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 study
on information flow security. This is comprehensively sur-
veyed 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 crit-
icized for the restriction that forbids any flow from high to
low. It will influence the usability of system because the de-
liberate release is pervasive in many situations, e.g. password
authentication, online shopping and encryption. Therefore, it
is important to specify more relaxed and practical polices
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 tech-
nique, 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 declassifi-
cation 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 reacha-
bility 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.
e ::= v | e ⊕ e′
C ::= skip | x ::= e | x ::= declass(e) | if e then C else C′ |
while e do C | C; C′ | input(x, I), | output(e, O)

Fig. 1. Program Syntax

\[
\begin{align*}
\mu(e) &= v \\
\mu(I, O, p, q, C) &= (\mu, I, O, p, q, O)
\end{align*}
\]

\[
\begin{align*}
\mu(I, O, p, q, x := e; C) &= (\mu[x := v], I, O, p, q, C) \\
\mu(I, O, p, q, if e then Ctrue else Cfalse) &= (\mu, I, O, p, q, C_b)
\end{align*}
\]

\[
\begin{align*}
\mu(I, O, p, q, while e do C) &= (\mu, I, O, p, q, C; while e do C) \\
\mu(I, O, p, q, while do C) &= (\mu, I, O, p, q, skip)
\end{align*}
\]

\[
\begin{align*}
\mu(I, O, p, q, input(x, I),) &= (\mu[x := v], I, O, p′, q′, C′), \\
\mu(I, O, p, q, output(e, O); C) &= (\mu, I, O′, p′, q′, C′)
\end{align*}
\]

\[
\begin{align*}
\mu(I, O, p, q, x := \text{declass}(e); C) &= (\mu[x := v], I, O, p, q, C) \\
\mu(I, O, p, q, x := \text{declass}(e); C) &= (\mu[x := v], I, O, p, q, C);
\end{align*}
\]

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 level. Here x can be considered as a low-level sink of data observable to the attacker. I and 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. Ii resp. Oi. The command input(x, Ii) indicates that the input from Ii is assigned to x, and the command output(e, Oi) stores the value of expression e into the correct position of Oi.

The computation is modeled by the small-step operational semantics in Fig 2. The inductive rules are defined over configurations of the form \((\mu, I, O, p, q, C)\). \(\mu : \text{Var} \rightarrow \mathbb{N}\) 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 Ii, and \(q_i\) is the index of location of Oi 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 \((D, \preceq, \rightsquigarrow, \sigma)\) where \((D, \preceq)\) 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 \(\sigma : \text{Var} \cup \mathcal{I} \cup \mathcal{O} \rightarrow D\) be the mapping from I/O channels and variables to security domains, and let \(\sigma(e) \equiv \bigcup_{x \in e} \sigma(x)\) be the least upper bound of the security domains of variables contained in e. When command \(x := \text{declass}(e)\) 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 \(\ell\) is defined as below.

**Definition 1** (\(\ell\)-indistinguishability). Memory store \(\mu_i\) and \(\mu_j\) are indistinguishable on \(\ell (\ell \in D)\), denoted by \(\mu_i \sim_\ell \mu_j\), iff \(\forall x \in \text{Var}; \sigma(x) \preceq \ell \Rightarrow \mu_i(x) = \mu_j(x)\). For input channel \(I_i\) and \(I_j\), \(I_i \sim_\ell I_j\) iff \((\sigma(I_i) = \sigma(I_j)) \preceq \ell \land (p_i = p_j \land \forall x \preceq k \land p_i \rho_x \leq k \Rightarrow p_i[I_i[k] = I_j[k]]).\) Similarly, for output channel \(O_i\) and \(O_j\), \(O_i \sim_\ell O_j\) iff \((\sigma(O_i) = \sigma(O_j)) \preceq \ell \land (q_i = q_j \land \forall x \preceq k \land q_i[I_i[k] = I_j[k]]).\)

For the two observable channels with same security domain, the indistinguishable linear lists should have the same length and identical content. Let \(I^\ell\) be the set of input channels with security domain \(\ell (\ell \preceq \ell)\). If the set \(I\) and \(I^\ell\) have the same domain, e.g. as the inputs of the same program, we can use \(I \sim_\ell I^\ell\) to express \(\forall \mu, I_i \in I \Rightarrow I_i \sim_\ell I_i^\ell\). The noninterference formalized here takes into consideration the I/O channels and is therefore different from what for batch-job model [I]. It is given as follows.

