} \approx_L \mathtt{q}$*. The non-interference policy is enforced if for all* $\mathtt{h} \in \mathtt{H}$*,* $\mathtt{h} \notin use_0$*.*

Proof. Direct result of Lemma 3, Def. 10 and notion of non-interference. □

Because the program transformation process employs first-order reasoning and partial evaluation in the symbolic execution phase, as well as using updates during program generation, we achieve a more precise information flow analysis than security type systems.

We analyze the programs in Ex. 4 by fixing $\mathtt{l}$ as observable locations. For (i), we generate the program $\mathtt{l} = \mathtt{h};$ and used variable set $use = \{\mathtt{h}\}$, so the program is *insecure*. For (ii), the first statement $\mathtt{l} = \mathtt{h};$ is not generated according to the assignment rule in Sect. 4.3, and $use = \emptyset$, so it is *secure*. The SNF update of (iii) is $\{\mathtt{h} := 0 \| \mathtt{l} := 0\}$, it generates program $\mathtt{l} = 0;$ which is *secure*. For (iv), the SNF update is $\{\mathtt{l} := 0\}$ so the generated program is *secure*. For (v), we generate $\mathtt{if}(\mathtt{h} > 0)\ \{\mathtt{l} = \mathtt{l};\}$ with used variable set $use = \{\mathtt{h}\}$, which cannot be classified as *secure*. For (vi), the generated program is identical to the source and $use = \{\mathtt{h}\}$, which is classified as *insecure*. For (vii), we generate $\mathtt{if}(\mathtt{h} > 0)\ \{\mathtt{l} = 0;\}\ \mathtt{else}\ \{\mathtt{l} = 0;\}$ and $use = \{\mathtt{h}\}$, which cannot be classified as *secure*. We cannot classify program (viii) as *secure* as well for the same reason.

While our approach achieves a more precise analysis of information flow than type-based approaches, we have trouble with (v), (vii), (viii). But this can be addressed by extended sequent rules tailored to information flow analysis:

assignNotSelf

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf}[\ \omega\ \between\ \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{l} = \mathtt{r}; \omega\ \between\ \mathtt{l} = \mathtt{r}_1; \overline{\omega}\ ]@(obs, use - \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use\ \wedge\ \mathtt{r}_1 \neq \mathtt{l} \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{l} = \mathtt{r}; \omega\ \between\ \overline{\omega}\ ]@(obs, use)\phi, \Delta & \textit{otherwise}\end{array}\right)}$$

where $\mathcal{U}_1^{snf} = \mathcal{U}_1^{snf_a}\{\ldots \| \mathtt{l} := \mathtt{r}_1\}$ is the SNF of $\mathcal{U}^{snf}\{\mathtt{l} := \mathtt{r}\}$).

ifElseUnify

$$\frac{\begin{array}{l}\Gamma, \mathcal{U}\mathtt{b} \Longrightarrow \mathcal{U}[\ \mathtt{p}; \omega\ \between\ \overline{\mathtt{p}; \omega}\ ]@(obs, use_{\mathtt{p};\omega})\phi, \Delta \\ \Gamma, \mathcal{U}\neg\mathtt{b} \Longrightarrow \mathcal{U}[\ q; \omega\ \between\ \overline{q; \omega}\ ]@(obs, use_{\mathtt{q};\omega})\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{if}\ (\mathtt{b})\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}; \omega\ \between\ \overline{\mathtt{p}; \omega}\ ]@(obs, use_{\mathtt{p};\omega})\phi, \Delta}$$

(with $\mathtt{b}$ boolean variable, $\overline{\mathtt{p}; \omega} \approx_{obs} \overline{\mathtt{q}; \omega}$, and $use_{\mathtt{p};\omega} = use_{\mathtt{q};\omega}$)

loopInvNoBody

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf} inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod} \\ \qquad\qquad\qquad\qquad [\ \mathtt{p}\ \between\ \overline{\mathtt{p}}\ ]@(use_1 \cup \{\mathtt{b}\}, use_2) inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\neg\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod}[\ \omega\ \between\ \overline{\omega}\ ]@(obs, use_1)\phi, \Delta\end{array}}{\left(\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{while}(\mathtt{b})\{\mathtt{p}\}\, \omega\ \between\ \overline{\omega}\ ]@(obs, use_1)\phi, \Delta & \text{if } use_1 = \emptyset \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{while}(\mathtt{b})\{\mathtt{p}\}\, \omega\ \between\ \mathtt{while}(\mathtt{b})\{\overline{\mathtt{p}}\}\, \overline{\omega}\ ]@(obs, use_1 \cup use_2 \cup \{\mathtt{b}\})\phi, \Delta & \textit{otherwise}\end{array}\right)}$$

The assignNotSelf rule avoids the generation of self assignments $\mathtt{l} = \mathtt{l};$. The ifElseUnify rule checks whether the `then` branch and `else` branch have the same effect, if so, we do not generate a conditional block. The loopInvNoBody rule avoids the generation of a loop body, if the used variable set obtained in the continuation of the loop is empty. Because in this case, the loop does not affect the values of the observable locations at all.

