A New Enforcement on Declassification with Reachability Analysis

Cong Sun Liyong Tang Zhong Chen
Institute of Software, School of Electronics Engineering and Computer Science, Peking University, China
Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
Key Laboratory of Network and Software Security Assurance, Ministry of Education, China
Email: {suncong,tly,chen}@infosec.pku.edu.cn

***Abstract*—Language-based information flow security aims to decide whether an action-observable program can unintentionally leak confidential information if it has the authority to access confidential data. Recent concerns about declassification polices have provided many choices for practical intended information release, but more precise enforcement mechanism for these policies is insufficiently studied. In this paper, we propose a security property on the where-dimension of declassification and present an enforcement based on automated verification. The approach automatically transforms the abstract model with a variant of self-composition, and checks the reachability of illegal-flow state of the model after transformation. The self-composition is equipped with a store-match pattern to reduce the state space and to model the equivalence of declassified expressions in the premise of property. The evaluation shows that our approach is more precise than type-based enforcement.**

***Index Terms*—information flow security; declassification; push-down system; program analysis**

I. Introduction

Information flow security is concerned with finding new techniques to ensure that the confidential data will not be illegally leaked to the public observation. The topic is popular at both language level and operating system level. Language-based techniques have been pervasively adopted in the s-tudy on information flow security. This is comprehensively surveyed in [1]. Noninterference [2] is commonly known as the baseline property of information flow security. The semantic-based definition of noninterference [3] on batch-job model characterizes a security condition specifying that the system behavior is indistinguishable from a perspective of attacker regardless of the confidential inputs. Noninterference is criticized for the restriction that forbids any flow from *high* to *low*. It will influence the usability of system because the deliberate release is pervasive in many situations, e.g. password authentication, online shopping and encryption. Therefore, it is important to specify more relaxed and practical policies for real application scenarios and develop precise enforcement mechanisms for these policies.

The confidentiality aspect of information downgrading, i.e. declassification [4], allows information release with different intentions along four dimensions [5]: *what* is released, *where* does the release happen, *when* the information can be released and *who* releases it. The security policy we propose is on the where-dimension. On this dimension, there have been several polices, e.g. *intransitive noninterference* [6], *non-disclosure* [7], *WHERE* [8], *flow locks* [9], and *gradual release* [10]. Each of them leverages a certain category of type system to enforce the security policy.

In this work, we first use an approach based on automated verification to enforce declassification policy on the where-dimension. As a flow-sensitive and context-sensitive technique, automated verification has been used as an enforcement to noninterference on both imperative languages [11,12] and object-oriented languages [13,14]. In these works declassification is only discussed in [12], where the specific property *relaxed noninterference* [15] is mostly on the what-dimension.

The approaches based on automated verification usually rely on some form of self-composition [11] that composes the program model with a variable-renamed copy to reduce the security property on original model to a safety property on the model after transformation. In our previous work [14], we have developed a framework that uses reachability analysis to ease the specification of temporal logic formula or the manual assertion encoding partial correctness judgement. The self-composition doubles the size of memory store and largely increases the state space of model. When the I/O channels are considered, this effect becomes more serious since each store of channel is modeled explicitly. On the other hand, the security property often requires the equivalence of declassified expressions to be satisfied. Therefore in our enforcement we propose a store-match pattern to 1. avoid duplicating the output channels, and 2. facilitate the self-composition by modeling the equivalence of declassified expressions in the premise of security property. We also evaluated the similarity of the properties and the preciseness of our enforcement mechanism compared with type system.

The main contributions of the paper include: (i) We propose a more relaxed security property enforceable with automated verification on the where-dimension; (ii) We give a flow-sensitive and context-sensitive enforcement based on reachability analysis of pushdown system. We show the mechanism is more precise than type-based approaches; (iii) We propose a store-match pattern that can be in common use for automated verifications to reduce the state space of model and the cost of security analysis.

The rest of the paper is organized as follows. In Section II, we introduce the language model and the baseline property.

$$
\begin{aligned}
e ::=& v \mid x \mid e \oplus e' \\
C ::=& \textbf{skip} \mid x := e \mid x := \mathit{declass}(e) \mid \textbf{if } e \textbf{ then } C \textbf{ else } C' \mid \\
& \textbf{while } e \textbf{ do } C \mid C; C' \mid \mathit{input}(x, \mathcal{I}_i) \mid \mathit{output}(e, \mathcal{O}_i)
\end{aligned}
$$

Fig. 1. Program Syntax

$$
\frac{}{(\mu, \mathcal{I}, \mathcal{O}, p, q, \textbf{skip}; C) \rightarrow (\mu, \mathcal{I}, \mathcal{O}, p, q, C)}
$$

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

$$
\frac{\mu(e) = b}{(\mu, \mathcal{I}, \mathcal{O}, p, q, \textbf{if } e \textbf{ then } C_{\textbf{true}} \textbf{ else } C_{\textbf{false}}) \rightarrow (\mu, \mathcal{I}, \mathcal{O}, p, q, C_b)}
$$