**Definition 2** (Noninterference). Program P satisfies noninterference w.r.t. security domain \(\ell_0\), iff \(\forall x \preceq \ell_0\), we have \((\forall x \preceq \ell_0, \delta)\), have\( (\mu_f, \mu, I, O, p, q, P) \Rightarrow (\mu_f, \mu, I, O, p, q, I, \sim_\ell I^\ell \land \mu \sim_\ell \mu')\).
In this definition, the noninterference property is related to a security domain $\ell_0$. The content of channels with security domain $\ell'(\ell' \prec \ell_0)$ is unobservable and irrelevant to the property. A more specific way to define noninterference is to require $\ell_0 = \bigcup D$. That means the proposition in Definition 2 has to be satisfied for each security domain in $D$. 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 \cite{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 $\rightarrow_d$ 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 for all $\ell \in D$, we have

$$\forall I, \mu, I', \mu': \exists n \geq 0 \therefore
\begin{align*}
\exists \Omega_f, \mu_f', (\mu', I', \Omega', p', q', P) \rightarrow^* \\
(\mu_f', I', \Omega_f', p_f', q_f', \text{skip}) \land \Omega_f \sim \Theta \land \mu_f \sim \ell \mu_f'
\end{align*}
$$

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 $\text{declas}$, 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 \cite{8} 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 $\Omega_{n+1} \sim \ell \Omega_{n+1}$.

Sabelfeld and Sands \cite{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 $P[x := \text{declass}(e)]/x := e$ 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 \cite{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 \cite{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 $P = (G, \Gamma \times L, \Delta)$. $G$ and $L$ are respectively the domain of global variables and local variables. $\Gamma$ is the stack alphabet. $\Delta$ is the set of symbolic pushdown rules $\{\gamma \mapsto \text{declass}(\gamma)/(\gamma, \gamma, \cdots, \gamma_n) \in G \times L \times (G \times L^n) \land n \leq 2\}$.

The stack symbols denote the flow graph nodes of program. The relation $R$ specifies the variation of abstract variables before and after a single step of symbolic execution directed by the pushdown rules. The operations on $R$ are compactly implemented with binary decision diagrams (BDDs) \cite{21} in Moped \cite{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 \cite{21}. In the pushdown system, the public channels are represented by global linear lists. In another word, for a security domain $\ell \in D$, we only model the channels in $I^\ell$ and $\Omega^\ell$. Take an input command for example, if the source channel is $I_1$, the pushdown rule has a form of $IR_{\ell'} \sigma(I_1) > \ell$ and $IR_{\ell'} \sigma(\Omega) \leq \ell$ in Table I where $\perp$ denotes an indefinite value.


