Synthesis of a Permissive Security Monitor

Narges Khakpour(✉) and Charilaos Skandylas

Linnaeus University, Växjö, Sweden
narges.khakpour@lnu.se

Abstract. In this paper, we propose a new sound method to synthesize a permissive monitor using boolean supervisory controller synthesis that observes a Java program at certain checkpoints, predicts information flow violations and applies suitable countermeasures to prevent violations. To improve the permissiveness, we train the monitor and remove false positives by executing the program along with its executable model. If a security violation is detected, the user can define sound countermeasures, including declassification to apply in the checkpoints. We implement a tool that automates the whole process and generates a monitor. We evaluate our method by applying it on the Droidbench benchmark and one real-life Android application.

1 Introduction

Confidentiality of secret information manipulated by a program is usually formalized as a noninterference baseline policy [13], which demands that low-sensitive outputs should not be influenced by high-sensitive inputs. Several methods and tools (e.g., JFlow JIF [19], Caml-based FlowCaml [25]) have been developed in the last decades to analyze or enforce confidentiality. Information flow monitors are a technique to enforce noninterference dynamically [4,7,11,14,15,22]. The idea is to monitor the executions of a program at runtime and control its compliance to security policies. As dynamic monitors only decide about the current execution, for which more information is available at runtime, they enable us to do a more precise analysis, and are usually more permissive compared to static methods [18], e.g. [21] proved that dynamic monitors are more permissive in the flow-insensitive case, where variables are assigned the security levels at the beginning of the execution and the security levels don't change during the execution. Hybrid monitors [14,20,24] are a class of dynamic monitors that combine static and dynamic analysis.

Consider the following program where `h` is secret and the rest of variables and objects are public:

```
obj1.x=h;
if(a>0)
   while(b>0){obj1.x=0;b=b-1;}
else  obj1.x=1;
 f(l); l=obj1.x; obj2.att=l; print(obj2);
```


J. Lopez et al. (Eds.): ESORICS 2018, LNCS 11098, pp. 48–65, 2018.
https://doi.org/10.1007/978-3-319-99073-6_3

If $\texttt{a} > 0 \wedge \texttt{b} \leq 0$ holds, then the value of `h` will flow to `l` through `obj1.x` and the program is insecure, otherwise the program is secure. Security type systems, one of the main techniques for static analysis, reject this program completely, while dynamic monitors allow the secure executions, i.e., if $\texttt{a} > 0 \wedge \texttt{b} \leq 0$ does not hold, the program is secure and executes normally, otherwise, the program is permitted to run and a certain strategy is designed to protect the system. The existing strategies either (a) manipulate the attacker's observation as soon as a violation is detected, i.e. at the observation point (e.g. `print(obj2)` in the above example) [14,20], (b) run several instances of the program simultaneously with various inputs to ensure that the program does not reach an insecure state [5,11], or (c) control assignment of low sensitive data in high contexts (i.e. a branch on high sensitive data) [4,26]. The approaches in category (b) are expensive and have a huge overhead, due to running several instances of the program simultaneously [12]. The methods in the categories (a) and (c) detect security violations one-step before their occurrence [20], and as a result, it becomes complicated and expensive, if possible at all, to apply a proper countermeasure to avoid information leakage.

In the above example, if executing `f(l)` results in modifying the database or sending data over a network and we detect the violation immediately before `print(obj2)`, then a suitable countermeasure to fix the violation might require us to recover the system to a state where a proper countermeasure can be applied, which is difficult, if possible at all. On the other hand if we know that the condition $\texttt{a} > 0 \wedge \texttt{b} \leq 0$ leads to a violation before executing the program, then we are able to apply a countermeasure before `f(l)`.

Although, dynamic monitors are usually more permissive than static methods, they still can produce false positives and are not always the most permissive monitor. Hence, it is crucial to construct sound dynamic and hybrid monitors that allow as many paths as possible. *In addition, to the best of our knowledge, there is no dynamic monitor that can predict confidentiality violations at runtime before the violation points and allows applying user-defined countermeasures, in particular declassification, to avoid security violations.*

To tackle the above challenges, we propose a new approach based on boolean supervisory controller synthesis [6] to synthesize a hybrid monitor that monitors a program written in a subset of Java at certain checkpoints, predicts security violations and applies suitable countermeasures in checkpoints to avoid future leakages. Given a program, a set of checkpoints from where the program can be observed by the monitor, a set of observation points where the attacker can observe the application in (See Fig. 2), we use the controller synthesis method proposed in [6] to synthesize a set of security guards for the checkpoints that guarantee no information leakage in future, up to the next checkpoint.

To improve the permissiveness of the monitor, we construct an executable model of the monitored program that contains only observation points and checkpoints. In the training phase, we run the program along with its executable model to train the monitor and improve its permissiveness; if a violation is predicted at runtime in a checkpoint, we execute the program model to check whether the

security guard of the current checkpoint is restrictive or not. If it is restrictive, we learn and relax the security guard to allow the current (symbolic) execution path in future. After the monitor training, we construct a more lightweight monitor that controls and predicts information flow using the learnt security guards in the checkpoints to protect the program.

Furthermore, we design a set of secure countermeasures to be applied in the checkpoints in case of security violations that prevent the program from reaching an insecure state. A user-defined countermeasure can be applied at runtime, provided that it satisfies certain conditions. One of the main countermeasures that can be applied is to declassify information, i.e. degrade the security level of variables. In [16], we proved that the method is sound and enforces localized delimited release [2]. If the monitor does not perform any declassification, it enforces termination-insensitive noninterference. Furthermore, we implement a tool-set to support our method and conduct some experiments to evaluate the method. Our contributions are the following:

- *Permissive Sound Monitor.* We propose a new approach using boolean controller synthesis to efficiently construct a hybrid flow-sensitive security monitor that predicts *future* information flow at a few *predefined checkpoints* in a Java program. To improve the monitor permissiveness, we train the monitor in a testing environment and eliminate false positives as far as possible.
- *Supporting User-Defined Countermeasures.* In contrast to the existing dynamic monitors that apply a few fixed countermeasures, detecting a violation multiple steps ahead of its occurrence enables the user to design and apply various countermeasures in the checkpoints, provided that they introduce no information leakage. Our method is the first method that allows dynamic correct-by-construction information disclosure, even though the declassification policies are simple. While existing approaches enforce a variation of noninterference, our method guarantees localized delimited release, and enforces termination-insensitive noninterference in case of no information release.
- *Tool Support.* Our method is supported by a tool-set to control information flow in programs written in a sub-language of Java. We also conducted experiments to evaluate the effectiveness of the method.

This paper is organized as follows. We briefly introduce the controller synthesis problem in Sect. 2, and give an overview of the approach in Sect. 3. Section 4 presents the program syntax, the security control flow model and the program executable model. We introduce our monitor construction approach in Sect. 5. In Sect. 6, we present the toolset and evaluate the approach. In Sect. 7, we discuss related work and Sect. 8 concludes the paper.