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

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

$$
\frac{(\mu, \mathcal{I}, \mathcal{O}, p, q, C_1) \rightarrow (\mu', \mathcal{I}', \mathcal{O}', p', q', C_1')}{(\mu, \mathcal{I}, \mathcal{O}, p, q, C_1; C_2) \rightarrow (\mu', \mathcal{I}', \mathcal{O}', p', q', C_1'; C_2)}
$$

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

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

$$
\frac{\mu(e) = v \qquad \sigma(e) \preceq \sigma(x)}{(\mu, \mathcal{I}, \mathcal{O}, p, q, x := \mathit{declass}(e); C) \rightarrow (\mu[x \mapsto v], \mathcal{I}, \mathcal{O}, p, q, C)}
$$

$$
\frac{\mu(e) = v \qquad \sigma(x) \prec \sigma(e) \qquad \sigma(e) \rightsquigarrow \sigma(x)}{(\mu, \mathcal{I}, \mathcal{O}, p, q, x := \mathit{declass}(e); C) \rightarrow_d (\mu[x \mapsto v], \mathcal{I}, \mathcal{O}, p, q, C)}
$$

Fig. 2. Operational Semantics

In Section III, we define the where-security and prove the compliance of property with the prudent principles. Section IV describes the enforcement mechanism. We show the evaluation in Section V and conclude in Section VI.

II. Program Model and Baseline Property

We use a sequential imperative language with I/O channels as the presentation language to illustrate our approach. The syntax is listed in Fig.1. The language is deterministic. The primitive *declass* stands for declassification that downgrades the confidential data of expression e to be assigned to variable x with a lower security domain. Here x can be considered as a low-level sink of data observable to the attacker. $\mathcal{I}$ and $\mathcal{O}$ are respectively the set of input and output channels. They are formally defined as a mapping from each channel identifier i to a linear list, e.g. $\mathcal{I}_i$ resp. $\mathcal{O}_i$. The command $\mathit{input}(x, \mathcal{I}_i)$ indicates that the input from $\mathcal{I}_i$ is assigned to x, and the command $\mathit{output}(e, \mathcal{O}_i)$ stores the value of expression e into the correct position of $\mathcal{O}_i$.

The computation is modeled by the small-step operational semantics in Fig.2. The inductive rules are defined over configurations of the form $(\mu, \mathcal{I}, \mathcal{O}, p, q, C)$. $\mu : \mathit{Var} \mapsto \mathbb{N}$ 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 output value will be stored. The elements in p and q are explicitly increased by the computation of inputs and outputs.

The security policy is a tuple $(\mathcal{D}, \preceq, \rightsquigarrow, \sigma)$ where $(\mathcal{D}, \preceq)$ is a finite security lattice on security domains and $\rightsquigarrow$ is an exceptional downgrading relation of security domains ($\rightsquigarrow \cap \preceq = \emptyset$) statically gathered from the program. Let $\sigma : \mathit{Var} \cup \mathcal{I} \cup \mathcal{O} \mapsto \mathcal{D}$ be the mapping from I/O channels and variables to security domains, and let $\sigma(e) \equiv \bigsqcup_{x \in e} \sigma(x)$ be the least upper bound of the security domains of variables contained in e. When command $x := \mathit{declass}(e)$ in program has $\sigma(x) \prec \sigma(e)$, the *declass* operation performs a real downgrading from some variable in e and only then an element $(\sigma(e), \sigma(x))$ is contained in the relation $\rightsquigarrow$, otherwise the operation is identical to an ordinary assignment. We label the transition of declassification with $\rightarrow_d$ in Fig.2. The security policy is different from the MLS policy with exceptions proposed in [6,8,16], where the set of exceptional relations $\rightsquigarrow$ is independent to the declassification operations. In our policy the exceptions are gathered from the *declass* commands. Our treatment is reasonable since developer should have right to decide the exception when they use the primitive *declass* explicitly. This is also supported in other work, e.g. [17].

We specify noninterference with the semantic-based PER-model [3]. Intuitively speaking, it specifies a relation between states of any two correlative runs of program, which is variation in the confidential initial state cannot cause variation in the public final state. In another word, the runs starting from indistinguishable initial states derive indistinguishable final states as well. For the language with I/Os, the indistinguishability relation on memory stores and I/O channels with respect to certain security domain ℓ is defined as below.

Definition 1 (ℓ-indistinguishability)**.** *Memory store μ_i and μ_j are indistinguishable on $\ell (\ell \in \mathcal{D})$, denoted by $\mu_i \sim_\ell \mu_j$, iff $\forall x \in \mathit{Var}.\sigma(x) \preceq \ell \Rightarrow \mu_i(x) = \mu_j(x)$. For input channel $\mathcal{I}_i$ and $\mathcal{I}_j$, $\mathcal{I}_i \sim_\ell \mathcal{I}_j$ iff $(\sigma(\mathcal{I}_i) = \sigma(\mathcal{I}_j) \preceq \ell) \wedge (p_i = p_j \wedge \forall 0 \leq k < p_i.\mathcal{I}_i[k] = \mathcal{I}_j[k])$. Similarly, for output channel $\mathcal{O}_i$ and $\mathcal{O}_j$, $\mathcal{O}_i \sim_\ell \mathcal{O}_j$ 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 observable channels with same security domain, the indistinguishable linear lists should have the same length and identical content. Let $\mathcal{I}^\ell$ be the set of input channels with security domain $\ell' (\ell' \preceq \ell)$. If the set $\mathcal{I}$ and $\mathcal{I}'$ have the same domain, e.g. as the inputs of the same program, we can use $\mathcal{I} \sim_\ell \mathcal{I}'$ to express $\forall i.\mathcal{I}_i \in \mathcal{I}^\ell \Rightarrow \mathcal{I}_i \sim_\ell \mathcal{I}_i'$. The noninterference formalized here takes into consideration the I/O channels and is therefore different from what for batch-job model [1]. It is given as follows.

