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 policies 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; pushdown 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 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.
are respectively the set of input and output channels. They are a low-level sink of data observable to the attacker.

We use a sequential imperative language with I/O channels \((I, O, \mu, p, q, \text{skip}; C)\) for the evaluation semantics in Fig. 1. The language is deterministic. The confidential data of expression \(C\) stands for declassification that downgrades the confidentiality of a variable in \(e\) and only then an element \((\sigma(e), \sigma(x))\) is contained in the relation \(\sim\). The language is independent to the declassification \(\text{declass}\) commands. Our treatment is reasonable since developer should have right to decide the exception when they use the primitive \(\text{declass}\) 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 confidenital 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 stores \(\mu_i\) and \(\mu_j\) are indistinguishable on \(\ell\) \((\in D)\), denoted by \(\mu_i \sim_{\ell} \mu_j\), iff \(\forall x \in \text{Var} \cdot \sigma(x) \leq \ell \Rightarrow \mu_i(x) = \mu_j(x)\). For input channel \(I_i\) and \(I_j\), \(I_i \sim_{\ell} I_j\) iff \((\sigma(I_i) = \sigma(I_j) \leq \ell) \land (p_i = p_j \land q_i \land q_j \land \forall 0 \leq k < p_i, I_i[k] = I_j[k])\). Similarly, for output channel \(O_i\) and \(O_j\), \(O_i \sim_{\ell} O_j\) iff \((\sigma(O_i) = \sigma(O_j) \leq \ell) \land (q_i = q_j \land q_i \land q_j \land \forall 0 \leq k < q_i, O_i[k] = O_j[k])\).

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 \(\ell' \leq \ell\). 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 \(\forall i, I_i \in \mathcal{I} \Rightarrow I_i \sim_{\ell} I_i'\). 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 \(\ell_0\), iff \(\forall \ell \leq \ell_0\), we have

\[
(\mu_f, \mathcal{I}, \mathcal{O}_f, p_f, q_f, \text{skip}) \wedge \mathcal{I} \sim_{\ell} \mathcal{I}' \land \mu \sim_{\ell} \mu'
\]
In this definition, the noninterference property is related to a security domain $\ell_0$. The content of channels with security domain $\ell' (\ell' \succ \ell_0)$ is unobservable and irrelevant to the property. A more specific way to define noninterference is to require $\ell_0 = \bigsqcup D$. 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 [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 wheredimension 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 security iff $\forall \ell \in D$, we have

$$\forall \ell, I, µ, I', µ'. \exists n \geq 0 :$$

$$\begin{align*}
&\exists O_{n+1}, µ_{n+1} : (µ, I, O, p, q, P) \rightarrow (* (µ_{k'}, I', O', p', q', P) \rightarrow (* (µ_{k'}, I', O', p', q', skip) \\
&\wedge O_{n+1} \sim_r I' \wedge µ_{n+1} \sim_r µ') \\
&\wedge x_k := declass(e_k); P_k \rightarrow_d (µ_{k'}, I', O', p', q', q_k) \\
&\wedge (µ_{n+1}, I', O_{n+1}, p_{n+1}, q_{n+1}, skip) \\
&\wedge (x_k \sim_r (µ_{k'}, I', O', p', q', q_k)) \\
&\wedge (µ_{n+1}, I', O_{n+1}, p_{n+1}, q_{n+1}, skip)
\end{align*}$$

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 of 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 $µ_{n+1} \sim_r µ'_{n+1}$ to the judgement of $O_{n+1} \sim_r O'_{n+1}$.

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 three new 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 $P[x := declass(e)/x := e]$ 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 $P = (G, Γ × L, Δ)$. $G$ and $L$ are respectively the domain of global variables and local variables. $Γ$ is the stack alphabet. $Δ$ is the set of symbolic pushdown rules $\{(γ) \Rightarrow (γ_1 \cdots γ_n)(R) | γ, γ_1, \cdots, γ_n ∈ Γ ∧ R ∈ (Γ × L) × (Γ × L^n) ∧ n ≤ 2\}$.

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) [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 $\ell \in D$, we only model the channels in $I^\ell$ and $O^\ell$. Take a input command for example, if the source channel is $I_1$, the pushdown rule has a form of $IR_1$ for $σ(I_1) > ℓ$ and $IR_2$ for $σ(I_1) ≤ ℓ$ in Table I, where $D$ denotes an indefinite value.
On the other hand, if the target channel of output is $O_i$, the pushdown rule has a form of $OR_H$ for $\sigma(O_i) \not\supset \ell$ and $OR_L$ for $\sigma(O_i) \leq \ell$ 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 reannal on value of global variables and on value of local variables in $\langle \gamma_j \rangle \leftarrow \langle \gamma_k \rangle$. $rt_2$ for a rule $\langle \gamma_j \rangle \leftarrow \langle \text{entry}_i \rangle \langle \gamma_k \rangle$ denotes reannal 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 channel and to guide the instrumented computation to fulfill 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 $p'$ 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 $O_i$ 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 $error$.

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 $D$ Suppose there are $m$ declassifications respectively at code location $\gamma_{d_i}$ $(0 \leq i < m)$ and a function $\rho$ mapping $\gamma_{d_i}$ to $i$. We give another pattern of store-match that stores the value of expression declassified at $\gamma_{d_i}$ to the site $D[\rho(\gamma_{d_i})]$, see $DS$ in Table II. The corresponding match operation has a form of $DM$ in Table II. Note that $\xi$ 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
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. $r.R_{x \in Var}^{\ell(x)}$ means a relation substituting each variable in $Var$ with the renamed companion variable.

**Theorem 1** (Correctness). Let $SC(P^t)$ 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 $SC(P^t)$ 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 $P_1$ in Table IV is secure (denoted by $\checkmark$) 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 $P_0$. 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 $0 \sim 2^{3^3} - 1$. 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 $\leq$ 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 immigrate our approach to the properties on other dimensions of declassification, e.g. the delimited release [17] on the what-

<table>
<thead>
<tr>
<th>Table III</th>
<th>Difference between Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WHERE</td>
</tr>
<tr>
<td>noninterference up-to</td>
<td>✓</td>
</tr>
<tr>
<td>persistence</td>
<td>✓</td>
</tr>
</tbody>
</table>

![Fig. 3. Cost Reduction with Store-Match Pattern](image-url)
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 Nature 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