2 Preliminaries

In this section, we briefly review the symbolic supervisory controller synthesis method proposed in [6], the goal of which is to construct a controller to control a

system behavior, so that the bad states are avoided. In this method, the system behavior is represented by a symbolic control flow graph. Let $V = \langle v_1, \ldots, v_n \rangle$ be a tuple of variables, $\mathcal{D}_{v_i}$ be the (infinite) domain of a variable v_i, and $\mathcal{D}_V = \prod_{i \in [1,n]} \mathcal{D}_{v_i}$. A *valuation* $\boldsymbol{\nu}$ of V is a tuple $\langle \boldsymbol{\nu}_1, \ldots, \boldsymbol{\nu}_n \rangle \in \mathcal{D}_V$, and we show the value of v_i in $\boldsymbol{\nu}$ by $\boldsymbol{\nu}(v_i)$, $1 \leq i \leq n$. A *predicate* P over a tuple V is defined as a subset $P \subseteq \mathcal{D}_V$ (a state set for which the predicate holds). We show the union of two vectors V_1 and V_2 by $V_1 \uplus V_2$.

Definition 1 (Symbolic Control Flow Graphs). *A* symbolic control flow graph *(SCFG) is a tuple* $\mathcal{G} = \langle L, V, I, l_o, v_0, \Delta \rangle$ *where* L *is a finite non-empty set of locations,* $V = \langle v_1, \ldots, v_n \rangle$ *is a tuple of variables,* I *is a vector of inputs,* l_0 *is the initial location,* $v_0 \in \mathcal{D}_V$ *shows the initial valuation of the variables, and* Δ *is a finite set of symbolic transitions* $\delta = \langle G_\delta, A_\delta \rangle$ *where* $G_\delta \subseteq \mathcal{D}_{V \uplus I}$ *is a predicate on* $V \uplus I$*, which guards the transition, and* $A_\delta : \mathcal{D}_V \mapsto \mathcal{D}_{V \uplus I}$ *is the update function of* δ*, defined as a set of assignments.*

Initially, $\mathcal{G}$ is in its initial state. A transition can only be fired if its guard is satisfied and when fired, the variables are updated according to its update function. Let l and l' be two locations. We use the notation $l \xrightarrow{\langle G_\delta, A_\delta \rangle} l'$ to represent a symbolic transition $\langle G_\delta, A_\delta \rangle$ with the source l and target l'. The semantics of a SCFG $\mathcal{G}$ is defined in terms of a deterministic finite state machine.

In this method, the inputs are partitioned into two sets of *controllable* and *uncontrollable* inputs: an input is uncontrollable if it can not be prevented from occurring in a system, while controllable inputs are issued by the controller to control the system behaviour. Let $\psi : L \rightarrow \mathcal{D}_V$ be the invariants defined for the locations (i.e. an invariant $\psi(l)$ is a condition on the valuation of variables that must always hold when the system enters the location l), and $I_c \subseteq I$ be the set of controllable inputs. Given an invariant ψ and a SCFG $\mathcal{G}$, a controller $\mathcal{C} : L \rightarrow \mathcal{D}_{V \uplus I_c}$ is synthesized to observe the system and allow or prohibit the controllable inputs, so that the system $\mathcal{G}$ avoids entering a bad state, i.e. a state that does not satisfy its invariant.

3 The Method Overview

Figure 1 shows an overview of our method. The Java program is annotated with checkpoints, observations points (can be avoided), initial security labels and entry points (See Fig. 2 and Sect. 4). A checkpoint is essentially a method call in which we monitor the program, and can apply a countermeasure if needed. The checkpoints are not permitted to exist under branch statements. An observation point is a point that leads to an observation by the attacker, that is either a method call or the exit point of a branch of a conditional/loop whose other branch contains a method call observation point. We construct a boolean symbolic control flow graph that describes the program control flow enriched with security typing information (See Sect. 4) which is fed to the Reax controller synthesis tool [6]. For each checkpoint, the tool generates the abstract security guards in terms of program paths and security types that in principle show the

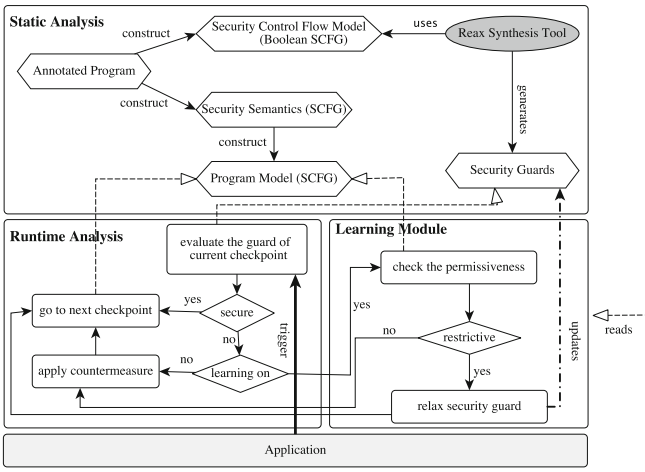

Fig. 1. The method overview

paths that do not lead to insecure states (See Sect. 5). We also express the (security) semantics of the program in terms of a symbolic control flow graph that includes both the program behaviour and the security typing information. Given the security semantics, we construct a model called program model that includes only observation points in addition to checkpoints (See Sect. 4). We propose a framework to construct a secure monitor in Sect. 5 that applies the countermeasures either in the checkpoints and/or in the observation points, depending on the user preferences.

The program is observed by the monitor in the checkpoints (e.g. the `run` method in Fig. 2) at runtime. The monitor checks the security guards of the current checkpoint to determine whether the program will reach an insecure state (e.g. in the `println` method in Fig. 2) or not. If not, the program will continue its execution. Otherwise, if the learning feature is enabled (e.g. in the training phase), the monitor executes its program model using a model execution engine to ensure that the generated security guard is not restrictive. If the generated security guard of the current checkpoint is restrictive, it is relaxed to allow this secure path henceforth, i.e. the security guards are learned and improved over time. Afterwards, the program continues its execution by applying a countermeasure. This monitor will be the most permissive monitor, if we train it sufficiently, as it will never block a secure path.

4 Security Control Flow Model

We consider a sub-language of Java whose simplified syntax of statements is shown in Fig. 3, that includes loop statements, conditional statements, assignments, a return command, constructors and method calls. In this figure, v is a variable of primitive type, e is an expression, stm is a statement, o is an

```
/* @EntryPoint */
/* @CheckPoint */
public void run(/*@SecurityInit(securityLevel="H", policyType="IC")*/
        int ah, int ad){ int ares = 0; Class1 object1 = new Class1(ad);
    int aw = object1.attr + 3;
    while (0 < aw) {
        ares = object1.attr + 10;
        object1.attr = ah + 1;
        ah = ares - 1;
        aw = aw - 3;}
    object1.attr = object1.attr * ad;
    /* @ObservationPoint (default="System.out.println(3000)")*/
    System.out.println(/* @SecurityPolicy(securityLevel="L", policyType="IC")
    */object1.attr);
}
```

Fig. 2. Java code snippet

object, *stms* is a sequence of statements, $o.m(\overrightarrow{e})$ is a method call with arguments $\overrightarrow{e} = e_1 \dots e_m$, and $\surd$ shows an empty sequence of statements. The statements in a bracket are optional and ϵ shows no argument.