Definition 2 (Noninterference)**.** *Program P satisfies noninterference w.r.t. security domain ℓ_0, iff $\forall \ell \preceq \ell_0$, we have*

$$
\left(\begin{array}{l} \forall \mathcal{I}, \mu, \mathcal{I}', \mu', \mathcal{O}_f, \mu_f.(\mu, \mathcal{I}, \mathcal{O}, p, q, P) \rightarrow^* \\ (\mu_f, \mathcal{I}, \mathcal{O}_f, p_f, q_f, \textbf{\textit{skip}}) \wedge \mathcal{I} \sim_\ell \mathcal{I}' \wedge \mu \sim_\ell \mu' \end{array} \right) \Rightarrow
$$

$$\left(\begin{array}{l}\exists\mathcal{O}'_f,\mu'_f.(\mu',\mathcal{I}',\mathcal{O}',p',q',P)\rightarrow^*\\(\mu'_f,\mathcal{I}',\mathcal{O}'_f,p'_f,q'_f,\textbf{\textit{skip}})\wedge\mathcal{O}_f\sim_\ell\mathcal{O}'_f\wedge\mu_f\sim_\ell\mu'_f\end{array}\right).$$

In this definition, the noninterference property is related to a security domain ℓ_0. The content of channels with security domain $\ell'(\ell'\succ\ell_0)$ is unobservable and irrelevant to the property. A more specific way to define noninterference is to require $\ell_0=\bigsqcup\mathcal{D}$. That means the proposition in Definition 2 has to be satisfied for each security domain in $\mathcal{D}$. We use this definition in the following. Our definition adopts a manner to consider the indistinguishability of the initial and final states but not to characterize the relation in each computation step as did by the bisimulation-based approach [18]. Another use of the security domain of variables is to specify where a valid declassification occurs. This will be discussed below.

III. Where-Security and Prudent Principles

In this section, we give a security condition to control the legitimate release of confidential information on the where-dimension of security goals. It considers both the code locality where the release occurs and the level locality to which security domain the release is legal. Let $\twoheadrightarrow$ represent a (possible empty) sequence of declassification-free transitions. A trace of computations is separated to the declassifications labeled with $\rightarrow_d$ and declassification-free computation sequences. The *where-security* is formally specified as below.

Definition 3 (Where-Security)**.** *Program P satisfies* where-security *iff $\forall\ell\in\mathcal{D}$, we have*

$$\forall\mathcal{I},\mu,\mathcal{I}',\mu'.\exists n\geq 0:\left(\begin{array}{l}\forall\mathcal{O}_{n+1},\mu_{n+1}:(\mu,\mathcal{I},\mathcal{O},p,q,P)[\twoheadrightarrow(\mu_{k_s},\mathcal{I},\mathcal{O}_k,p_k,q_k,\\x_k:=\textit{declass}(e_k);P_k)\rightarrow_d(\mu_{k_t},\mathcal{I},\mathcal{O}_k,p_k,q_k,P_k)]_{k=1..n}\\\twoheadrightarrow(\mu_{n+1},\mathcal{I},\mathcal{O}_{n+1},p_{n+1},q_{n+1},\textbf{\textit{skip}})\\\wedge\mathcal{I}\sim_\ell\mathcal{I}'\wedge\mu\sim_\ell\mu'\end{array}\right)\Rightarrow$$

$$\left(\begin{array}{l}\exists\mathcal{O}'_{n+1},\mu'_{n+1}:(\mu',\mathcal{I}',\mathcal{O}',p',q',P)[\twoheadrightarrow(\mu'_{k_s},\mathcal{I}',\mathcal{O}'_k,p'_k,q'_k,\\x'_k:=\textit{declass}(e'_k);P'_k)\rightarrow_d(\mu'_{k_t},\mathcal{I}',\mathcal{O}'_k,p'_k,q'_k,P'_k)]_{k=1..n}\\\twoheadrightarrow(\mu'_{n+1},\mathcal{I}',\mathcal{O}'_{n+1},p_{n+1},q'_{n+1},\textbf{\textit{skip}})\\\wedge\bigwedge_{k=1..n}(\mu_{k_s}\sim_\ell\mu'_{k_s}\wedge\mu_{k_s}(e_k)=\mu'_{k_s}(e'_k)\Rightarrow\mu_{k_t}\sim_\ell\mu'_{k_t})\\\wedge\left(\begin{array}{l}\bigwedge_{k=1..n}(\mu_{k_s}(e_k)=\mu'_{k_s}(e'_k))\Rightarrow\\\mu_{n+1}\sim_\ell\mu'_{n+1}\wedge\mathcal{O}_{n+1}\sim_\ell\mathcal{O}'_{n+1}\end{array}\right)\end{array}\right)$$