## Table I

| RST | $\langle \gamma_j \rangle \mapsto \langle \xi(\gamma_j) \rangle \ (\forall p_i \in p^i, p'_i = 0) \land (\forall q_j, q'_j, q''_j = 0) \land rt(\mu, \xi(\mu), T^i, O^i, \ell, \ldots)$ |
|-----|----------------------------------------------------------------------------------------------------------------------------------|
| ORH | $\langle \gamma_j \rangle \mapsto \langle \gamma_k \rangle \ (\forall r \in \mathcal{D}(\mu) \land rt(\mu, T^i, O^i, p^i, q^i, q''_i, \ldots)$ |
| ORL | $\langle \gamma_j \rangle \mapsto \langle \gamma_k \rangle \ (\forall r \in \mathcal{D}(\mu) \land rt(\mu, T^i, O^i, p^i, q^i, q''_i, \ldots)$ |
| ORM | $\langle \gamma_j \rangle \mapsto \langle \gamma_k \rangle \ (\forall r \in \mathcal{D}(\mu) \land rt(\mu, T^i, O^i, p^i, q^i, q''_i, \ldots)$ |
| ORP | $\langle \gamma_j \rangle \mapsto \langle \gamma_k \rangle \ (\forall r \in \mathcal{D}(\mu) \land rt(\mu, T^i, O^i, p^i, q^i, q''_i, \ldots)$ |

## Table II

| Table II Stuffer PDS Rules for Model Transformation |
|------------------------------------------|
| RST | $\langle \gamma_j \rangle \mapsto \langle \xi(\gamma_j) \rangle \ (\forall p_i \in p^i, p'_i = 0) \land (\forall q_j, q'_j, q''_j = 0) \land rt(\mu, \xi(\mu), T^i, O^i, \ell, \ldots)$ |
| OS | $\langle \text{output}_\text{entry} \rangle \mapsto \langle \text{output}_\text{entry} \rangle \ (\forall r \in \mathcal{D}(\mu) \land rt(\mu, T^i, O^i, p^i, q^i, q''_i, \ldots)$ |
| OM | $\langle \text{output}_\text{entry} \rangle \mapsto \langle \text{output}_\text{entry} \rangle \ (\forall r \in \mathcal{D}(\mu) \land rt(\mu, T^i, O^i, p^i, q^i, q''_i, \ldots)$ |
| DM | $\langle \text{declase}_\text{entry} \rangle \mapsto \langle \text{declase}_\text{entry} \rangle \ (\forall r \in \mathcal{D}(\mu) \land rt(\mu, T^i, O^i, p^i, q^i, q''_i, \ldots)$ |

### Algorithm 1 Model Transformation

1. $\Delta' \leftarrow \{\text{startConf}(\mu)\} \ (\forall x \in \text{dom}(\mu) : \xi(x) = x) \land rt(\mu, T^i, O^i, p^i, q^i)$
2. end for

end for

end for

end for

end for

end for

end if

end if

end for

end if

end if

end if

end for

end for

end for

end for

end if

end if

end if

end for

end if

end if

end if

end if

end if

end if

end if

end if

end if

end if

end if

end if

end if
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 $\Delta'$ denotes the initial interleaving assignments from public variables to their companion variables. $r.R^{G(x)}_{x \in \text{Var}}$ means a relation substituting each variable in Var with the renamed companion variable.

**Theorem 1** (Correctness). Let $SC(P^f)$ be the pushdown system w.r.t. security domain $\ell$ generated by our self-composition on the model of program $P$. If $\forall \ell \in \mathcal{D}$, the state error of $SC(P^f)$ 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 and gradual release [10]. Lux and Mantel [10] 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 conformance of the properties with these principles are given in Table III. Similar to the gradual release, the program $P_1$ in Table IV is secure (denoted by $\triangleright$) 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 $\text{declass}$ primitives. A typical example is $P_0$. 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.

Then we evaluate the preciseness of our enforcement mechanism. In Table IV $\tau_1$ is the well-typeness of program judged by the type system in Fig.4, [8]. $\tau_2$ is the judgement of the type system given in Fig.3, [10]. RA is the reachability analysis result using our mechanism. $\checkmark$ 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 ($P_2,P_6,P_7$ for WHERE and $P_1,P_2,P_6$ 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 migrate our approach to the properties on other dimensions of declassification, e.g. the delimited release [17] on the what-
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 Enman 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 meted 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 systems,” 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. Guttmann, and F. Martineili, 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. 5233. Springer, 2005, 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://infosc.pku.edu.cn/~suncong/sun2010d-tr.pdf](http://infosc.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/](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. Holecek, D. Suwimonteerabuth, S. Schwoon, and J. Esparza, “Introduction to remopla,” 2006, [http://www.fmi.uni-stuttgart.de/szs/tools/moped/remopla-intro.pdf](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.