Now programs (v), (vii), (viii) in Ex. 4 can be classified properly. For (v), according to assignNotSelf, we do not generate any program in the `then` branch, then apply ifElseUnify rule (both branches are empty), and obtain the empty program, with used variable set $use = \{\mathtt{l}\}$. It can be classified as *secure*. For (vii), we generate the program `l = 0;` and $use = \emptyset$, it is *secure*. Program of (viii) is also *secure* for the same reason as (vii).

Example 5. Consider the following program with loop invariant $\mathtt{l} > \mathtt{0}$ and post condition $\mathtt{l} \doteq \mathtt{2}$ ($\doteq$ being first-order equality). Let `l` be `Low` and `h` be `High`.

```
l = 1; while(h>0) {l++; h--;} if(l>0) {l = 2;}
```

After symbolic execution of the loop we have three branches. In the branch that continues after the loop, we encounter a conditional. With the loop invariant we can infer that the guard holds, so we only execute the then branch with `l = 2;`. Every open goal is closeable, so the program is proven. We start to analyze information flow security with $obs = \{\mathtt{l}\}$. In the first step, the statement `l = 2;` is generated empty used variable set. According to loopInvNoBody, we do not generate loop body code. Continuing with `l = 1;`, we obtain the program `l = 2;` and an empty used variable set. According to Theorem 1, this program is secure.

Remark 3. We can perform the program transformation without suitable loop invariants (just use *true*), as discussed previously [16,17]. This achieves a higher degree of automation, which is desirable in the context of program specialization. However, proper loop invariants will increase the precision of the information flow analysis. Without the loop invariant $\mathtt{l} > \mathtt{0}$ in Ex. 5, we have to generate the conditional as well as the loop body, and then we cannot classify the program.

6 Conclusion

We presented a novel approach to analyze information flow security based on sound program simplification and verification. It ensures correctness and security of a program at once. First-order reasoning analyzes variable dependencies, aliasing, and eliminates infeasible execution paths. Interleaving partial evaluation with symbolic execution reduces SETs. Sound program transformation generates a simplified program that represents the dependency flow of the low variables. The set of used variables is maintained during synthesis, allowing to check non-interference by a simple lookup. As compared to approaches based on security type systems [1], we obtain higher precision due to value and control flow sensitivity, as well as first-order reasoning.

In contrast to other approaches based on deductive verification [7,6,18] (see also discussion in the Introduction), we completely avoid adding complexity to the target program or complex quantification. An orthogonal approach to ours that uses abstraction to increase automation is [18]. It could be easily combined.

In the future we plan to implement our approach, perform larger case studies, and to look at more realistic security policies than just non-interference, such as *declassification* [19].

References

1. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1) (2003) 5–19
2. Cohen, E.S.: Information transmission in computational systems. In: SOSP. (1977) 133–139
3. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. (1982) 11–20
4. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2/3) (1996) 167–188
5. Hunt, S., Sands, D.: On flow-sensitive security types. In: POPL. (2006) 79–90
6. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In Hutter, D., Ullmann, M., eds.: 2nd Intl. Conf. on Security in Pervasive Computing. Volume 3450 of LNCS., Springer (2005) 193–209
7. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, CSFW-17, IEEE Computer Society (2004) 100–114
8. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In Giacobazzi, R., ed.: 11th Static Analysis Symposium (SAS), Verona, Italy. Volume 3148 of LNCS., Springer (2004) 100–115
9. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In Butler, M., Schulte, W., eds.: FM. Volume 6664 of LNCS., Springer (2011) 200–214
10. Bubel, R., Hähnle, R., Ji, R.: Interleaving symbolic execution and partial evaluation. In: Post Conf. Proc. FMCO 2008. LNCS, Springer-Verlag (2009)
11. Ji, R., Hähnle, R., Bubel, R.: Program transformation based on symbolic execution and deduction. In: SEFM. (2013) 289–304
12. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
13. Ji, R., Hähnle, R., Bubel, R.: Program transformation based on symbolic execution and deduction. Technical Report CS-2013-0348, TU Darmstadt, Fachbereich Informatik (2013) `https://www.se.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Page_Content/Group_Members/ran_ji/TUD-CS-2013-0348.pdf`.
14. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In Hermann, M., Voronkov, A., eds.: LPAR. Volume 4246 of LNCS., Springer (2006) 422–436
15. Beckert, B., Hähnle, R., Schmitt, P., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2006)
16. Bubel, R., Hähnle, R., Ji, R.: Program specialization via a software verification tool. In Aichernig, B., de Boer, F.S., Bonsangue, M.M., eds.: Post Conf. Proc. of FMCO 2009. LNCS, Springer (2010)
17. Ji, R., Bubel, R.: PE-KeY: A Partial Evaluator for Java Programs. In: IFM. LNCS, Springer (2012) 283–295
18. Bubel, R., Hähnle, R., Weiss, B.: Abstract interpretation of symbolic execution with explicit state updates. In de Boer, F., Bonsangue, M.M., Madelaine, E., eds.: Post Conf. Proc. FMCO 2008. Volume 5751 of LNCS., Springer (2009) 247–277
19. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: CSFW. (2005) 255–269