Intuitively speaking, when the indistinguishable relation on the final states is violated, the contrapositive implies that it is caused by the variation of declassified expressions. This variation is indicated valid by the premise our property. If the leakage of confidential information is caused by a computation other than the primitive *declass*, it will be captured because without constraining the equality of released expression, the final indistinguishability cannot hold. Our where-security property is more relaxed than WHERE [8,16] which uses strong-bisimulation and requires each declassification-free computation step meets the baseline noninterference. We can use explicit final output of public variables to adapt the judgement of $\mu_{n+1}\sim_\ell\mu'_{n+1}$ to the judgement of $\mathcal{O}_{n+1}\sim_\ell\mathcal{O}'_{n+1}$.

Sabelfeld and Sands [5] clarify four basic prudent principles for declassification policies as sanity checks for the new definition: *semantic consistency*, *conservativity*, *monotonicity of release*, and *non-occlusion*. Our where-security property can be proved to comply with the first three principles. Let $P[C]$ represent a program contains command C. $P[C'/C]$ substitutes each occurrence of C in P with C'. The principles with respect to the where-security are defined as follows.

Lemma 1 (Semantic Consistency)**.** *Suppose C and C' are declassification-free commands and semantically equivalent on the same domain of configuration. If program $P[C]$ is where-secure, the $P[C'/C]$ is where-secure.*

Lemma 2 (Conservativity)**.** *If program P is where-secure and P contains no declassification, then P satisfies noninterference property.*

Lemma 3 (Monotonicity of Release)**.** *If program $P[x:=e]$ is where-secure, then $P[x:=\textit{declass}(e)/x:=e]$ is where-secure.*

Corollary 1. *The where-security satisfies semantic consistency, conservativity, and monotonicity of release.*

This corollary indicates that the where-security complies with the three prudent principles given by the above lemmas. The proofs of the lemmas are presented in [19]. The non-occlusion principle cannot be formally proved since a proof would require a characterization of secure information flow which is what we want to check against the prudent principles.

IV. Enforcement

In this section, we provide a new enforcement for the where-security based on reachability analysis of symbolic pushdown system [20]. A pushdown system is a stack-based state transition system whose stack contained in each state can be unbounded. It is a natural model of sequential program with procedures. Symbolic pushdown system is a compact representation of pushdown system encoding the variables and computations symbolically.

Definition 4 (Symbolic Pushdown System, SPDS)**.** *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 stack symbols denote the flow graph nodes of program. The relation $\mathcal{R}$ specifies the variation of abstract variables before and after a single step of symbolic execution directed by the pushdown rules. The operations on $\mathcal{R}$ are compactedly implemented with binary decision diagrams (BDDs) [21] in Moped [22] which we use as the back-end verification engine.

The model construction of commands other than I/O operations is similar to the one in our previous work [23]. In the pushdown system, the public channels are represented by global linear lists. In another word, for a security domain $\ell\in\mathcal{D}$, we only model the channels in $\mathcal{I}^\ell$ and $\mathcal{O}^\ell$. Take a input command for example, if the source channel is $\mathcal{I}_i$, the pushdown rule has a form of IR_H for $\sigma(\mathcal{I}_i)\succ\ell$ and IR_L for $\sigma(\mathcal{I}_i)\preceq\ell$ in Table I, where $\perp$ denotes an indefinite value.

TABLE I
PDS RULES FOR MODEL CONSTRUCTION

IR_H	$\langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle\ (x' = \bot) \wedge rt(\mu \setminus \{x\}, \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots)$
IR_L	$\langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle\ (x' = \mathcal{I}_i[p_i]) \wedge (p_i' = p_i + 1) \wedge rt(\mu \setminus \{x\}, \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell \setminus \{p_i\}, q^\ell, \cdots)$
OR_H	$\langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle\ rt(\mu, \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots)$
OR_L	$\langle\gamma_j\rangle \hookrightarrow \langle \mathit{output}_{\text{entry}}\gamma_k\rangle\ (\mathit{tmp}' = e) \wedge rt(\mu, \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots) \wedge rt_2(\cdots)$ $\langle \mathit{output}_{\text{exit}}\rangle \hookrightarrow \langle\epsilon\rangle\ rt(\mu, \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots)$
DR	$\langle\gamma_j\rangle \hookrightarrow \langle \mathit{declass}^{\gamma_j}_{\text{entry}}\rangle\ (\mathit{tmp}' = e) \wedge rt(\mu, \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots)$ $\langle \mathit{declass}^{\gamma_j}_{\text{exit}}\rangle \hookrightarrow \langle\gamma_k\rangle\ rt(\mu, \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell, \cdots)$