We follow a type-based flow-sensitive method and assign a security type to each variable, i.e. the security type of a variable may change during the program execution. A variable is either a primitive variable or an instance variable of a user-defined type. We consider a two-level security lattice $\langle \mathcal{L}, \sqsubseteq, \sqcup \rangle$ where $\mathcal{L} = \{H, L\}$ is the set of security types, $\sqsubseteq$ is a partial order defined over $\mathcal{L}$ and $\sqcup$ is an operator that gives the least upper bound of two elements in $\mathcal{L}$ (i.e. disjunction). The function $var(e)$ returns the variables that appear in the expression e, and if e is an object, it returns the object itself along with all its accessible attributes (i.e. its own attributes, the attributes of its attributes, etc). The notation $\bar{e}$ represents the security type of an expression e, defined as $\bigsqcup_{v \in var(e)} \bar{v}$, i.e. the security type of an instance variable is defined based on the security types of all its attributes.

We define an abstract security semantics for our language in terms of boolean symbolic control flow graphs partially shown in Fig. 4. We abstract away the program variables in this semantics and only consider the program control flow in addition to the variables' security types. We assign a unique abstract boolean variable called *a branch variable* to each branch that denotes if that branch is enabled or not. A loop body might change the loop guard, and subsequently, the value of its branch variable might change in each iteration. Since, we don't model the program variables and consequently the loop body behaviour, we consider an uncontrollable boolean input called *uncontrollable loop guard* for each loop and each of its internal branches that non-deterministically takes a boolean value in each state and is assigned to the corresponding branch variable after execution of the loop body.

Let $\mathcal{G} = \langle L, V, I, l_o, v_0, \Delta \rangle$ represent a SCFG that shows the security semantics of a program where Δ is defined using the rules in Fig. 4. The locations L are the set of configurations where a configuration is defined as a stack $\sigma_0 : \dots : \sigma_n$ of currently active contexts. A context σ_k, $0 \leq k \leq n$ shows the statements of a method body that remain to be executed or a block of instructions (e.g. loop body), and pc_{σ_k} shows the security type of the context σ_k. The state variables V include the branch variables, the security types assigned to the program variables

c ::= o | **new** $m(\vec{e})$ | $o.m(\vec{e})$
stm ::= $v = e$ | $o = c$ | $o.m(\vec{e})$ | **if** (e) $stms$ [**else** $stms$] | **while** (e) $stms$ | **return** $[e]$ | $\surd$
$stms$ ::= stm; $stms$ | stm;

Fig. 3. The statements syntax

and the set of variables representing whether two instance variables point to the same object or not. The uncontrollable inputs of I include the uncontrollable loop guards and τ that is a boolean variable associated with the non-checkpoint transitions, and its controllable inputs are boolean inputs associated with each checkpoint transition.

The rule ASSIGNL defines the semantics of a variable of primitive type where e is a method call free expression. The security type of v is modified to the upper bound of e's security level ($\bar{e}$) and the security level of current context pc_{σ_n}. To handle object aliasing in our pure boolean SCFG, for each two arbitrary object instance variables of the same type, we consider a boolean variable called *points-to variable* to indicate whether they point to the same object or not. The function alias returns a boolean variable to show if two instance variables are in aliasing relation or not, where for all o, o', $\mathsf{alias}(o, o') = \mathsf{alias}(o', o)$. When an instance variable is updated, the points-to variables in addition to the security types of the associated instance variables are updated. The rule ASSIGNO defines the semantics of an assignment where the assignee is not an attribute instance variable. This rule relates the assignee to the assigner and all the instance variables related to the assigner (i.e. UpdatePointsToVars sets their corresponding points-to variables), and changes the type of assignee to the upper bound of the assigner's type and pc_{σ_n}. It will update the security types of the attributes of instance variables newly related to the assigner (UpdateAttributesLabels) (more details in [16]).

The rule COND defines the semantics of conditional statements, and the rule WHILE1 defines the semantics of loops. In these rules, the function $mc(stms)$ shows the variables that might be modified by $stms$ and basically returns all left-hand side variables of the assignments in $stms$, and $[stms]$ indicates that the code $stms$ is executing under a branch. When the program enters a branch, a new context σ_{n+1} is created whose security type is defined as the upper bound of the current context security label (pc_{σ_n}) and the security label of e. In addition, the security labels of all variables of the unexecuted branch in the new context are updated in order to detect indirect implicit flows. The function $\chi(\sigma_0 : \ldots : \sigma_n)$ returns two unique branch variables, assigned to each branch from a configuration $\sigma_0 : \ldots : \sigma_n$. When a program exits a branch or finishes the execution of the loop body, the latest context is removed (the rule EXIT and the rule WHILE2). In addition, the branch variables of a loop body ($bv(c)$) are updated to their corresponding uncontrollable loop guard variables (LoopGuard the rule WHILE2).

$$\textsc{assignL}\ \frac{U = \{\bar{v} = \bar{e} \sqcup \mathrm{pc}_{\sigma_n}\}}{\langle \sigma_0 : \ldots : \sigma_n = \{v = e;\}\rangle \xrightarrow{\top, U} \langle \sigma_0 : \ldots : \sigma_n = \{\surd\}\rangle}$$

$$\textsc{assignO}\ \frac{\begin{array}{c}\neg\mathsf{Attribute}(o),\\ U = \{\bar{o} = \bar{o'} \sqcup \mathrm{pc}_{\sigma_n}, \mathsf{alias}(o, o') = \top\} \cup \mathsf{UpdatePointsToVars}(o, o') \cup \mathsf{UpdateAttributesLabels}(o, o')\end{array}}{\langle \sigma_0 : \ldots : \sigma_n = \{o = o';\}\rangle \xrightarrow{\top, U} \langle \sigma_0 : \ldots : \sigma_n = \{\surd\}\rangle}$$