| Case | From | WHERE | $\tau_1$ | $\tau_2$ | $\text{GR}$ | where | RA | $T$(ms) | $N_{\text{miss}}$ |
|------|------|-------|---------|---------|---------|-------|----|-------|------------|
| 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   | $l := h; h := \text{declass}(h)$; | | | | | | | | |
| P1   | $l := \text{declass}(h); h := h$; | | | | | | | | |
| P2   | $h_1 := h_2; l := \text{declass}(h_1)$; | | | | | | | | |
| P3   | $h_1 := h_2; h_2 := 0$; | | | | | | | | |
| P4   | $h_2 := 0$; if $h_1$ then $l := \text{declass}(h_1)$ else skip; | | | | | | | | |
| P5   | $l := 0$; if $l$ then $l := \text{declass}(h)$ else skip; | | | | | | | | |
| P6   | $h_2 := 0$; if $h_1$ then $l := \text{declass}(h_2)$ else $l := 0$; | | | | | | | | |
| P7   | $l := \text{declass}(h!) = 0$; if $l$ then $h_1 := \text{declass}(h_1)$ else skip; | | | | | | | | |
Proof of Lemma 7: Suppose any trace of the program

\[ P[C'/C] \] is in a form of

\[ (μ, I, O, p, q, P[C'/C]) \rightarrow^* (μ_j, I, O_j, p_j, q_j, C'; P_j) \rightarrow (μ_k, I, O_k, p_k, q_k, P_k) \rightarrow^* (μ_f, I, O_f, p_f, q_f, \text{skip}) \]

Because C and C' are semantically equivalent, we also have

\[ (μ_j, I, O_j, p_j, q_j, C; P_j) \rightarrow (μ_k, I, O_k, p_k, q_k, P_j) \].

Moreover, since C and C' are declassification-free, the substitution will not influence the conjunction of equivalence on declassified expressions in \( P[C'/C] \). Therefore the indistinguishability on the final configurations, that is \( μ_{n+1} \sim_\ell μ'_{n+1} \land O_{n+1} \sim_\ell O'_{n+1} \), holds before and after the substitution.

Proof of Lemma 8: From the operational semantics we can see →_δ can only occurs when a \text{declass} command is executed and the declassified expression contains some information with a security domain higher than the security domain of x. P has no declassification implies that in any trace of computation of P there is no →_δ. The where-security of P degenerates to have n = 0. Therefore the where-security becomes noninterference according to the definition and \( μ_\ell \equiv μ_1, O_\ell \equiv O_1 \).

Proof of Lemma 9: There are actually two cases on whether the substitution introduces a real declassification:

1. If \( σ(e) ≤ σ(x) \), the computation of \( x := \text{declass}(e) \) is identical to the ordinary assignment \( x := e \) and \( →_δ \) is not labeled as →_δ. The where-security of \( P[x := \text{declass}(e)/x := e] \) does not change compared with the where-security of \( P[x := e] \).

2. Suppose we have \( σ(x) ≈ σ(e) \). The computation of \( x := e \) in the two correlative runs of \( P[x := e] \) are like

\[ (μ, I, O, p, q, P[x := e]) \rightarrow^* (μ_j, I, O_j, p_j, q_j, x := e; P_j) \rightarrow (μ_j[x := μ_j(e)], I, O_j, p_j, q_j, P_j) \rightarrow^* (μ_{n+1}, I, O_{n+1}, p_{n+1}, q_{n+1}, \text{skip}) \]

and

\[ (μ', I', O', p', q, P[x := e]) \rightarrow^* (μ_j', I', O_j', p_j', q_j', x := e; P_j') \rightarrow (μ_j'[x := μ_j'(e)], I', O_j', p_j', q_j', P_j') \rightarrow^* (μ'_{n+1}, I', O'_{n+1}, p'_{n+1}, q'_{n+1}, \text{skip}) \].