TABLE II
STUFFER PDS RULES FOR MODEL TRANSFORMATION

RST	$\langle\gamma_j\rangle \hookrightarrow \langle\xi(\gamma_0)\rangle\ (\forall p_i \in p^\ell.p_i' = 0) \wedge (\forall q_i \in q^\ell.q_i' = 0) \wedge rt(\mu, \xi(\mu), \mathcal{I}^\ell, \mathcal{O}^\ell, \cdots)$
OS_i	$\langle \mathit{output}_{\text{entry}}\rangle \hookrightarrow \langle \mathit{output}_{\text{exit}}\rangle\ (\mathcal{O}_i'[q_i] = \mathit{tmp}) \wedge (q_i' = q_i + 1) \wedge rt(\mu, \xi(\mu), \mathcal{I}^\ell, \mathcal{O}^\ell \setminus \{\mathcal{O}_i[q_i]\}, p^\ell, q^\ell \setminus \{q_i\}, \cdots)$
OM_i	$\langle\xi(\mathit{output}_{\text{entry}})\rangle \hookrightarrow \langle \mathit{error}\rangle\ (\mathcal{O}_i[q_i] \neq \mathit{tmp}) \wedge rt(\cdots)$ $\langle\xi(\mathit{output}_{\text{entry}})\rangle \hookrightarrow \langle\xi(\mathit{output}_{\text{exit}})\rangle\ (\mathcal{O}_i[q_i] = \mathit{tmp}) \wedge (q_i' = q_i + 1) \wedge rt(\mu, \xi(\mu), \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell \setminus \{q_i\}, \cdots)$
DS_{γ_j}	$\langle \mathit{declass}^{\gamma_j}_{\text{entry}}\rangle \hookrightarrow \langle \mathit{declass}^{\gamma_j}_{\text{exit}}\rangle\ (\mathcal{D}'[\rho(\gamma_j)] = \mathit{tmp}) \wedge (x' = \mathit{tmp}) \wedge rt(\mathcal{D} \setminus \{\mathcal{D}[\rho(\gamma_j)]\}, \mu \setminus \{x\}, \xi(\mu), \cdots)$
DM_{γ_j}	$\langle\xi(\mathit{declass}^{\gamma_j}_{\text{entry}})\rangle \hookrightarrow \langle \mathit{idle}\rangle\ (\mathcal{D}[\rho(\gamma_j)] \neq \mathit{tmp}) \wedge rt(\mathcal{D}, \cdots)$ $\langle\xi(\mathit{declass}^{\gamma_j}_{\text{entry}})\rangle \hookrightarrow \langle\xi(\mathit{declass}^{\gamma_j}_{\text{exit}})\rangle\ (\mathcal{D}[\rho(\gamma_j)] = \mathit{tmp}) \wedge (\xi(x)' = \mathit{tmp}) \wedge rt(\mathcal{D}, \mu, \xi(\mu) \setminus \{\xi(x)\}, \cdots)$

On the other hand, if the target channel of output is $\mathcal{O}_i$, the pushdown rule has a form of OR_H for $\sigma(\mathcal{O}_i) \succ \ell$ and OR_L for $\sigma(\mathcal{O}_i) \preceq \ell$ in Table I. OR_H is just like a transition of **skip** since the confidential outputs do not influence the public part of subsequent states. The variable *tmp* stores the value of expression to be outputted or declassified. rt means retainment on value of global variables and on value of local variables in $\langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle$. rt_2 for a rule $\langle\gamma_j\rangle \hookrightarrow \langle \text{f}_{\text{entry}}\gamma_k\rangle$ denotes retainment on value of local variables of the caller of procedure f. The declassifications are modeled with DR in Table I. The bodies of outputs to different public channel and the bodies of declassifications are vacuous. These absent parts of model will be filled by the self-composition. This treatment is decided by the store-match pattern which we develop to avoid the duplication of public channels and to guide the instrumented computation to fulfil the premise of where-security property.

We follow the principle of reachability analysis for noninterference which we proposed in [14]. The self-composition is evolved into three phases: basic self-composition, auxiliary initial interleaving assignments, and illegal-flow state construction. For simplicity, we use the *compact self-composition* [23] as basic self-composition. To avoid duplicating the input channels, we reuse the content of public input channels by resetting the indices of p^ℓ to 0 at the beginning of the pairing part of model, see RST in Table II. This treatment is safe because from the semantics we know that no computation actually modifies the content of input channels. In order to avoid duplicating the output channels, we propose a store-match pattern of output actions. This is to stuff the model after basic self-composition with the pushdown rules OS and OM in Table II parameterized with the channel identifier i. The OM rules show that when the output to channel $\mathcal{O}_i$ is computed in the second run, it is compared with the corresponding output stored during the first run. If they are not equal, the symbolic execution is directed to the illegal-flow state *error*.