$$\textsc{cond}\ \frac{\begin{array}{c}U_1 := \{\mathrm{pc}_{\{[c_1]\}} = \bar{e} \sqcup \mathrm{pc}_{\sigma_n}\} \cup \bigcup_{x \in mc(c_2)} \bar{x} = \bar{x} \sqcup \mathrm{pc}_{\sigma_n}\ ,\\ U_2 := \{\mathrm{pc}_{\{[c_2]\}} = \bar{e} \sqcup \mathrm{pc}_{\sigma_n}\} \cup \bigcup_{x \in mc(c_1)} \bar{x} = \bar{x} \sqcup \mathrm{pc}_{\sigma_n}\ ,\ (\phi_1, \phi_2) = \chi(\sigma_0 : \ldots : \sigma_n)\end{array}}{\begin{array}{l}\langle \sigma_0 : \ldots : \sigma_n = \{\textbf{if}\ (e)\ c_1\ \textbf{else}\ c_2\}\rangle \xrightarrow{\phi_1, U_1} \langle \sigma_0 : \ldots : \{\surd\} : \{[c_1]\}\rangle\\ \langle \sigma_0 : \ldots : \sigma_n = \{\textbf{if}\ (e)\ c_1\ \textbf{else}\ c_2\}\rangle \xrightarrow{\phi_2, U_2} \langle \sigma_0 : \ldots : \{\surd\} : \{[c_2]\}\rangle\end{array}}$$

$$\textsc{while1}\ \frac{U_1 := \{\mathrm{pc}_{\sigma_{n+1}} = \bar{e} \sqcup \mathrm{pc}_{\sigma_n}\}\ ,\ U_2 := \bigcup_{x \in mc(c)} \bar{x} = \bar{x} \sqcup \mathrm{pc}_{\sigma_n}\ ,\ (\phi_1, \phi_2) = \chi(\sigma_0 : \ldots : \sigma_n)}{\begin{array}{l}\langle \sigma_0 : \ldots : \sigma_n = \{\textbf{while}\ (e)\ c;\}\rangle \xrightarrow{\phi_1, U_1} \langle \sigma_0 : \ldots : \{\surd\} : \sigma_{n+1} = \{[c; \textbf{while}\ (e)\ c]\}\rangle\\ \langle \sigma_0 : \ldots : \sigma_n = \{\textbf{while}\ (e)\ c;\}\rangle \xrightarrow{\phi_2, U_2} \langle \sigma_0 : \ldots : \{\surd\}\rangle\end{array}}$$

$$\textsc{while2}\ \frac{U := \bigcup_{\phi_i \in bv(c)} \phi_i = \mathsf{LoopGuard}(\phi_i)}{\langle \sigma_0 : \ldots : \{stms\} : \{[\textbf{while}\ (e)\ c]\}\rangle \xrightarrow{\top, \emptyset} \langle \sigma_0 : \ldots : \{\textbf{while}\ (e)\ c;\}\rangle}$$

$$\textsc{exit}\ \frac{}{\langle \sigma_0 : \ldots : \{stms\} : \{[\surd]\}\rangle \xrightarrow{\top, U} \langle \sigma_0 : \ldots : \{stms\}\rangle}$$

$$\textsc{callNT}\ \frac{\mathsf{NonThirdParty}(m), U := \{\mathrm{pc}_{\sigma_{n+1}} = \mathrm{pc}_{\sigma_n}\}\ ,}{\langle \sigma_0 : \ldots : \sigma_n = \{v = o.m(\vec{e})\}\rangle \xrightarrow{\top, U} \langle \sigma_0 : \ldots : \{\overline{\textbf{return}\ v}\} : \sigma_{n+1} = \{body[\vec{e}/pr(m)]\}\rangle}$$

$$\textsc{return}\ \frac{}{\langle \sigma_0 : \ldots : \{\overline{\textbf{return}\ v};\} : \{\textbf{return}\ x;\}\rangle \xrightarrow{\top, \emptyset} \langle \sigma_0 : \ldots \{v = x;\}\rangle}$$

$$\textsc{callT}\ \frac{\mathsf{ThirdParty(m)}\ ,\ l = \bar{e_1} \sqcup \ldots \sqcup \bar{e_m} \sqcup \bar{o} \sqcup \mathrm{pc}_{\sigma_n}\ ,\ U_1 = \{\bar{v} = l\} \cup \bigcup_{0 \le i \le m} \bar{e_i} = l}{\begin{array}{l}\langle \sigma_0 : \ldots : \sigma_n = \{v = o.m(\vec{e})\}\rangle \xrightarrow{\vec{e}, U_1} \langle \sigma_0 : \ldots : \sigma_n = \{\surd\}\rangle\\ \langle \sigma_0 : \ldots : \sigma_n = \{v = o.m(\vec{e})\}\rangle \xrightarrow{\neg\vec{e}, \emptyset} \langle \sigma_0 : \ldots : \sigma_n = \{\surd\}\rangle\end{array}}$$

Fig. 4. The security control flow semantics

The rule CALLNT describes the security semantics of a non-third party public method invocation defined for a class of type t that creates a new context with the statements $body[\vec{e}/pr(m)]$ that is obtained by substituting the method parameters $pr(m)$ in the method body with the arguments $\vec{e}$. The return statement pops the context and populates the variable v with the return value x (the rule RETURN) where x is a variable. For third-party methods, we set the security labels of all pass-by-reference arguments and the caller to high, if the method is invoked with a high-sensitive argument or the caller is high-sensitive (rule CALLT). We assume that the caller has no static attribute.

Example 1. Figure 5(a) shows the simplified security control flow model of the while loop in Fig. 2 generated by our tool. In this figure, the conditions `WA41` and `NA41` are branch variables and `EWA41` and `ENA41` are uncontrollable loop guards.

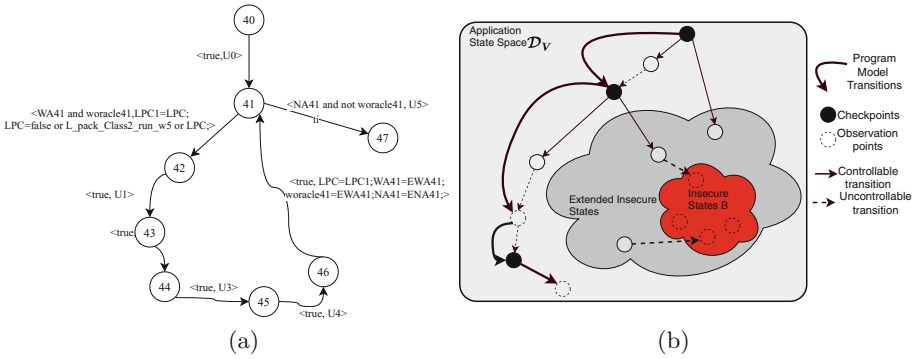

Fig. 5. (a) Security control flow model example; (b) Insecure state avoidance

Program Model. From the program semantics that is obtained by adding program variables to the security control flow semantics, we construct a program model that contains only the checkpoints and the observation points by merging the transitions (See Fig. 5(b)). We remove an unmonitorable transition t (i.e. its source is not a checkpoint or an observation point) by first propagating the transitions' guard and updates backwards to its incoming transitions, and then eliminating it. If there is no other transition from the source location of t, we remove the source location as well. The propagation continues until there is no further unmonitorable transition to process. We proved the soundness of the propagation algorithm [16].

