A Multi-compositional Enforcement on Information Flow Security

Cong Sun[1,2,3], Ennan Zhai[4], Zhong Chen[2,3], and Jianfeng Ma[1]

[1] Key Lab. of Computer Networks and Information Security, Xidian Univ., MoE, China
[2] Key Lab. of High Confidence Software Technologies, Peking Univ., MoE, China
[3] Key Lab. of Network and Software Security Assurance, Peking Univ., MoE, China
[4] Institute of Software, Chinese Academy of Sciences
suncong@xidian.edu.cn

Abstract. Interactive/Reactive computational model is known to be proper abstraction of many pervasively used systems, such as client-side web-based applications. The critical task of information flow control mechanisms aims to determine whether the interactive program can guarantee the confidentiality of secret data. We propose an efficient and flow-sensitive static analysis to enforce information flow policy on program with interactive I/Os. A reachability analysis is performed on the abstract model after a form of transformation, called multi-composition, to check the conformance with the policy. In the multi-composition we develop a store-match pattern to avoid duplicating the I/O channels in the model, and use the principle of secure multi-execution to generalize the security lattice model which is supported by other approaches based on automated verification. We also extend our approach to support a stronger version of termination-insensitive noninterference. The results of preliminary experiments show that our approach is more precise than existing flow-sensitive analysis and the cost of verification is reduced through the store-match pattern.

Keywords: Information flow security, pushdown system, interactive model, security policy, program analysis.

1 Introduction

Security-sensitive resources in computing system need to be protected from untrusted applications. Enforcing the information flow policies focuses on protecting confidentiality of these resources and ensures that attackers cannot learn any secret by observing the public behavior of multiple runs of program. Language-based techniques have been widely used for a long time in the studies on information flow security, surveyed in [16]. Goguen and Meseguer [9] introduce *noninterference* as the baseline confidential requirement to formalize the condition which enables secret system input to avoid being inferred by untrusted users of that system. Intuitively speaking, noninterference requires that the system behaviors should be indistinguishable from a perspective of attacker regardless of the confidential inputs to the system.

S. Qing et al. (Eds.): ICICS 2011, LNCS 7043, pp. 345–359, 2011.

There has been great progress on tracking information flow in languages with increasing complexity. Recently the community has paid increasing attention to the information flow security problem for languages with interactive I/Os, esp. through some forms of input/output channels [10,18,8,3,15,5,7]. In contrast to batch-job model which takes input before execution and generates output at termination, the computational model with interactive I/Os involves ongoing communications with external environment. It is a proper abstraction of various client-side web-based applications. Generally speaking, the interactivity can be characterized differently. In some existing efforts [15,5], the intermediate inputs can be decided by the previous outputs and this dependency is formally defined, e.g., through the user strategy [15], while in other models [18,8,7], the intermediate inputs can be considered simply as indefinite or abstracted as security levels. It means that compared with the first type of interactivity, the second type of interactivity can be preestablished and even totally indefinite throughout the execution. We mainly focus on programs with the second type of interactivity.

From a perspective of enforcement mechanisms, the approaches involving program with interactive I/Os can be either dynamic [10,7] or static based on type system [18,15,5] or abstract interpretation [8]. Automated verification has been used to check conformance with noninterference property on batch-job models [4,23,14]. This category of approach is flow-sensitive and commonly considered more precise than type system [4]. Because noninterference is not a safety property [23], it relies on program transformation, more specifically, some form of *self-composition* [4], to reduce noninterference on original model to a safety property on the model after transformation. With a requirement on duplicating the inputs and outputs, this approach will largely increase the state space of model and the cost of verification when it is adopted on interactive models. Automated verification is also restricted to support the simple security level lattice $L \preceq H$. Although in [23] the authors suggested to partition the memory stores multiple times to adopt more general lattices, there is no specific approach and it is still not clear how I/O channels leverage this category of partitioning approach.

In this work, we propose a novel approach using algorithmic verification technique to analyze information flow security on language with interactive I/Os. We utilize our previous idea to incorporate self-composition with reachability analysis [20] to check the conformance with noninterference property, and adopt our store-match pattern [21] to avoid duplicating I/O channels. Moreover, we adopt the idea of *secure multi-execution* proposed by Devriese et al. [7] to deal with complex lattice models. With secure multi-execution, the program is executed once on each security level. The outputs to channels with different security levels are handled respectively by each execution on the level. The high inputs of low execution are replaced by default value, while the low inputs of high execution are dependent on whether they have been obtained in the low execution. The secure multi-execution of noninterferent/secure program preserves the semantic soundness w.r.t. the normal execution. For interferent/insecure program, the multi-execution changes the semantics of original program by replacing interferent behaviors with noninterferent behaviors. This variation makes

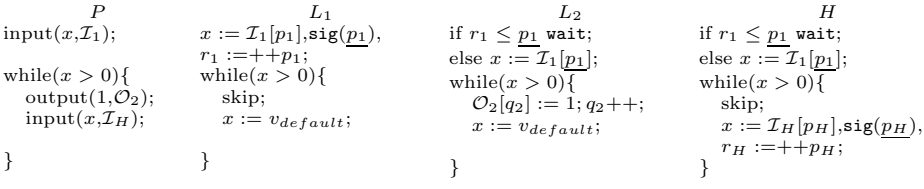

Fig. 1. Multi-Execution of Motivating Example

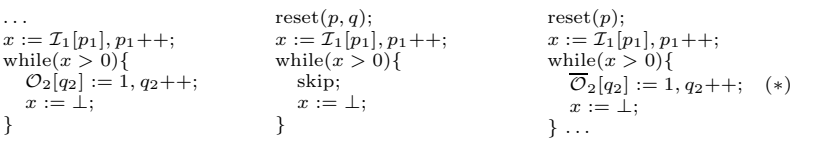

Fig. 2. Result of Composition

the normal execution and the multi-execution different from a perspective of observer on public output channels. That means when the normal execution and multi-execution behave diversely on public output channels, we can infer that the program violates noninterference. Our transformation composes the model w.r.t. normal execution with the model w.r.t. a serialized multi-execution using schedular $\mathsf{select}_{\text{lowprio}}$ [7, Sec.II.D], which stipulates that lowest security execution runs first.