Algorithm 1 Model Transformation

1. $\Delta' \leftarrow \{\langle\gamma_{\text{init}}\rangle \hookrightarrow \langle \mathit{startConf}(\mathcal{P})\rangle\ (\forall x \in \text{dom}(\mu^\ell).\xi(x)' = x) \wedge rt(\mu, \mathcal{I}^\ell, \mathcal{O}^\ell, p^\ell, q^\ell)\}$
2. **for all** $r \in \Delta \wedge r \neq \mathit{LastTrans}(\mathcal{P})$ **do**
3. $\quad\Delta' \leftarrow \Delta' \cup \{r.expr\ r.\mathcal{R} \wedge rt(\xi(\mu))\}$
4. **end for**
5. **for all** $r \in \Delta$ **do**
6. $\quad$**if** $r.expr = \langle\gamma_j\rangle \hookrightarrow \langle\gamma_s\gamma_k\rangle$ **then**
7. $\quad\quad\Delta' \leftarrow \Delta' \cup \{\langle\xi(\gamma_j)\rangle \hookrightarrow \langle\xi(\gamma_s)\xi(\gamma_k)\rangle\ r.\mathcal{R}^{\xi(x)}_{x \in \mathit{Var}} \wedge rt(\mu)\}$
8. $\quad$**else if** $r.expr = \langle\gamma_j\rangle \hookrightarrow \langle \mathit{declass}^{\gamma_j}_{\text{entry}}\rangle$ **then**
9. $\quad\quad\Delta' \leftarrow \Delta' \cup \{\langle\xi(\gamma_j)\rangle \hookrightarrow \langle\xi(\mathit{declass}^{\gamma_j}_{\text{entry}})\rangle\ r.\mathcal{R}^{\xi(x)}_{x \in \mathit{Var}} \wedge rt(\mu)\} \cup \text{DS}_{\gamma_j} \cup \text{DM}_{\gamma_j}$
10. $\quad$**else if** $r.expr = \langle\gamma_j\rangle \hookrightarrow \langle\gamma_k\rangle$ **then**
11. $\quad\quad\Delta' \leftarrow \Delta' \cup \{\langle\xi(\gamma_j)\rangle \hookrightarrow \langle\xi(\gamma_k)\rangle\ r.\mathcal{R}^{\xi(x)}_{x \in \mathit{Var}} \wedge rt(\mu)\}$
12. $\quad$**else if** $r \neq \mathit{LastTrans}(\mathcal{P})$ **then**
13. $\quad\quad\Delta' \leftarrow \Delta' \cup \{\langle\xi(\gamma_j)\rangle \hookrightarrow \langle\epsilon\rangle\ r.\mathcal{R}^{\xi(x)}_{x \in \mathit{Var}} \wedge rt(\mu)\}$
14. $\quad$**else**
15. $\quad\quad\Delta' \leftarrow \Delta' \cup \{\langle\xi(\gamma_j)\rangle \hookrightarrow \langle\xi(\gamma_j)\rangle\ r.\mathcal{R}^{\xi(x)}_{x \in \mathit{Var}} \wedge rt(\mu)\} \cup \{\langle\gamma_j\rangle \hookrightarrow \langle\xi(\mathit{startConf}(\mathcal{P}))\rangle\ \text{RST}\}$
16. $\quad$**end if**
17. **end for**
18. $\Delta' \leftarrow \Delta' \cup \bigcup_{\mathcal{O}_i \in \mathcal{O}^\ell} (\text{OS}_i \cup \text{OM}_i)$

Compared with the noninterference property, the premise of where-security contains equality relations on the declassified expressions, therefore we need some structure to instrument the semantics of abstract model to make sure the computation can proceed only when the equality relations are satisfied. We define another global linear list $\mathcal{D}$. Suppose there are m declassifications respectively at code location $\gamma_{d_i} (0 \leq i < m)$ and a function ρ mapping γ_{d_i} to i. We give another pattern of store-match that stores the value of expression declassified at γ_{d_i} to the site $\mathcal{D}[\rho(\gamma_{d_i})]$, see DS in Table II. The corresponding match operation has a form of DM in Table II. Note that ξ is the rename function on the stack symbols to generate new flow graph nodes as well as on the variables to generate the companion variables for the pairing part of model. The state *idle* has only itself as the next state. From the reachability of *error* we can ensure the violation of where-security without

TABLE III
DIFFERENCE BETWEEN PROPERTIES

	WHERE	*gradual release*	*where*
noninterference up-to	✓	✓	×
persistence	✓	×	×

considering the equality relations on the subsequent outputs. The self-composition algorithm is given in Algorithm 1. The *LastTrans* returns the pushdown rule with respect to the last return command of program. The first rule added to Δ' denotes the initial interleaving assignments from public variables to their companion variables. $r.\mathcal{R}^{\xi(x)}_{x\in Var}$ means a relation substituting each variable in *Var* with the renamed companion variable.

Theorem 1 (Correctness)**.** *Let $SC(\mathcal{P}^\ell)$ be the pushdown system w.r.t. security domain ℓ generated by our self-composition on the model of program P. If $\forall \ell \in \mathcal{D}$, the state* error *of $SC(\mathcal{P}^\ell)$ is unreachable from any initial state, we have P satisfies the where-security.*

(The proof is sketched in the technical report [19])

V. EVALUATION

We implement Algorithm 1 as part of the parser of Remopla [24] and use Moped as the black-box back-end engine for the reachability analysis. Here we use experiments to evaluate:

1. whether the property defined by where-security is similar to the existing properties on the where-dimension, e.g. [8,10], and what is the real difference between these properties.
2. the preciseness of the mechanism compared with the type systems on enforcing the respective security properties.
3. whether the store-match pattern can really reduce the state space as well as the cost of verification.

The experiments are performed on a laptop with 1.66GHz Intel Core 2 CPU, 1GB RAM and Linux kernel 2.6.27-14-generic. The test cases are chosen from related works, see Table IV.