5 Monitor Synthesis

The monitor synthesis process consists of two steps discussed in this section.

Step 1 - Generating Checkpoint Security Guards

A program is in an insecure state if it is in an observation point whose security policies have been violated, i.e. leaks information. An observation point is either a third-party method call, or the exit point of the unexecuted branch of a branch statement where the executed branch contains an observation point that is a method call. We consider the latter to be able to detect indirect information flows. For example, consider the following program where `print` is an observation point:

```
if(h>0) print(l0) else h=1;
```

If `h>0`, then the attacker observes `l0` in output and will know that `h` was greater than 0. If the else branch executes, since nothing is printed out, the attacker will know that `h<=0` holds. It is obvious that executing either of the branches causes information leakage. To prevent any leakage, we consider two

points in this program that must be avoided: `print(l)` that should always be called with low-sensitive data, and the outgoing transition of the else branch that should be in a low-sensitive context. Insecure states are formally specified as boolean expressions defined over security labels for the locations, e.g. $\neg\ \overline{\texttt{l0}}$ in the configuration `print(l0)` in the above example.

Given the (boolean) security control flow semantics described in Sect. 4 and the specification of insecure states, we use the boolean controller synthesis method described in Sect. 2 to obtain the abstract security guards (See Fig. 5(b)). An abstract security guard describes the execution paths and security types that lead to an insecure state. The guard of a checkpoint's transition is restricted to allow only execution paths that do not cause a security violation, and the insecure paths are controlled by applying countermeasures to avoid a violation. Observe that in the security control flow model, all the transitions from the checkpoints are considered controllable and the rest of the transitions are uncontrollable (Fig. 5(b)).

To obtain the security guards in terms of program variables, we propagate each branch guard along its path to its controlling checkpoint. For instance, in our example, the simplified generated guard for the checkpoint `run` is $\neg\ \overline{\texttt{ad}}\ \wedge \neg\texttt{WA41}$. To be able to evaluate this condition in the checkpoint, we propagate `WA41` to the checkpoint `run` that results in `0<(d+3)`.

If there is a conditional statement after the loop in our example, we cannot propagate its conditions to the checkpoint `run`, as we need to propagate the conditions through the loop which is not always possible. To solve this problem, we assume a dummy checkpoint after the loop body, called *loop checkpoint* that is used to propagate the conditions to, instead of the controlling checkpoint (e.g. the transition from 46 to 41 in Fig. 5(a)).

Step 2 - Monitor Construction

In the second step, we design a monitor to observe a program in the checkpoints and control the information flow. In the checkpoints, if the security guard of the current checkpoint, produced in the first step, allows the execution, the program will continue its execution and the monitor state will also be updated and evolved to the next checkpoint. Otherwise, a countermeasure will be applied to protect the program. One of the main countermeasures that the user can apply is to declassify the high-sensitive information to prevent reaching insecure states. Declassifying a variable leads to downgrading its security label.

We represent a program state by $\langle c, \boldsymbol{\nu}\rangle$ where c is the configuration and $\boldsymbol{\nu}$ indicates the program variables valuation. A monitor state is represented by $\langle \rho, \mathit{mode}, I, \mathrm{pc}, \Gamma\rangle$ where ρ is the current checkpoint of the monitor, *mode* is a variable that shows the monitoring mode (will be discussed later), I is the set of variables declassified so far, pc is the stack of security contexts, and the function Γ shows the valuation of security type variables. We represent the state of the monitored program by $\langle c, \boldsymbol{\nu}\rangle \parallel \langle \rho, \mathit{mode}, I, \mathrm{pc}, \Gamma\rangle$.

Let $\mathbb{C}$ be the set of checkpoint configurations, $\mathbb{L}$ be the set of observation point configurations, $\mathbb{P}$ be the set of security policies and $\rho \xrightarrow{G,A} \rho'$ represent

$$\text{NCP-SEC}\ \frac{\langle c, \boldsymbol{\nu}\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle\ ,\ c \notin \mathbb{C}\ ,\ c \notin \mathbb{L}}{\langle c, \boldsymbol{\nu}\rangle \parallel \langle \rho, mode, I, \mathrm{pc}, \Gamma\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle \parallel \langle \rho, mode, I, \mathrm{pc}, \Gamma\rangle}$$

$$\text{CP-INSEC1}\ \frac{\langle c, \boldsymbol{\nu}\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle, c \xrightarrow{G,A} \rho\ ,\ \boldsymbol{\nu} \models G\ ,\ (\boldsymbol{\nu}, \Gamma) \not\models \mathsf{Guard}(c)\ ,\ \neg\mathsf{Restrictive}(c, \boldsymbol{\nu}, \Gamma, \mathbb{C}, \mathbb{P})\ ,\ \mathsf{cmeasure}(\boldsymbol{\nu}, I) = \langle \boldsymbol{\nu}'', I'\rangle\ ,\ \Gamma' = \Gamma \downarrow (I' \backslash I)\ ,\ \mathsf{secure}(\mathsf{cmeasure})}{\langle c, \boldsymbol{\nu}\rangle \parallel \langle c, mode, I, \mathrm{pc}, \Gamma\rangle \rightarrow \langle c, \boldsymbol{\nu}''\rangle \parallel \langle c, mode, I', \mathrm{pc}, \Gamma'\rangle}$$

$$\text{CP-INSEC2}\ \frac{\langle c, \boldsymbol{\nu}\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle, c \xrightarrow{G,A} \rho\ ,\ \boldsymbol{\nu} \models G\ ,\ (\boldsymbol{\nu}, \Gamma) \not\models \mathsf{Guard}(c)\ ,\ \neg\mathsf{Restrictive}(c, \boldsymbol{\nu}, \Gamma, \mathbb{C}, \mathbb{P})\ ,\ \mathrm{pc}' = A(\mathrm{pc})\ ,\ \Gamma' = A(\Gamma)}{\langle c, \boldsymbol{\nu}\rangle \parallel \langle c, mode, I, \mathrm{pc}, \Gamma\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle \parallel \langle \rho, \top, I, \mathrm{pc}', \Gamma'\rangle}$$

$$\text{CP-INSEC3}\ \frac{\langle c, \boldsymbol{\nu}\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle, c \xrightarrow{G,A} \rho\ ,\ \boldsymbol{\nu} \models G\ ,\ (\boldsymbol{\nu}, \Gamma) \not\models \mathsf{Guard}(c)\ ,\ \mathsf{Restrictive}(c, \boldsymbol{\nu}, \Gamma, \mathbb{C}, \mathbb{P})\ ,\ \mathrm{pc}' = A(\mathrm{pc})\ ,\ \Gamma' = A(\Gamma)}{\langle c, \boldsymbol{\nu}\rangle \parallel \langle c, mode, I, \mathrm{pc}, \Gamma\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle \parallel \langle \rho, mode, I', \mathrm{pc}', \Gamma'\rangle\ \ \mathsf{Guard}(c) = \mathsf{Guard}(c) \wedge \neg path(c, \rho, \boldsymbol{\nu})}$$