1.1 Motivating Example

When we use reachability analysis instead of deduction on partial correctness judgements to check noninterference of program, e.g. $l := h$, the program can be transformed to

$$l' := l; l := h; l' := h'; \textbf{if } l' \neq l \textbf{ then } \text{goto } error;$$

Here the self-composition is evolved into three phases: basic self-composition, auxiliary interleaving assignments between initial low variables, and illegal-flow state construction. Consider program P in Fig.1 with a security lattice $L_1 \preceq L_2 \preceq H$, the outputs to channel $\mathcal{O}_2$ are dependent on the inputs from $\mathcal{I}_1$ and $\mathcal{I}_H$. This program is interferent because the inputs from $\mathcal{I}_H$ flow implicitly to $\mathcal{O}_2$ when the first input from $\mathcal{I}_1$ is greater than zero. In the multi-execution given in Fig.1, the outputs to $\mathcal{O}_2$ have turned to depend on default value, which we models as indefinite value. $sig(p_1)$ allows the thread L_2 and H to proceed after waiting for thread L_1 to obtain input from $\mathcal{I}_1$. Because thread H becomes noninterferent to the low outputs, from a perspective of low observer, we only need to serially model the low threads and compose the result with the model of normal execution. With a lowest-first scheduler, the input side-effect mentioned in [7] can be achieved by resetting the index of low channels and reusing the inputs each time a higher level execution starts, instead of using the *sig*/*wait* signal. The result of composition is given in Fig.2. The low inputs are used

$$e ::= v \mid x \mid e \oplus e'$$
$$c ::= \mathbf{skip} \mid x := e \mid c; c' \mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid \mathit{input}(x, \mathcal{I}_i) \mid \mathit{output}(e, \mathcal{O}_i)$$

Fig. 3. Program Syntax

$$\frac{}{(\mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{skip}; c) \rightarrow (\mu, \mathcal{I}, \mathcal{O}, p, q, c)} \qquad \frac{\mu(e) = v}{(\mu, \mathcal{I}, \mathcal{O}, p, q, x := e; c) \rightarrow (\mu[x \mapsto v], \mathcal{I}, \mathcal{O}, p, q, c)}$$

$$\frac{\mu(e) = b}{(\mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{if}\ e\ \mathbf{then}\ c_{\mathbf{true}}\ \mathbf{else}\ c_{\mathbf{false}}) \rightarrow (\mu, \mathcal{I}, \mathcal{O}, p, q, c_b)}$$

$$\frac{\mu(e) = \mathbf{true}}{(\mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{while}\ e\ \mathbf{do}\ c) \rightarrow (\mu, \mathcal{I}, \mathcal{O}, p, q, c; \mathbf{while}\ e\ \mathbf{do}\ c)}$$

$$\frac{\mu(e) = \mathbf{false}}{(\mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{while}\ e\ \mathbf{do}\ c) \rightarrow (\mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{skip})}$$

$$\frac{\mathcal{I}_i[p_i] = v \qquad p_i' = p_i + 1}{(\mu, \mathcal{I}, \mathcal{O}, p, q, \mathit{input}(x, \mathcal{I}_i); c) \rightarrow (\mu[x \mapsto v], \mathcal{I}, \mathcal{O}, p', q, c)}$$

$$\frac{\mu(e) = \mathcal{O}_i'[q_i] \qquad q_i' = q_i + 1}{(\mu, \mathcal{I}, \mathcal{O}, p, q, \mathit{output}(e, \mathcal{O}_i); c) \rightarrow (\mu, \mathcal{I}, \mathcal{O}', p, q', c)} \qquad \frac{(\mu, \mathcal{I}, \mathcal{O}, p, q, c_1) \rightarrow (\mu', \mathcal{I}', \mathcal{O}', p', q', c_1')}{(\mu, \mathcal{I}, \mathcal{O}, p, q, c_1; c_2) \rightarrow (\mu', \mathcal{I}', \mathcal{O}', p', q', c_1'; c_2)}$$

Fig. 4. Operational Semantics

multiple times to avoid the initial interleaving assignment, while the low output channels need to be duplicated, e.g. $\overline{\mathcal{O}}$ in Fig.2, to construct the illegal-flow state later. In order to avoid this duplication on low output channels, we match the low outputs generated in the serialized multi-execution with what generated and stored in the normal execution. For example we can substitute the command $(*)$ in Fig.2 with

$$\mathbf{if}\ \mathcal{O}_2[q_2] \neq 1\ \mathbf{then}\ \text{goto}\ \mathit{error};\ \mathbf{else}\ q_2{+}{+};$$

The state *error* is the target state of reachability analysis. This variation reduces the state space of model. If the program is secure, e.g. substituting $\mathit{input}(x, \mathcal{I}_H)$ with $\mathit{input}(x, \mathcal{I}_2)$ in program P, the state *error* will be unreachable. The approach also captures the flow from channel on L_2 to channel on L_1 within the same model, therefore it is different from a memory-partitioning approach.

The structure of the paper is as follows. Section 2 presents the program model and the information flow security property for program with interactive I/Os. In Section 3 we describe the reachability analysis based on the multi-composition and extend the approach to a stronger termination-insensitive noninterference. Section 4 shows the experimental studies for evaluating precision and performance improvement. We conclude in Section 5.

2 Program Model and Security Property

The presentation language is deterministic and the syntax is given in Fig.3. $\mathcal{I}$ and $\mathcal{O}$ are respectively the set of input and output channels. $\mathcal{I}$ maps each channel identifier i to a linear list, denoted by $\mathcal{I}_i$, and $\mathcal{O}$ is defined similarly.

The command $input(x, \mathcal{I}_i)$ indicates the sink of input from $\mathcal{I}_i$ is x, and $output(e, \mathcal{O}_i)$ stores the value of expression e in the correct position of $\mathcal{O}_i$. The small-step operational semantics of the presentation language are given in Fig.4. Here we assume the evaluation of expression is atomic and unambiguous. A *configuration* is a tuple $(\mu, \mathcal{I}, \mathcal{O}, p, q, c)$, where $\mu : Var \mapsto \mathbb{N}$ is a memory store mapping variables to values and c is the command to be executed. p and q are set of indices. p_i denotes the index of next element to be input from $\mathcal{I}_i$, and q_i is the index of location of $\mathcal{O}_i$ where the next value is stored. The elements of p and q are explicitly increased by the computation of inputs and outputs.

Although the confidential data of high inputs may be related to previous low outputs according to the specification of environment, it cannot be exposed in any form during the execution of program, especially through the subsequent low outputs. In order to define the security condition for noninterference property, we assume $\sigma : \mathcal{I} \cup \mathcal{O} \mapsto \mathcal{D}$ maps the I/O channels to security levels of the lattice $\mathcal{D}$. We suppose initial values of variables in μ are irrelevant to the definition of noninterference and initialized to indefinite. This is different from batch-job model which attaches security level on each variable and uses equivalence relation on low part of memory stores to semantically specify noninterference. We can specify certain variable with input from a special channel to obtain the flexibility of security level of variables. The indistinguishability relation on $\mathcal{I}$ and $\mathcal{O}$ w.r.t. certain security level ℓ is given as below.