Firstly, we illustrate that where-security is more relaxed than WHERE [8,16] and gradual release [10]. Lux and Mantel [16] have proposed another two prudent principles: *noninterference up-to* and *persistence*. Compared with the four basic principles, the two principles are not generally used for policies on different dimensions. The conformances of the properties with these principles are given in Table III. Similar to the gradual release, the program P1 in Table IV is secure (denoted by ✓) w.r.t. where-security. This indicates the two properties do not comply with persistence since the reachable command $l := h$ is obviously not secure. On the contrary, WHERE rejects this program. Our where-security does not comply with noninterference up-to because the definition deduces relations on final states but not on the states before *declass* primitives. A typical example is P0. It is where-secure but judged insecure by WHERE and gradual release. Although different on these special cases, the where-security can characterize a similar property to WHERE and gradual release for the most cases in Table IV, see the column *WHERE*, *GR* and *where*.

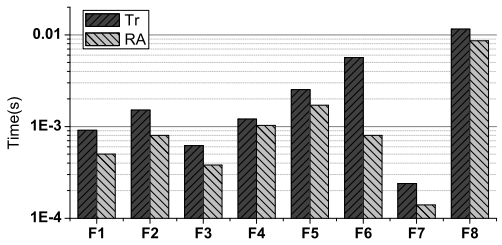

Fig. 3. Cost Reduction with Store-Match Pattern

Then we evaluate the preciseness of our enforcement mechanism. In Table IV τ_1 is the well-typeness of program judged by the type system in Fig.4, [8]. τ_2 is the judgement of the type system given in Fig.3, [10]. RA is the reachability analysis result using our mechanism. ✓means the state *error* is not reachable. The analysis time T is related to the number of bits of each variable, which we set to 3 and that means each variable in the model has a range of $0\sim2^3$-1. Larger number of bits corresponds to the increase on state space of model and the analysis time. On the other hand, the number of bits of variable is meaningful also because if it is too small for the model of insecure program, the illegal path cannot be caught. This causes a false-positive which can be avoided by setting the number of bits of variable sufficiently large. We record the minimum number of bits to avoid false-positive as N_{min}. The analysis might be time consuming when N_{min} is large. For secure program, the illegal-flow state will be unreachable for any number of bits therefore N_{min} is not recorded. The program *filter* in Table IV has a more complex policy. From the escape hatch information we have *reader* $\preceq$ *network*. The model is constructed and transformed on respective security domains. On each security domain different public variables are modeled outputted in the end and state *error* of transformed model is unreachable. Our enforcement is more precise compared with the type systems that reject some secure programs (P2,P6,P7 for WHERE and P1,P2,P6 for gradual release).

Finally, we evaluate the reduction on the cost of verification provided by the store-match pattern. We compare our mechanism with a model transformation, i.e. *Tr* in Fig.3, which duplicates the public output channels and constructs the illegal-flow state following the pairing part of model. The test cases containing I/Os are from Fig.4, [26], and named $F_1\sim F_8$ in Fig.3. These experiments show that the store-match pattern can give an overall 41.4% reduction on the cost of verification. The number of bits of variable is set to 3 as well.

VI. CONCLUSION

We propose a security property on the where-dimension of declassification. The property is proved complying with the three classical prudent principles. We also give a precise enforcement based on the reachability analysis of pushdown system derived by a variant of self-composition. To immigrate our approach to the properties on other dimensions of declassification, e.g. the *delimited release* [17] on the what-

TABLE IV
PROPERTY AND ENFORCEMENT COMPARISON WITH WHERE AND GRADUAL RELEASE

Case	From	*WHERE*	τ_1	*GR*	τ_2	*where*	RA	T(ms)	N_{min}
Ex2	Example 2, [6]	×	×	×	×	×	×	39.2	2
RSA	Example 5, [6]	×	×	×	×	×	×	1.09	1
C1	Example 1, [8]	×	×	×	×	×	×	0.55	1
C2	Example 1, [8]	✓	✓	✓	✓	✓	✓	0.59	–
C3	Example 1, [8]	✓	✓	✓	✓	✓	✓	0.49	–
filter	Fig.6, [8]	✓	✓	✓	✓	✓	✓	5.47	–
P0	Sec.1, [25]	×	×	×	×	✓	✓	0.44	–
P1	Sec.2, [10]	×	×	✓	×	✓	✓	0.53	–
P2	Sec.3, [25]	✓	×	✓	×	✓	✓	0.64	–
P3	Sec.2, [10]	×	×	×	×	×	×	3.53	1
P4	Sec.4, [25]	×	×	×	×	×	×	2.03	1
P5	Sec.4, [25]	×	×	×	×	×	×	0.61	1
P6	Sec.5, [25]	✓	×	✓	×	✓	✓	0.37	–
P7	Sec.2, [10]	✓	×	✓	✓	✓	✓	1.91	–

P0	$l := h; l := \mathit{declass}(h);$
P1	$l := \mathit{declass}(h); l := h;$
P2	$h_1 := h_2; l := \mathit{declass}(h_1);$
P3	$h_1 := h_2; h_2 := 0;$
	$l_1 := \mathit{declass}(h_2); h_2 := h_1; l_2 := h_2;$
P4	$h_2 := 0;$
	if h_1 then $l := \mathit{declass}(h_1)$
	else $l := \mathit{declass}(h_2);$
P5	$l := 0;$
	if l then $l := \mathit{declass}(h)$ else skip;
	$l := h;$
P6	$h_2 := 0;$
	if h_1 then $l := \mathit{declass}(h_2)$ else $l := 0;$
P7	$l := \mathit{declass}(h! = 0);$
	if l then $l_1 := \mathit{declass}(h_1)$ else skip;

dimension, the key point is to focus on the indistinguishability of declassified expressions on the pair of initial states. The study on the enforcement of properties on the other dimensions is left to our future work.