$$\text{OP-LINSEC}\ \frac{\langle c, \boldsymbol{\nu}\rangle \rightarrow \langle c', \boldsymbol{\nu}\rangle\ ,\ \mathrm{pc} = \mathrm{pc}_1 : \ldots : \mathrm{pc}_{\sigma_n}\ ,\ \mathrm{pc}_{\sigma_n} = L\ ,\ c \neq \rho\ ,\ c \notin \mathbb{C}\ ,\ c \in \mathbb{L}}{\langle c, \boldsymbol{\nu}\rangle \parallel \langle \rho, \top, I, \mathrm{pc}, \Gamma\rangle \rightarrow \langle c'', \boldsymbol{\nu}\rangle \parallel \langle \rho, \top, I, \mathrm{pc}, \Gamma\rangle}$$

$$\text{OP-HINSEC}\ \frac{\mathrm{pc} = \mathrm{pc}_1 : \ldots : \mathrm{pc}_{\sigma_n}\ ,\ \mathrm{pc}_{\sigma_n} = H\ ,\ c \neq \rho\ ,\ c \notin \mathbb{C}\ ,\ c \in \mathbb{L}}{\langle c, \boldsymbol{\nu}\rangle \parallel \langle \rho, \top, I, \mathrm{pc}, \Gamma\rangle \rightarrow \langle \surd, \boldsymbol{\nu}\rangle \parallel \langle \rho, \top, I, \mathrm{pc}, \Gamma\rangle}$$

$$\text{CP-SEC}\ \frac{\langle c, \boldsymbol{\nu}\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle\ ,\ c \xrightarrow{G,A} \rho'\ ,\ \boldsymbol{\nu} \models G\ ,\ (\boldsymbol{\nu}, \Gamma) \models \mathsf{Guard}(c)\ ,\ \mathrm{pc}' = A(\mathrm{pc})\ ,\ \Gamma' = A(\Gamma)}{\langle c, \boldsymbol{\nu}\rangle \parallel \langle c, mode, I, \mathrm{pc}, \Gamma\rangle \rightarrow \langle c', \boldsymbol{\nu}'\rangle \parallel \langle \rho', \bot, I, \mathrm{pc}', \Gamma'\rangle}$$

Fig. 6. The behaviour of a monitored program

a symbolic transition from a checkpoint ρ to ρ' of the program model. The behavior of the monitored program is described by the rules in Fig. 6. The first rule states that if c is neither a checkpoint nor an observation point, then the program continues its normal execution. When a security violation is predicted in a checkpoint, we propose three general strategies for protection and the system administrator should apply the proper one to react to a security violation. We say a security violation is predicted in a checkpoint c in a state, if the propagated security guard generated for that checkpoint ($\mathsf{Guard}(c)$) is not satisfied in that state.

The guards generated in the first step can sometimes be restrictive. To check if a violation prediction is restrictive or not, we execute the program model up to the next checkpoint and check if the security policies have been violated along the path or not. If there is a violated security policy along the path, it means that the prediction is correct, otherwise, the security guard is restrictive for this specific path and must be relaxed. The predicate $\mathsf{Restrictive}(c, \boldsymbol{\nu}, \Gamma, \mathbb{C}, \mathbb{P})$ states that no security policy of $\mathbb{P}$ is violated in the states along the path from the program state $\langle c, \boldsymbol{\nu}\rangle$ to the next checkpoint.

When a violation is predicted, the monitor can apply a user-defined countermeasure $\mathsf{cmeasure}$ provided that this countermeasure is *secure* and the prediction is not restrictive (the rule cp-insec1 in Fig. 6). Let $\Gamma \downarrow V$ be a typing environment that degrades the security level of the variables of V in Γ. The countermeasure $\mathsf{cmeasure}$ should not change the value of the low-variables. In addition, it can only declassify variables that have not been modified by the program so far, i.e. $I' \backslash I \cap \boldsymbol{\nu}(mv) = \emptyset$ where $I' \backslash I$ is the set of declassified variables and mv is the set of variables modified so far. For instance, consider the following program:

```
h1=h2; f(); if (l1<10) {l2=h1;} else l2=l1; print(l2);
```

where l1 and l2 are low-sensitive, and h1 and h2 are high-sensitive. Let f() be the checkpoint and initially $\Gamma(\texttt{h1}) = \Gamma(\texttt{h2}) = H$. If we declassify h1 in the checkpoint, it also reveals h2. The reason is that the value of h1 is set to h2 before the checkpoint and if the **if** branch executes, h1 (and h2) will be copied to l2 that will be printed and revealed. Hence, we only allow declassification of variables that have not been modified. In addition, the variables declassified by applying a countermeasure shouldn't depend on the program state except for the program location. For instance, consider the following program

```
if(h3)  { h1=5;} else  h2=l1; f(); l=h1; print(l1);
```

If h3 is true, h1 becomes modified and we cannot declassify it. If h3 is false, even though h1 does not change, we do not allow it to be declassified, as it leads to the disclosure of h3 as well. Furthermore, the countermeasure should not lead the program into an insecure state again. Consider the program

```
f(); if(l1<10) {l2=h;} else l2=l1; print(l2);
```

If $\texttt{l1<10} \wedge \Gamma(\texttt{h}) = H$ holds in the checkpoint, the program is insecure, otherwise it's secure. As mentioned above, cmeasure cannot change any low-sensitive variable such as l1. Hence, a countermeasure that prevents the program from reaching an insecure state should include declassification of h, otherwise, $\texttt{l1<10} \wedge \Gamma(\texttt{h}) = H$ holds infinitely and this leads to a live lock situation where the program makes no progress and keeps constantly applying the same countermeasure. To avoid this situation, applying a countermeasure should lead to triggering a permissible transition, i.e. after applying the countermeasure, there should be a transition in the monitor that can be triggered.

Based on the above issues, a countermeasure cmeasure is secure, if for all $\boldsymbol{\nu}$ that $\mathsf{cmeasure}(\boldsymbol{\nu}, I) = \langle \boldsymbol{\nu}', I' \rangle$, (i) applying cmeasure does not lead the program into an insecure state, i.e. a transition from the location c in the monitor with a guard G' exists such that $\boldsymbol{\nu}' \models G'$, (ii) the condition $\boldsymbol{\nu} =_\Gamma \boldsymbol{\nu}' \wedge I' \cap \boldsymbol{\nu}'(mv) = \emptyset$ holds, and (iii) for all $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$, if $\mathsf{cmeasure}(\boldsymbol{\nu}_1, I) = \langle \boldsymbol{\nu}'_1, I'_1 \rangle$ and $\mathsf{cmeasure}(\boldsymbol{\nu}_2, I) = \langle \boldsymbol{\nu}'_2, I'_2 \rangle$, then $I'_1 = I'_2$. We say two memories $\boldsymbol{\nu}$ and $\boldsymbol{\nu}'$ are low-equal w.r.t. Γ, denoted by $\boldsymbol{\nu} =_\Gamma \boldsymbol{\nu}'$, if their low variables according to the security typing function Γ are identical, i.e. $\boldsymbol{\nu}(v) = \boldsymbol{\nu}'(v)$ where $\Gamma(v) = L$, $\forall v \in V$ and V is the set of program variables.