Definition 1 (ℓ-indistinguishability). *For security level $\ell(\ell \in \mathcal{D})$, The* ℓ-indistinguishability *relation, denoted by $\sim_\ell$, is defined respectively on input and output channels of a program:*

1. $\mathcal{I} \sim_\ell \mathcal{I}'$, iff $\forall i : \sigma(\mathcal{I}_i) \preceq \ell \Rightarrow \mathcal{I}_i \sim_\ell \mathcal{I}'_i$
2. $\mathcal{O} \sim_\ell \mathcal{O}'$, iff $\forall i : \sigma(\mathcal{O}_i) \preceq \ell \Rightarrow \mathcal{O}_i \sim_\ell \mathcal{O}'_i$
where $\mathcal{I}_i \sim_\ell \mathcal{I}_j$ iff $(\sigma(\mathcal{I}_i) = \sigma(\mathcal{I}_j) \preceq \ell) \wedge (p_i = p_j \wedge \forall 0 \leq k < p_i : \mathcal{I}_i[k] = \mathcal{I}_j[k])$,
and $\mathcal{O}_i \sim_\ell \mathcal{O}_j$ iff $(\sigma(\mathcal{O}_i) = \sigma(\mathcal{O}_j) \preceq \ell) \wedge (q_i = q_j \wedge \forall 0 \leq k < q_i : \mathcal{O}_i[k] = \mathcal{O}_j[k])$.

For the two channels with same security level, the linear lists should have the same length and identical content. The content of channels with $\ell'(\ell' \succ \ell)$ is unobservable and irrelevant to the ℓ-indistinguishability as well as noninterference specification.

Then we specify the multi-execution of program with the simple lowest-first scheduler $\mathsf{select}_{\mathrm{lowprio}}$. This deterministic scheduler depends on a total order of the security lattice $\mathcal{D}$. If $\mathcal{D}$ is not a totally ordered lattice, because $\mathcal{D}$ is finite (the number of I/O channels is finite), a totally ordered extension $\hat{\mathcal{D}}$ of $\mathcal{D}$ always exists (see *linear extension*, 1.29,[6]). The extension can bring in additional legitimate channel for information leakage, e.g., if the example P in Sec.1.1 has a lattice $\{L_1 \preceq L_2, L_1 \preceq H, L_2 \preceq \top, H \preceq \top\}$ (H and L_2 are incomparable), P is judged secure under the extension $L_1 \preceq H \preceq L_2 \preceq \top$. To close this kind of channel from H to L_2, we require that each time we extend the lattice with $\ell_i \preceq \ell_j$, there should be $\exists \mathcal{I}_x \in \mathcal{I}.\sigma(\mathcal{I}_x) = \ell_j \vee \forall \mathcal{I}_x \in \mathcal{I}.\sigma(\mathcal{I}_x) \neq \ell_i$. That means ℓ_j should be attached to an input channel otherwise there are only outputs on ℓ_i. With the lowest-first scheduler, the secure multi-execution will be serialized

$$\frac{}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{skip}; c) \twoheadrightarrow (\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, c)} \qquad \frac{\mu(e) = v}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, x := e; c) \twoheadrightarrow (\hat{\mathcal{D}}, \mu[x \mapsto v], \mathcal{I}, \mathcal{O}, p, q, c)}$$

$$\frac{\mu(e) = b}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{if}\ e\ \mathbf{then}\ c_{\mathbf{true}}\ \mathbf{else}\ c_{\mathbf{false}}) \twoheadrightarrow (\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, c_b)}$$

$$\frac{\mu(e) = \mathbf{true}}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{while}\ e\ \mathbf{do}\ c) \twoheadrightarrow (\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, c; \mathbf{while}\ e\ \mathbf{do}\ c)}$$

$$\frac{\mu(e) = \mathbf{false}}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{while}\ e\ \mathbf{do}\ c) \twoheadrightarrow (\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{skip})}$$

$$\frac{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, c_1) \twoheadrightarrow (\hat{\mathcal{D}}', \mu', \mathcal{I}', \mathcal{O}', p', q', c_1')}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, c_1; c_2) \twoheadrightarrow (\hat{\mathcal{D}}', \mu', \mathcal{I}', \mathcal{O}', p', q', c_1'; c_2)}$$

$$\frac{\sigma(\mathcal{O}_i) \neq \hat{\mathcal{D}}_\perp}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathit{output}(e, \mathcal{O}_i); c) \twoheadrightarrow (\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, c)}$$

$$\frac{\sigma(\mathcal{O}_i) = \hat{\mathcal{D}}_\perp \qquad \mu(e) = \mathcal{O}_i'[q_i] \qquad q_i' = q_i + 1}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathit{output}(e, \mathcal{O}_i); c) \twoheadrightarrow (\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}', p, q', c)}$$

$$\frac{\hat{\mathcal{D}}_\perp \prec \sigma(\mathcal{I}_i)}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathit{input}(x, \mathcal{I}_i); c) \twoheadrightarrow (\hat{\mathcal{D}}, \mu[x \mapsto \perp], \mathcal{I}, \mathcal{O}, p, q, c)}$$

$$\frac{\sigma(\mathcal{I}_i) \preceq \hat{\mathcal{D}}_\perp \qquad \mathcal{I}_i[p_i] = v \qquad p_i' = p_i + 1}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathit{input}(x, \mathcal{I}_i); c) \twoheadrightarrow (\hat{\mathcal{D}}, \mu[x \mapsto v], \mathcal{I}, \mathcal{O}, p', q, c)}$$

$$\frac{\hat{\mathcal{D}} \neq \{\hat{\mathcal{D}}_\perp\} \qquad \hat{\mathcal{D}}' = \hat{\mathcal{D}} \setminus \{\hat{\mathcal{D}}_\perp\} \qquad \forall i : p_i' = 0}{(\hat{\mathcal{D}}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{skip}) \twoheadrightarrow (\hat{\mathcal{D}}', \mu, \mathcal{I}, \mathcal{O}, p', q, P)} \qquad \frac{}{(\{\hat{\mathcal{D}}_\perp\}, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{skip}) \twoheadrightarrow (\emptyset, \mu, \mathcal{I}, \mathcal{O}, p, q, \mathbf{skip})}$$

Fig. 5. Semantics of Serialized SME with $\mathsf{select}_{\mathrm{lowprio}}$

in that an execution on higher level should start after the executions on lower levels finished. For this special case we do not need to emit any signal between executions as a general scheduler does in [7], because the value should have been read by the execution on $\sigma(\mathcal{I}_i)$ when the execution on ℓ wants to read a value from $\mathcal{I}_i$ where $\sigma(\mathcal{I}_i) \preceq \ell$, otherwise the execution will be stuck and we do not need to weak up the waiting queue since it is always empty.