From the premise of where-security of \( P[x := \text{declass}(e)/x := e] \) we have

\[ \bigwedge_{k=1..n} (μ_{k+1} \sim_\ell μ_k, O_{k+1} \sim_\ell O_k) \].

This implies

\[ \bigwedge_{k=1..n} k \neq j (μ_{k+1} \sim_\ell μ_k, O_{k+1} \sim_\ell O_k) \].

Because \( μ_j \sim_\ell μ_j \land μ_j(e) = μ_j'(e) \), according to the semantics, we have \( μ_j[x := μ_j(e)] \sim_\ell μ_j'[x := μ_j'(e)] \), that is \( μ_{j+1} \sim_\ell μ_{j+1}' \) for \( P[x := \text{declass}(e)/x := e] \) and therefore

\[ \bigwedge_{k=1..n} (μ_{k+1} \sim_\ell μ_k) \].

On the other hand, since the substitution does not change the semantics of program, restricting the premise

\[ \bigwedge_{k=1..n} k \neq j (μ_{k+1} \sim_\ell μ_k, O_{k+1} \sim_\ell O_k) \]

with a conjunction to \( μ_j(e) = μ_j'(e) \) will not influence the consequence that \( μ_{n+1} \sim_\ell μ'_{n+1} \land O_{n+1} \sim_\ell O'_{n+1} \). The where-security of \( P[x := \text{declass}(e)/x := e] \) is proved.

Proof of Theorem 1: Suppose program P violates the where-security property, that means

\[ \exists k_0. μ_{k_0+s} \sim_\ell μ_{k_0,s} \land μ_{k_0,s}(e_k) = μ_{k_0,s}(e_k') \land \neg(μ_{k_0,t} \sim_\ell μ_{k_0,t}) \]

or

\[ \bigwedge_{k=1..n} (μ_{k+1} \sim_\ell μ_k, O_{k+1} \sim_\ell O_k) \]

Theorem 1 has been adapted to \( O_{n+1} \sim_\ell O'_{n+1} \) by modeling final outputs of public variables. If the first relation is satisfied, we have in \( x_k := \text{declass}(e_k) \) and \( x_k' := \text{declass}(e_k') \) the variable \( x_k \) and \( x_k' \) are different variables. Therefore the respective pushdown rules must have different \( γ_k \) as the label for the stack symbol \( \text{declass}(e_k) \), which we suppose to be \( γ_k, γ_k' \) and \( γ_k \neq γ_k' \). From the DS_γ_k and DM_γ_k we have \( D[ρ(γ_k')] = e_k' \). The value in \( D[ρ(γ_k')] \) is irrelevant to \( e_k \) and \( x_k \) in the second run is not restricted by DM_γ_k. When the final \( x_k \) and \( x_k' \) are outputted, the inequality of final \( x_k \) of correlative executions makes the state error reachable according to the rule OM_{n+1}(x_k). If the second relation is satisfied, \( ∃ i. q_i \neq q_i' \lor (∃ i \leq k_0, μ_i \land q_i \neq O_i[|] \neq O_i'[|] \land q_i \neq q_i' \land q_i' \neq q_i \). Because \( q_1 \neq q_1' \), we can suppose \( q_i < q_i' \) and \( q_i' < q_i \). The correlative runs are symmetrical. Then there must be some e of output(e,O') in P that should be compared with the indefinite value in \( O_i,n+1[|] \) during the execution of the second run. Otherwise we have \( O_i,n+1[|] \neq O_i,n+1[|] \). Then if \( O_i,n+1[|] \) is generated by output(e,O_i), the second run is directed by \( O_i[|] \neq e \) according to the rule OM and error is reachable. From the contrapositive the theorem is proved.