If a prediction about a violation is incorrect in a checkpoint c, the program will be allowed to execute and the security guard of the checkpoint ($\mathsf{Guard}(c)$) will be weakened (the rule cp-insec3). The function $path(c, \rho, \boldsymbol{\nu})$ returns the conditions in the state $\boldsymbol{\nu}$ that enable the path from c to ρ.

If the violation is predicted correctly but there is no countermeasure to apply in that checkpoint and all the future observation points up to the next checkpoints are side-effect free (i.e. return void), the execution mode is changed to *secure* ($mode = \top$) and a countermeasure is applied in the observation points, as done in [20] (the rule cp-insec2). The rule cp-sec states that if the program is

in a checkpoint, and the monitor allows its transition ($\boldsymbol{\nu} \models G$), then the monitor and the program evolve into their new states, and the monitoring mode changes to normal ($\perp$). In the secure mode execution, if the context is low and executing a statement in an observation point leads to a security policy violation, a default side-effect-free action c'' is performed, e.g. sending default data (the rule op-linsec), otherwise nothing happens (the rule op-hinsec). We assume that the observation points are side-effect free so that the countermeasures do not change the program semantics. The rules for the case that the learning feature is inactive are defined similarly.

In [16], we proved that a monitored program satisfies localized delimited release property [2], which states that, for any initial memory states s and s' whose secret parts may only differ, if the value of all declassified variables is the same in both s and s', then the observation sequence of the program running in state s and s' will be the same, or one is a prefix of the other. The reason for the latter case is that our method guarantees a termination-insensitive property. This notion disallows data release before it is declassified but allows release after declassification. In the case of no information release, it satisfies termination-insensitive non-interference.

6 Implementation and Evaluation

The Tool Set. We have implemented a tool to demonstrate the proposed method targeting Java applications. The tool consists of two main components: the static analysis component and the model execution engine. The static analysis requires the annotated Java application as input and (i) generates security guards for the checkpoints by employing the Reax [6] synthesis tool, (ii) automatically constructs the program model, and (iii) instruments the code for the monitoring purpose. The model execution engine executes symbolic control flow graphs and is used to run the program models.

Two versions of the monitor have been implemented. In the first version, we use the aforementioned engine to run the program model and train the monitor to eliminate false positives. In the entry point, the monitor initiates its state and loads the required information for it to function. On each of the checkpoints, the engine executes the program model until the next checkpoint, and checks if a violation has been predicted correctly. If the security guards of the current checkpoint are restrictive, it then relaxes the security guards.

In the second version, called model-execution free monitor, the program model is not executed and subsequently the monitor cannot learn new security guards. In this monitor, the security guards are checked at the checkpoints and the proper follow-up is executed if needed. If there is no violation, the security labels are updated to their values in the next checkpoint.

To assess the permissiveness of our method and the performance of tool, we applied it to a real world android application as well as multiple test cases of the Droidbench test suite. The application used is pedometer [1] with 1483 lines of code. The static experiments were performed on a Intel i7-6700 at 3.4 GHz and

32 GB of DDR4 Ram running a 64bit version of Ubuntu Linux. The dynamic experiments were performed on a Galaxy Tab S3 running android version 7.0.

We used 70 test cases from the Droidbench benchmark to evaluate the permissiveness of our method. We have achieved a precision of 100% and had 4(5%) false positives. The static analysis performance depends on the size of code, number of variables, the number of checkpoints and the average distance between them. The more checkpoints the program contains, the shorter the distance between the checkpoints and the more performant the static analysis usually should be. Figure 7(a) shows the performance results for static analysis of pedometer. That is mainly due to the guards being propagated along shorter paths when constructing the program model. The analysis of test cases in the Droidbench benchmark takes a fraction of second, as they are very small programs. Due to the small size of test cases in the Droidbench benchmark, it was not possible to have more than one checkpoint in a test case to evaluate the affect of number of checkpoints on the performance. In general, since we use *boolean* controller synthesis and state space partitioning to tackle complexity, we believe that static analysis should not be expensive, as confirmed by our current experiments so far.

The performance of the runtime monitor with learning feature is dependent on the number of the lines of code of the original program (See Fig. 7(b) for pedometer). For each instruction in the original program the monitor has to execute that instruction and update the security labels. Additionally the checkpoint guards have to be checked. As a result, we expect the runtime monitor to incur a significant performance overhead compared to the program with no monitor.

The monitor-execution free instance only checks the guards at each checkpoint and usually outperforms the runtime monitor. Its performance depends on the number of checkpoints; it sounds that the more the checkpoints the program has, the fewer checks have to be run at each one which improves performance. Note that the guards are propagated and simplified statically. An outside factor that seems to impact the monitor's performance is the JVM's optimization; when the checkpoints run many times, we noticed that the performance increases by at least an order of magnitude, e.g. from a 30% monitor running time to <1%.

Discussion. We believe that the results of static analysis are promising, mainly because the method uses boolean analysis and state partitioning. However, the performance overhead of dynamic monitor for our current test cases is scattered in quite a wide interval, e.g. from less than 1% to 40% for the model-execution free monitoring. We believe that we need to conduct many more experiments on different programs with various sizes, number of checkpoints, number of branches, number of variables etc, to be able to make a valid conclusion about the performance of the dynamic monitor. To this end, we should extend the method and tool to support exceptions, to be able to apply it on more real-life case studies. Furthermore, we are working on a new solution to run the monitor concurrently with the original program that is expected to improve the performance.

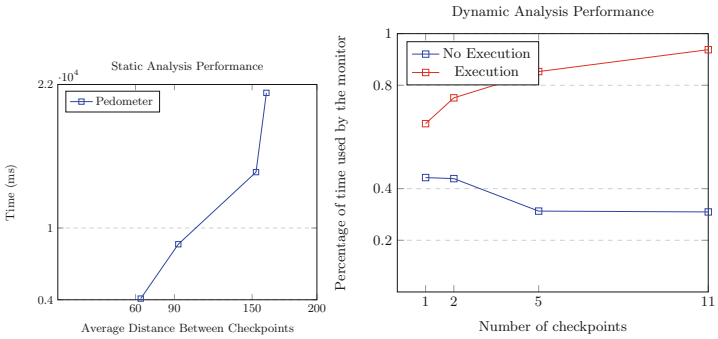

Fig. 7. Performance results

7 Related Work

There is a large body of work on verification and enforcement of noninterference as a policy to enforce confidentiality [13]. We have compared our approach with the related work in [16]. In this section, we discuss some related work.