The formal semantics of the serialized secure multi-execution is given in Fig.5. Here $\twoheadrightarrow$ represents the secure multi-execution relation. Each configuration is extended with the total order lattice $\hat{\mathcal{D}}$. $\hat{\mathcal{D}}_\perp$ is the lower bound of $\hat{\mathcal{D}}$. $\hat{\mathcal{D}}$ shrinks during the execution. Each time an execution on $\hat{\mathcal{D}}_\perp$ is finished and $\hat{\mathcal{D}}_\perp$ is not the upper bound of the current $\hat{\mathcal{D}}$, $\hat{\mathcal{D}}_\perp$ is excluded from $\hat{\mathcal{D}}$ and an execution on a higher level is launched, otherwise $\hat{\mathcal{D}}$ reduces to $\emptyset$ and the execution terminates, see the last two rules of Fig.5 for details. The most obvious difference with common secure multi-execution [7] is the absence of global input pointer r. This will influence the manner of inputs from channel $\mathcal{I}_i$ on security level ℓ where $\sigma(\mathcal{I}_i) \prec \ell$. For the common secure multi-execution, when the input from a lower-level channel is allowed, the local input pointer does not need to be increased because the boundary of current input is held by $r(i)$ and this input is modeled as shared use. But in our semantics the reuse of these inputs is achieved by resetting the index of next element for each input channel. Another difference is the termination behavior of execution. The common secure multi-execution permits some execution to be divergent (then the final L_f won't be $\emptyset$) while our serialized version requires that each execution terminates to launch the execution on a higher level. Based on this requirement, the noninterference property we

enforce is termination-insensitive. Let $\hat{\mathcal{D}}^{\ell}$ be the sublattice of $\hat{\mathcal{D}}$ consisting of all security levels lower or equal to ℓ. Noninterference is defined as follows.

Definition 2 (Noninterference). *Let* $(\mu, \mathcal{I}, \mathcal{O}, p, q, P) \rightarrow^{*} (\mu_f, \mathcal{I}, \mathcal{O}_f, p_f, q_f,$ ***skip***$)$ *be any normal execution on inputs* $\mathcal{I}$*, and* $(\hat{\mathcal{D}}^{\ell}, \mu', \mathcal{I}', \mathcal{O}', p', q', P) \twoheadrightarrow^{*} (\emptyset, \mu'_f, \mathcal{I}', \mathcal{O}'_f, p'_f, q'_f,$ ***skip***$)$ *be any multi-execution on inputs* $\mathcal{I}'$*. Program* P *is noninterferent w.r.t. security level* ℓ*, if for any* $\mathcal{I}$ *and* $\mathcal{I}'$*, we have* $\forall \ell' \preceq \ell . \mathcal{I} \sim_{\ell'} \mathcal{I}'$ *implies* $\mathcal{O}_f \sim_{\ell'} \mathcal{O}'_f$*.*

This definition is more specific than the normal noninterference in [7] since we specify that the two correlative executions are taken firstly as normal execution and secondly as multi-execution. Because for terminating runs of noninterferent program, the secure multi-execution produces the same low observable outputs with normal execution, when the low inputs are indistinguishable, we can identify the interferent program by observing whether the outputs are different. We do not concentrate on whether the outputs generated by multi-execution of interferent program are meaningful, but just use the technique as a judgement on the conformance of program with security property. The enforcement is static, based on reachability analysis of pushdown model derived by a multi-compositional transformation, which is introduced in detail below.

3 Multi-compositional Enforcement

Self-composition is a model transformation technique composing program and its variable-renamed copy in order to reduce noninterference to a safety property on finite computations of program after composition. The multi-compositional approach does not compose program with a variable-renamed copy, but composes it with a sequential program serially modeling the multi-execution on the low security levels. We adapt the store-match pattern [21] to the multi-compositional approach. With this technique, only low channels need to be modeled and no channel needs to be duplicated in the model after multi-composition.

3.1 Model Construction

The abstract model we use is symbolic pushdown system. A pushdown system is a stack-based state transition system whose stack contained in each state can be of unbounded length. It is a natural model for sequential program with procedures. Symbolic pushdown system is a compact representation of pushdown system encoding the variables and computations symbolically.

Definition 3 (Symbolic Pushdown System). *Symbolic Pushdown System is a triple* $\mathcal{P} = (\mathcal{G}, \Gamma \times \mathcal{L}, \Delta)$*.* $\mathcal{G}$ *and* $\mathcal{L}$ *are respectively the domain of global variables and local variables.* Γ *is the stack alphabet.* Δ *is the set of symbolic pushdown rules* $\{\langle\gamma\rangle \hookrightarrow \langle\gamma_1 \cdots \gamma_n\rangle(\mathcal{R}) \mid \gamma, \gamma_1, \cdots, \gamma_n \in \Gamma \wedge \mathcal{R} \subseteq (\mathcal{G} \times \mathcal{L}) \times (\mathcal{G} \times \mathcal{L}^n) \wedge n \leq 2\}$.

The relation $\mathcal{R}$ specifies the variation of abstract variables before and after a single step of symbolic execution directed by the pushdown rule. The stack symbols denote the flow graph nodes of program.

Table 1. Normal Model Construction

c	$\Phi(c, \gamma_j, \gamma_k)$
skip	$\{\langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle\ rt(\mu, \cdots)\}$
$x := e$	$\{\langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle\ (x' = e) \wedge rt(\mu \setminus \{x\}, \cdots)\}$
IF	$\{\langle\gamma_j\rangle \hookrightarrow \langle\gamma_\mathsf{t}\rangle\ rt(\mu, \cdots) \wedge e\} \cup \Phi(c_{\mathsf{true}}, \gamma_\mathsf{t}, \gamma_k) \cup$
	$\{\langle\gamma_j\rangle \hookrightarrow \langle\gamma_\mathsf{f}\rangle\ rt(\mu, \cdots) \wedge \neg e\} \cup \Phi(c_{\mathsf{false}}, \gamma_\mathsf{f}, \gamma_k)$
WHILE	$\{\langle\gamma_j\rangle \hookrightarrow \langle\gamma_\mathsf{t}\rangle\ rt(\mu, \cdots) \wedge e\} \cup \Phi(c_{\mathrm{body}}, \gamma_\mathsf{t}, \gamma_j) \cup$
	$\{\langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle\ rt(\mu, \cdots) \wedge \neg e\}$
$c_1; c_2$	$\Phi(c_1, \gamma_j, \gamma_{\mathrm{mid}}) \cup \Phi(c_2, \gamma_{\mathrm{mid}}, \gamma_k)$

The model construction of the commands other than I/O operations in Fig.3 is similar to the one in our previous work [19]. The abstract variable context with respect to certain stack symbol maps the abstract global and local variables to the value in $\mathcal{G}$ and $\mathcal{L}$. The model construction adds constraints to regulate each $\mathcal{R}$ of pushdown rule. The constraint is expressed with logical operation on abstract variables. The construction function Φ is presented in Table 1. Here rt means retainment on value of global variables and on value of local variables of the procedure locating the pushdown rule.