ACKNOWLEDGMENT

We thank Alexander Lux for providing the valuable proofs and explanations of the theorems in their work. We also thank Ennan Zhai for helpful comments and the anonymous reviewers for useful feedback. This research is partially supported by the National Natural Science Foundation of China under Grant No.60773163, No.60911140102, The National Key Technology R&D Program in the 11th five-year Period under Grant No.2008BAH33B01, as well as the PKU Project PKU-PY2010-005.

REFERENCES

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

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

[3] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," *Higher-Order and Symbolic Computation*, vol. 14, no. 1, pp. 59–91, 2001.

[4] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *SOSP*, 1997, pp. 129–142.

[5] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[6] H. Mantel and D. Sands, "Controlled declassification based on intransitive noninterference," in *APLAS*, ser. Lecture Notes in Computer Science, W.-N. Chin, Ed., vol. 3302. Springer, 2004, pp. 129–145.

[7] A. A. Matos and G. Boudol, "On declassification and the non-disclosure policy," *Journal of Computer Security*, vol. 17, no. 5, pp. 549–597, 2009.

[8] H. Mantel and A. Reinhard, "Controlling the what and where of declassification in language-based security," in *ESOP*, ser. Lecture Notes in Computer Science, R. D. Nicola, Ed., vol. 4421. Springer, 2007, pp. 141–156.

[9] N. Broberg and D. Sands, "Flow locks: Towards a core calculus for dynamic flow policies," in *ESOP*, ser. Lecture Notes in Computer Science, P. Sestoft, Ed., vol. 3924. Springer, 2006, pp. 180–196.

[10] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007, pp. 207–221.

[11] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *CSFW*. IEEE Computer Society, 2004, pp. 100–114.

[12] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *SAS*, ser. Lecture Notes in Computer Science, C. Hankin and I. Siveroni, Eds., vol. 3672. Springer, 2005, pp. 352–367.

[13] D. A. Naumann, "From coupling relations to mated invariants for checking information flow," in *ESORICS*, ser. Lecture Notes in Computer Science, D. Gollmann, J. Meier, and A. Sabelfeld, Eds., vol. 4189. Springer, 2006, pp. 279–296.

[14] C. Sun, L. Tang, and Z. Chen, "Secure information flow in java via reachability analysis of pushdown system," in *QSIC '10*. IEEE Computer Society, 2010, pp. 142–150.

[15] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *POPL*, J. Palsberg and M. Abadi, Eds. ACM, 2005, pp. 158–170.

[16] A. Lux and H. Mantel, "Who can declassify?" in *FAST*, ser. Lecture Notes in Computer Science, P. Degano, J. D. Guttman, and F. Martinelli, Eds., vol. 5491. Springer, 2008, pp. 35–49.

[17] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *ISSS*, ser. Lecture Notes in Computer Science, K. Futatsugi, F. Mizoguchi, and N. Yonezaki, Eds., vol. 3233. Springer, 2003, pp. 174–191.

[18] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *CSFW*, 2000, pp. 200–214.

[19] C. Sun, L. Tang, and Z. Chen, "A new enforcement on declassification with reachability analysis," Institute of Software, School of EECS, Peking University, Tech. Rep., 2010, http://infosec.pku.edu.cn/~suncong/sun2010d-tr.pdf.

[20] S. Schwoon, "Model checking pushdown systems," Ph.D. dissertation, Technical University of Munich, Munich, Germany, 2002.

[21] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[22] S. Kiefer, S. Schwoon, and D. Suwimonteerabuth, "Moped: A model-checker for pushdown systems," 2002, http://www.fmi.uni-stuttgart.de/szs/tools/moped/.

[23] C. Sun, L. Tang, and Z. Chen, "Secure information flow by model checking pushdown system," in *UIC-ATC '09*. IEEE Computer Society, 2009, pp. 586–591.

[24] J. Holeček, D. Suwimonteerabuth, S. Schwoon, and J. Esparza, "Introduction to remopla," 2006, http://www.fmi.uni-stuttgart.de/szs/tools/moped/remopla-intro.pdf.

[25] A. Askarov and A. Sabelfeld, "Localized delimited release: combining the what and where dimensions of information release," in *PLAS*, M. W. Hicks, Ed. ACM, 2007, pp. 53–60.

[26] N. De Francesco and L. Martini, "Instruction-level security typing by abstract interpretation," *Int. J. Inf. Sec.*, vol. 6, no. 2-3, pp. 85–106, 2007.