The authors in [8] present a taxonomy of existing dynamic and hybrid monitors: no-sensitive-upgrade (NSU), permissive-upgrade (PU), hybrid monitor (HM), secure multi-execution (SME), and multiple facets (MF). The NSU [3,26] approach generates a purely dynamic monitor, that controls only one execution and disallows any upgrade of a low sensitive variable in a high context. This approach is improved in [4] by using a less-restrictive strategy in upgrading low variables in a high context, called permissive upgrade. In SME [11,17] and MF [5], multiple versions of a program are executed simultaneously, one for each security level, and the variable updates are controlled in a way that there will be no information leakage. These two categories of approaches introduce no information flow, however, they suffer from high performance overhead at runtime [5,12] that increases with the number of used security levels. Moreover, some repairable executions get blocked and the only applicable countermeasure is replacing the value of violating variables with some low-secure and safe constants.

In [9,14], the authors apply a flow-sensitive type system to instrument semantics of a program and consider unexecuted paths to detect indirect flows. Then, they statically construct a monitoring automaton that is traversed at runtime to detect security violations and apply countermeasures. In [20], the authors proposed a framework for hybrid monitors that is proven to be sound and guarantees termination insensitive noninterference for a simple language with output. It uses the countermeasures stop, suppress, or rewrite to react to a violation in output points. We extended their flow-sensitive type system with objects and method calls to instrument the program semantics. We predict violations at certain checkpoints which allows us to enforce a wider range of countermeasures at runtime to handle and resolve a security violation. Our "monitor mode" is

inspired from this work as well. Taint checking is another dynamic mechanism to control information flow, by tracking data dependencies as data is propagated in the system, that is well-surveyed in [23]. However, as it only tracks explicit flows [10] and ignores implicit flows, it enforces a weaker property than noninterference.

In contrast to the existing hybrid and dynamic monitors (e.g. [3,9,11,12,14, 14,17,20,24,26]), (i) our framework provides a learning feature that enables us to train the monitor and improve its permissiveness, (ii) it supports declassification and enforces localized delimited declassification while the existing monitors usually enforce a noninterference property, and (iii) we detect a violation in the checkpoints, in several steps before its occurrence, that allows us to enforce a wider range of countermeasures at runtime to protect against leakages. The main drawback of our method is its performance overhead that we are currently trying to improve by providing concurrent versions and optimizing the security guards.

8 Concluding Remarks

In this paper, we proposed an approach and its supporting tool for generating a hybrid security monitor for a subset of Java programs. This method synthesizes a sound symbolic monitor to predict undesired information flows and apply secure (user-defined) countermeasures to prevent information leakage and enforce localized delimited declassification. Given an annotated Java program, we implemented a tool-set to automatically generate a monitor. We also carried out some preliminary experiments to assess the method.

The results of our static analysis technique are promising in terms of both performance and the number of false positives. Hence, it can be used by the users to re-design their programs to fix information leakage problems at design time. In general, dynamic and hybrid monitors suffer from performance overhead [5,12], and so does our method. To improve its performance overhead, we are working on extending the method to support concurrent execution of monitors with the program, as well as simplifying the generated guards. We will also extend the supported sub-language of Java and conduct more experiments to evaluate the effectiveness of the tool properly.

References

1. Pedometer. https://f-droid.org/packages/name.bagi.levente.pedometer/
2. Askarov, A., Sabelfeld, A.: Localized delimited release: combining the what and where dimensions of information release. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, 14 June 2007, pp. 53–60 (2007)
3. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS 2009, New York, NY, USA, pp. 113–124 (2009)

4. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2010, New York, NY, USA, pp. 3:1–3:12. ACM, New York (2010)
5. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, New York, NY, USA, pp. 165–178. ACM, New York (2012)
6. Berthier, N., Marchand, H.: Discrete controller synthesis for infinite state systems with ReaX. In: 12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, 14–16 May 2014, pp. 46–53 (2014)
7. Besson, F., Bielova, N., Jensen, T.P.: Hybrid information flow monitoring against web tracking. In: 2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, 26–28 June 2013, pp. 240–254 (2013)
8. Bielova, N., Rezk, T.: A taxonomy of information flow monitors. In: Piessens, F., Viganò, L. (eds.) POST 2016. LNCS, vol. 9635, pp. 46–67. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49635-0_3
9. Dam, M., Le Guernic, G., Lundblad, A.: TreeDroid: a tree automaton based approach to enforcing data processing policies. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012, New York, NY, USA, pp. 894–905. ACM, New York (2012)
10. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
11. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, Berleley/Oakland, California, USA, 16–19 May 2010, pp. 109–124 (2010)
12. Desharnais, J., Kozyri, E., Tawbi, N.: Block-safe information flow control. Technical report, Cornell University (2016)
13. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 26–28 April 1982, pp. 11–20 (1982)
14. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-based confidentiality monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77505-8_7
15. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: 25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, 25–27 June 2012, pp. 3–18 (2012)
16. Khakpour, N., Skandylas, C.: Symbolic synthesis of a permissive security monitor: the extended version. Technical report, Linnaeus University (2018)
17. Kwon, Y., et al.: LDX: causality inference by lightweight dual execution. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, New York, NY, USA, pp. 503–515. ACM, New York (2016)
18. Le Guernic, G.: Confidentiality enforcement using dynamic information flow analyses. Ph.D. thesis, Manhattan, KS, USA (2007)
19. Pullicino, K.: Jif: language-based information-flow security in Java. CoRR, abs/1412.8639 (2014)
20. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, 17–19 July 2010, pp. 186–199 (2010)

21. Sabelfeld, A., Russo, A.: From dynamic to static and back: riding the roller coaster of information-flow control research. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 352–365. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11486-1_30
22. Santos, J.F., Rezk, T.: An information flow monitor-inlining compiler for securing a core of JavaScript. In: Proceedings of the ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, 2–4 June 2014, pp. 278–292 (2014)
23. Schoepe, D., Balliu, M., Pierce, B.C., Sabelfeld, A.: Explicit secrecy: a policy for taint tracking. In: IEEE European Symposium on Security and Privacy, Euro S&P 2016, Saarbrücken, Germany, 21–24 March 2016, pp. 15–30 (2016)
24. Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: 20th IEEE Computer Security Foundations Symposium (CSF 2007), Venice, Italy, 6–8 July 2007, pp. 203–217, July 2007
25. Simonet, V.: The flow caml system. Software release, vol. 116, pp. 119–156 (2003). http://cristal.inria.fr/~simonet/soft/flowcaml
26. Zdancewic, S.A.: Programming languages for information security. Ph.D. thesis, Ithaca, NY, USA (2002)