Then we explain how to construct the model for I/O operations. In our approach all channels are global. Because no variable can rely on an element of input channel except through an *input* command, we can simply omit the confidential channel $\mathcal{I}_i$ ($\sigma(\mathcal{I}_i) \succ \ell$) and model $input(x, \mathcal{I}_i)$ as

$$\langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle\ (x' = \bot) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots)$$

where $\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell$ are set of channels or indices with security level $\ell'(\ell' \preceq \ell)$. For $input(x, \mathcal{I}_i)$ where $\sigma(\mathcal{I}_i) \preceq \ell$, we need to model $\mathcal{I}_i$ explicitly and repeatedly use it in the model of multi-execution at level ℓ' where $\sigma(\mathcal{I}_i) \preceq \ell' \preceq \ell$:

$$\langle\gamma_j\rangle \overset{i}{\hookrightarrow} \langle\gamma_k\rangle\ (x' = \mathcal{I}_i[p_i]) \wedge (p_i' = p_i + 1) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell \setminus \{p_i\}, q^\ell)$$

Because there may be a certain thread in multi-execution with level $\ell'' \prec \sigma(\mathcal{I}_i)$ and we also want to identify the leakage to this lower level, we record the channel identifier i to distinguish different low threads in the multi-composition.

The output to confidential channel $\mathcal{O}_i$ with $\sigma(\mathcal{O}_i) \succ \ell$ can be substitute with a rule for **skip** since the confidential outputs do neither interfere with low part of subsequent states in normal execution nor interfere with the low threads in multi-execution. But for the public outputs on channel $\mathcal{O}_i$ with $\sigma(\mathcal{O}_i) \preceq \ell$, the command $output(e, \mathcal{O}_i)$ is modeled by the following pushdown rules

$$\begin{array}{ll} \langle\gamma_j\rangle \overset{i}{\hookrightarrow} \langle \mathrm{out}_e \gamma_k\rangle & (tmp' = e) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell) \wedge rt_2(\cdots) \\ \langle \mathrm{out}_x\rangle \hookrightarrow \langle\epsilon\rangle & rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell) \end{array}$$

Here *tmp* is a global variable. out_e and out_x are respectively the entry and exit node of flow graph of procedure *output*. rt_2 denotes retainment on value of local

variables of the caller of procedure f in $\langle \gamma_j \rangle \hookrightarrow \langle \mathtt{f}_{\mathrm{entry}} \gamma_k \rangle$. The channel identifier is also explicitly recorded when entering procedure *output*. The body of model of procedure *output* is vacuous and will be constructed with pushdown rules acting a store-match pattern by the following model transformation.

3.2 Model Transformation

We perform the model transformation, i.e. multi-composition, on pushdown system constructed in the last section. The result has two parts. The first part is the pushdown rules w.r.t. the normal execution, which are actually the result of model construction except for the body of procedure *output*. The second part is the pushdown rules w.r.t. the multi-execution on low security levels. According to Definition 2, the input sequences used on certain low channel by normal execution and multi-execution should be identical as the precondition of proposition. Moreover, according to the semantics of serialized secure multi-execution (see Fig.5), the indices of input channels should be reset at the beginning of each thread. Therefore on each level of multi-execution, we reset the indices of input channels to reuse the elements on these channels. The output sequences are treated in a store-match pattern. When an output to $\mathcal{O}_i$ with $\sigma(\mathcal{O}_i) \preceq \ell$ has been computed in the multi-execution, we compare it with the corresponding output in the normal execution instead of storing it. If they are not equal, we direct the symbolic execution to the illegal-flow state, which has only itself as the next state. The pushdown rule for the body of procedure *output* in the normal execution is given as Out_s in Table 2. In the thread of multi-execution with security level $\ell' = \sigma(\mathcal{O}_i)$, the Out_m rules in Table 2 are used as the model of the body of *output*. Note that Out_s and Out_m are parameterized by the identifier i of output channel. Out_m is also parameterized by the security level of thread, i.e. ℓ'. ξ is a rename function on the stack symbols for generating new flow graph nodes for each low thread of multi-execution. When the illegal-flow state is reached, the postcondition of noninterference is violated. The precondition is satisfied by reusing the public input channels therefore from the reachability of state *error* we can ensure the violation of noninterference without considering the relation on the subsequent outputs.

The multi-composition algorithm is given in Algorithm 1. It derives the final set of symbolic pushdown rules Δ' from Δ of the result of model construction. *LastTrans* returns the pushdown rule corresponding to the last return command of program. The pushdown rules modeling the multi-execution are in fact parameterized by the security level ℓ_0 of each thread. ℓ_1 is recorded to construct the entry to the next thread. In particular, the pushdown rules w.r.t. the inputs and outputs in each thread are related to both the identifier of channel and the security level of thread.

Theorem 1 (Correctness). *Suppose $\mathcal{D}$ is a finite security lattice. For a security level ℓ ($\ell \in \mathcal{D}$), $\mathcal{P} \triangleright ME(\mathcal{P})$ is the pushdown system generated by the Multi-Composition on the model of program P, if the state* error *of $\mathcal{P} \triangleright ME(\mathcal{P})$ is not reachable from any possible initial state, P is noninterferent w.r.t. ℓ.*

(The proof is sketched in our technical report [22].)

Table 2. Stuffer Rules for Multi-Compositions

Abbr.	Pushdown Rules
$Out_s(i)$	$\langle \text{out}_e \rangle \hookrightarrow \langle \text{out}_x \rangle \ (\mathcal{O}'_i[q_i] = tmp) \wedge (q'_i = q_i + 1) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell \setminus \{\mathcal{O}_i[q_i]\}, p^\ell, q^\ell \setminus \{q_i\})$
$Out_m(i, \ell')$	$\langle \xi(\ell', \text{out}_e) \rangle \hookrightarrow \langle error \rangle \ (\mathcal{O}_i[q_i] \neq tmp) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell)$
	$\langle \xi(\ell', \text{out}_e) \rangle \hookrightarrow \langle \xi(\ell', \text{out}_x) \rangle \ (\mathcal{O}_i[q_i] = tmp) \wedge (q'_i = q_i + 1) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell \setminus \{q_i\})$
$Out'_s(i)$	$\langle \text{out}_e \rangle \hookrightarrow \langle \text{out}_m \rangle \ (\mathcal{O}'_i[q_i] = tmp) \wedge (q'_i = q_i + 1) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell \setminus \{\mathcal{O}_i[q_i]\}, p^\ell, q^\ell \setminus \{q_i\})$
	$\langle \text{out}_m \rangle \hookrightarrow \langle end \rangle \ (q_i = len) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell)$
	$\langle \text{out}_m \rangle \hookrightarrow \langle \text{out}_x \rangle \ (q_i \neq len) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell)$
$Out'_m(i, \ell')$	$\langle \xi(\ell', \text{out}_e) \rangle \hookrightarrow \langle error \rangle \ (\mathcal{O}_i[q_i] \neq tmp) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell)$
	$\langle \xi(\ell', \text{out}_e) \rangle \hookrightarrow \langle \xi(\ell', \text{out}_m) \rangle \ (\mathcal{O}_i[q_i] = tmp) \wedge (q'_i = q_i + 1) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell \setminus \{q_i\})$
	$\langle \xi(\ell', \text{out}_m) \rangle \hookrightarrow \langle \xi(\ell', succ) \rangle \ (q_i = len) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell)$
	$\langle \xi(\ell', \text{out}_m) \rangle \hookrightarrow \langle \xi(\ell', \text{out}_x) \rangle \ (q_i \neq len) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell)$

3.3 Security Enforcement for Divergent Program

As illustrated in [23], the automated safety analysis based on the self-composition actually enforces a *termination-insensitive noninterference* [24] (TINI), which is weaker than *termination-sensitive noninterference* [25] (TSNI). TSNI requires when an execution terminates on some input, any correlative execution on the low indistinguishable inputs should also terminate, and both executions generate low indistinguishable outputs. In another word, the executions from two indistinguishable inputs should both terminate or both diverge to satisfy TSNI. On the contrary, TINI allows the termination behavior to leak information. When an execution terminates on some input, there may be some divergent execution on low indistinguishable input. It only validates the indistinguishability on low outputs when both executions terminate. As mentioned in Section 2, the serialized secure multi-execution requires each thread terminates to launch the thread on a higher security level. Therefore the noninterference specified in Definition 2 and enforced in the last section is termination insensitive.

Askarov el al. [3] have demonstrated that for language with I/Os a divergent run can possibly leak all secret. Nonterminating program, which does not violate batch-job termination-insensitive noninterference, should be judged insecure. In order to adapt this requirement on termination-insensitive noninterference, we have to explicitly terminate the normal execution of program and ensure the terminated execution has the same semantic effect as the original divergent execution. Suppose the length of I/Os is finite. The key is to find the upper bound UB of the length of outputs. The algorithm to check conformance with this stronger termination-insensitive noninterference is given in Algorithm 2. The model $\mathcal{P}_{len}$ is parameterized by the length of channel *len*. We find the upper bound of the length of output sequence through a stepwise exponential reduction and linear extension. When the normal execution generates the *len*-th output, we explicitly direct to state *end* to terminate the execution. $Out_s(i)$ has to be extended to $Out'_s(i)$ in Table 2. If the upper bound is smaller than *len*, the divergent run will not get to *end*. For terminated program, we direct the last transition of normal execution to an idle state *noend* in order to capture the upper bound of length of outputs. $\mathcal{P}_{len} \triangleright ME(\mathcal{P}_{len})$ is the model with pushdown rules generated by Algorithm 1. In the model the $Out_m(i, \ell')$ rules w.r.t. different security levels

Algorithm 1. Multi-Composition

$\Delta' \leftarrow \Delta \setminus LastTrans(\mathcal{P})$
for all $r \in \Delta$ **do** /*add connection from normal execution to multi-execution*/
  if $r.expr = \langle \gamma_j \rangle \hookrightarrow \langle \epsilon \rangle \wedge r = LastTrans(\mathcal{P})$ **then**
    $\Delta' \leftarrow \Delta' \cup \{\langle \gamma_j \rangle \hookrightarrow \langle \xi(Lowest(\hat{\mathcal{D}}^\ell), startConf(\mathcal{P})) \rangle \ Reset(p^\ell, q^\ell)\}$
  end if
end for
while $(\ell_0, \ell_1) \leftarrow LowestTwo(\hat{\mathcal{D}}^\ell) \wedge \ell_0 \preceq \ell$ **do** /*deal with model on security level ℓ_0*/
  $\hat{\mathcal{D}}^\ell \leftarrow \hat{\mathcal{D}}^\ell \setminus \{\ell_0\}$
  for all $r \in \Delta$ **do**
    if $r.expr = \langle \gamma_j \rangle \overset{i}{\hookrightarrow} \langle \mathrm{out}_e \gamma_k \rangle$ **then** /*meet an output on channel $\mathcal{O}_i$*/
      $\Delta' \leftarrow \Delta' \cup Out_s(i)$
      if $\sigma(\mathcal{O}_i) = \ell_0$ **then** /*the channel is on current security level*/
        $\Delta' \leftarrow \Delta' \cup \{\langle \xi(\ell_0, \gamma_j) \rangle \hookrightarrow \langle \xi(\ell_0, \mathrm{out}_e) \xi(\ell_0, \gamma_k) \rangle \ r.\mathcal{R}\} \cup Out_m(i, \ell_0)$
      else
        $\Delta' \leftarrow \Delta' \cup \{\langle \xi(\ell_0, \gamma_j) \rangle \hookrightarrow \langle \xi(\ell_0, \gamma_k) \ rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots) \rangle\}$
      end if
    else if $r.expr = \langle \gamma_j \rangle \hookrightarrow \langle \gamma_s \gamma_k \rangle$ **then** /*normal inter-procedural calls other than outputs*/
      $\Delta' \leftarrow \Delta' \cup \{\langle \xi(\ell_0, \gamma_j) \rangle \hookrightarrow \langle \xi(\ell_0, \gamma_s) \xi(\ell_0, \gamma_k) \rangle \ r.\mathcal{R}\}$
    else if $r.expr = \langle \gamma_j \rangle \overset{i}{\hookrightarrow} \langle \gamma_k \rangle$ **then** /*meet an input on channel $\mathcal{I}_i$*/
      if $\sigma(\mathcal{I}_i) \succ \ell_0$ **then** /*public input but confidential to the current security level*/
        $\Delta' \leftarrow \Delta' \cup \{\langle \xi(\ell_0, \gamma_j) \rangle \hookrightarrow \langle \xi(\ell_0, \gamma_k) \rangle \ (x' = \perp) \wedge rt(\mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots)\}$
      else
        $\Delta' \leftarrow \Delta' \cup \{\langle \xi(\ell_0, \gamma_j) \rangle \hookrightarrow \langle \xi(\ell_0, \gamma_k) \rangle \ r.\mathcal{R}\}$
      end if
    else if $r.expr = \langle \gamma_j \rangle \hookrightarrow \langle \gamma_k \rangle$ **then** /*normal intra-procedural transitions*/
      $\Delta' \leftarrow \Delta' \cup \{\langle \xi(\ell_0, \gamma_j) \rangle \hookrightarrow \langle \xi(\ell_0, \gamma_k) \rangle \ r.\mathcal{R}\}$
    else if $r \neq LastTrans(\mathcal{P})$ **then** /*normal return*/
      $\Delta' \leftarrow \Delta' \cup \{\langle \xi(\ell_0, \gamma_j) \rangle \hookrightarrow \langle \epsilon \rangle \ r.\mathcal{R}\}$
    else /*the last return, add connection to the next thread*/
      $\Delta' \leftarrow \Delta' \cup \{\langle \xi(\ell_0, \gamma_j) \rangle \hookrightarrow \langle \xi(\ell_1, startConf(\mathcal{P})) \rangle \ Reset(p^\ell)\}$
    end if
  end for
end while

Algorithm 2. TINI for divergent program

$len \leftarrow 2^{MAX}$;
while $\neg Reachable(\mathcal{P}_{len}, end) \wedge len \neq 0$ **do** /*exponential reduction for region of upper bound*/
  $len \leftarrow len/2$;
end while
$UB \leftarrow len$; $N \leftarrow len + 1$;
while $Reachable(\mathcal{P}_N, end) \wedge N < 2 \cdot len$ **do** /*linear extension to find the upper bound*/
  $UB \leftarrow N$; $N \leftarrow N + 1$;
end while
return $\neg \bigvee_{len=1}^{UB} Reachable(\mathcal{P}_{len} \triangleright ME(\mathcal{P}_{len}), error)$;

are extended to $Out'_m(i, \ell')$ in Table 2, where $\xi(\ell', succ)$ labels the last transition of thread on level ℓ' in the model of multi-execution. We can observe that the complexity is largely increased to $O((MAX + UB/2) \cdot C(\mathcal{P}) + UB \cdot C(\mathcal{P} \triangleright ME(\mathcal{P})))$ where $C(\mathcal{P})$ is the complexity of reachability problem introduced in [17].

4 Evaluation

We embed our implementation in the parser of Remopla [11] and use the model checker Moped [12] as the back-end black-box engine for reachability analysis. The experimental environment is 1.66GHz×2 Intel CPU/1GB RAM/Linux kernal 2.6.27-14-generic. We investigated the following research questions: (1) Is

the new approach as precise as other flow-sensitive static analysis, e.g. based on abstract interpretation [8]? (2) Does the store-match pattern really improve the performance of analysis compared with the common self-composition? (3) What is the real cost when we adapt the new approach to analyze termination-insensitive noninterference of divergent program? There are several factors that contribute to the experimental results to answer these questions. First is the length of channels. Because the low channels are semantically modeled in our approach, the increase on length of channels will also increase the size of BDDs as well as the state space of model. When we evaluate the efficiency of the new approach, we time the experimental results correlated with different lengths of channels. The second factor is the number of bits for each element of channels. The regions and operations on regions expressed with BDDs may require overall larger number of bits and state space to capture the behavior of concrete model.

To answer the questions, we first choose the test cases from related work. `P1`∼`P8` are from Fig.4 of [8] and `tax` is from Section 7 of [8]. `3_7`, `3_8` and `3_11` are respectively the example 3.7, 3.8 and 3.11 from [5]. `P0` is the motivating example in Section 1. A comparison of precision is given in Table 3. *NI* means the security of program w.r.t. the definition of noninterference. ✓means the program is noninterferent and × means interferent. `3_7`, `3_8` and `3_11` are judged by the definition of ID-security in [5]. *AI* means the analysis result using Iflow [13] and *SM* means the analysis result using our approach with store-match pattern. *LEN* is the length of channels used in our approach. Here we suppose all of the elements in channel are binary.

Table 3. Precision

Case	*From*	*NI*	*AI*	*SM(LEN)*	
				=1	≥2
`P0`	Sec.1	×	×	✓	×
`P1`	Fig.4,[8]	×	×	×	×
`P2`	Fig.4,[8]	×	×	×	×
`P3`	Fig.4,[8]	×	×	×	×
`P4`	Fig.4,[8]	×	×	×	×
`P5`	Fig.4,[8]	×	×	✓	×
`P6`	Fig.4,[8]	✓	✓	✓	✓
`P7`	Fig.4,[8]	✓	✓	✓	✓
`P8`	Fig.4,[8]	×	×	×	×
`tax`	Sec.7,[8]	×	×	✓	×
`3_7`	Ex3.7,[5]	×	×	×	×
`3_8`	Ex3.8,[5]	×	×	✓	×
`3_11`	Ex3.11,[5]	✓	×	✓	✓

We use static structure in `P8` and `tax` to model dynamical object allocation. `P1` and `P2` show respectively typical explicit and implicit flow. `P3` is verified insecure when we allow the boundary of input channel to be stored explicitly as output. At the end of the model of multi-execution we compare the boundary with the stored one and direct the state to *error* when they are not equal. `P4` shows the leakage caused by the length of public output sequence. This leakage is captured by our approach since we can reach a case when the outputs of multi-execution is longer than that of normal execution and some newly output 1 is unequal to an indefinite value not covered by the output of normal execution. `3_8` is similar to `P4`. Our approach cannot close the terminating channel therefore `3_11` is treated as noninterferent program. On the other hand, the approach based on abstract interpretation is termination-sensitive. It captures the leakage of `3_11` though the ID-security is termination-insensitive. Because in `P0`,`P5`,`tax` and `3_8` the

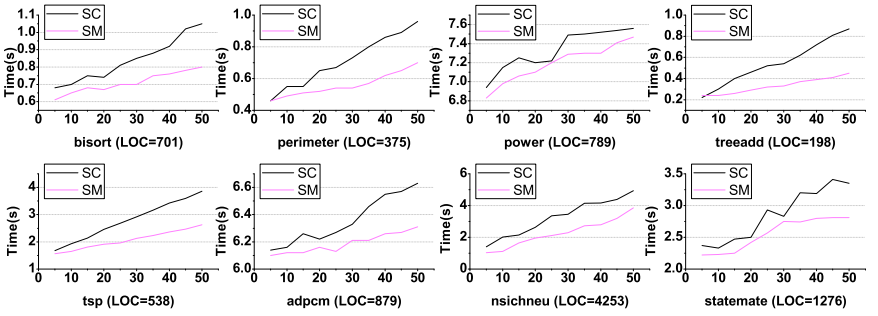

Fig. 6. Performance Improvement by Store-Match Pattern

leakages of confidential input are reflected by the second output, we can verify the program insecure only when $LEN \geq 2$. But with a sufficiently large boundary, our approach will not lose precision in these cases.

There are two limits on the precision of approach based on abstract interpretation. First, when the security lattice is more complex than $L \preceq H$, it has to partition the security lattice and the corresponding memory multiple times to capture all the leakages. For example, Iflow can capture the leakage of `P0` from H to L_2 when partitioning the lattice by L_2 (that means `P0` is the same as `P5`). But it cannot capture the leakage from L_2 to L_1 meanwhile with the same partitioning. Instead we need an individual model to capture this leakage. Second, although the both approaches are flow-sensitive, our approach can capture the value-dependent behavior of program while the approach based on abstract interpretation cannot. For example, although Iflow can judge `P7` as secure program, it will mistakenly treat the program `inH?x;y:=x-x;outL!y;` to be insecure. Another example can be found in [23, Fig.1]: When we use Iflow to analyze the program, x is abstracted to H and y is abstracted to L in the final state. The security level of the output channel is raised from L to H if l is the output and the program is conservatively rejected. On the other hand, our approach recording the value of computation can verify it secure. The program in [23, Fig.9] is a similar case. The results indicate that our approach is more precise than the analysis based on abstract interpretation to enforce termination-insensitive noninterference.

Then we evaluate the performance improvement achieved by the store-match pattern. With multi-execution, the common self-composition duplicates the low output channels and constructs the illegal-flow state following the model of multi-execution. Here we choose 8 C-programs: the first five are from the Olden benchmarks [1], while `adpcm`, `nsichneu` and `statemate` are from the WCET benchmarks [2]. We model these programs with Remopla. The system calls and library calls are treated as stubs. The standard I/Os are considered as the I/Os to the channels. The external values and random values are modeled to be indefinite. The confidential input channels are randomly selected. Fig.6 shows the experimental results. *SC* denotes the results of common self-composition.

The length of channels ranges from 5 to 50. The number of bits of each integer variable and element in channels is set to 2. The results indicate the store-match pattern, which avoids duplication of channels, can improve the performance of reachability analysis.

We choose `P0`, `P6`, `tax`, `3_8` and `3_11` to evaluate the reduction on efficiency caused by adapting our approach to analyze termination-insensitive noninterference of divergent program. We suppose the initial *MAX* is 5 in Algorithm 2. The experimental results are presented in Fig.7. `T_UB` is the time to derive the upper bound of length of outputs (corresponding to the cost of line 1∼8 of Algorithm 2), while `T_reach` is the time of conjunction of reachability analysis (corresponding to the cost of line 9 of Algorithm 2). `T0` is the time of reachability analysis proposed in Section 3.2. In `P6` and `3_11` the length of output is 1 therefore `T0` equals to `T_reach` in these cases. For terminating programs, the precision will not be lost when using Algorithm 2. The results also show that the cost to derive the upper bound *UB* becomes a major cost when *UB* is small. Another factor is *MAX*. It is clear that greater *MAX* will increase the cost to derive *UB*.

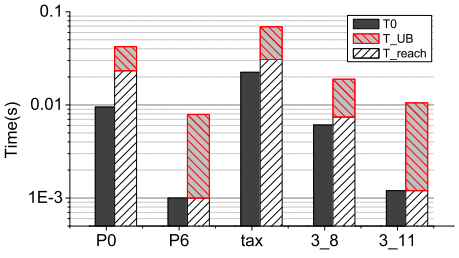

Fig. 7. Efficiency Reduction by divergence

5 Conclusion

We present an approach based on automated verification to analyze two different versions of termination-insensitive noninterference of program with interactive I/Os on more complex security lattice. A store-match pattern is used to reduce the state space of model. The precision of the approach and the effect of store-match pattern are evaluated.

Acknowledgments. This work is supported by the Major National S&T Program (2011ZX03005-002), the National Natural Science Foundation of China under grant Nos. 60821003,60872041,60970135, the Fundamental Research Funds for the Central Universities (JY10000903001), and GAD Advanced Research Foundation (9140A15040210HK6101).

References

1. The olden benchmark suite v1.0, `http://www.martincarlisle.com/olden.html`
2. The worst-case execution time (wcet) analysis project/benchmarks (2006), `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

3. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive Noninterference Leaks More Than Just a Bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008)
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW, pp. 100–114. IEEE (2004)
5. Bohannon, A., Pierce, B.C., Sjöberg, V., Weirich, S., Zdancewic, S.: Reactive noninterference. In: CCS, pp. 79–90. ACM (2009)
6. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press (2002)
7. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: IEEE Symposium on Security and Privacy, pp. 109–124 (2010)
8. Francesco, N.D., Martini, L.: Instruction-level security typing by abstract interpretation. Int. J. Inf. Sec. 6(2-3), 85–106 (2007)
9. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
10. Le Guernic, G., Banerjee, A., Jensen, T.P., Schmidt, D.A.: Automata-based Confidentiality Monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2008)
11. Holeček, J., Suwimonteerabuth, D., Schwoon, S., Esparza, J.: Introduction to remopla, `http://www.fmi.uni-stuttgart.de/szs/tools/moped/remopla-intro.pdf`
12. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: Moped: A model-checker for pushdown systems (2002), `http://www.fmi.uni-stuttgart.de/szs/tools/moped/`
13. Martini, L.: Iflow: a tool for information flow checking (2005), `http://www.iet.unipi.it/l.martini/iflow.html`
14. Naumann, D.A.: From Coupling Relations to Mated Invariants for Checking Information Flow. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 279–296. Springer, Heidelberg (2006)
15. O'Neill, K.R., Clarkson, M.R., Chong, S.: Information-flow security for interactive programs. In: CSFW, pp. 190–201. IEEE (2006)
16. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
17. Schwoon, S.: Model Checking Pushdown Systems. Ph.D. thesis, Technical University of Munich, Munich, Germany (2002)
18. Smith, S.F., Thober, M.: Improving usability of information flow security in java. In: PLAS, pp. 11–20. ACM (2007)
19. Sun, C., Tang, L., Chen, Z.: Secure information flow by model checking pushdown system. In: UIC-ATC, pp. 586–591. IEEE (2009)
20. Sun, C., Tang, L., Chen, Z.: Secure information flow in java via reachability analysis of pushdown system. In: QSIC, pp. 142–150. IEEE (2010)
21. Sun, C., Tang, L., Chen, Z.: A new enforcement on declassification with reachability analysis. In: INFOCOM Workshops, pp. 1024–1029. IEEE (2011)
22. Sun, C., Zhai, E., Chen, Z., Ma, J.: A multi-compositional enforcement on information flow security. Tech. rep., Institute of Software, School of EECS, Peking University (2011), `http://infosec.pku.edu.cn/~suncong/sun2011a-tr.pdf`
23. Terauchi, T., Aiken, A.: Secure Information Flow as a Safety Problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
24. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4(2/3), 167–188 (1996)
25. Volpano, D.M., Smith, G.: Eliminating covert flows with minimum typings. In: CSFW, pp. 156–169. IEEE (1997)