Flow-Sensitive Automaton-Based Monitoring of a Declassification Policy

Hao Zhu
School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
School of Computer Science and Technology, Nantong University, Nantong, China
Email: searain@nuaa.edu.cn

Yi Zhuang
School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
Email: zhuangyi@263.net

***Abstract*—Declassification policies aim to guarantee trusted release of confidential information. The semantic security conditions of declassification policies focus on different dimensions. In order to prevent the special attacks aiming to compromise the mechanisms of declassification, it is important for a declassification policy to combine different dimensions. Moreover, current body of work on the enforcement of the declassification policy focuses on static and flow-insensitive information-flow analysis, which is over-restrictive and imprecise. Dynamic and flow-sensitive information flow analysis techniques offer distinct advantages in permissiveness and precision. As a step in these directions, this paper first presents a declassification policy combining two dimensions, which control the amount and the location of confidential information release respectively, based on the security-typed language proposed. Then we presents an automaton-based monitoring mechanisms of the declassification policy. Abstractions of events occurring during the execution of a program are sent to the automaton as inputs, and the automaton uses these inputs to track the information flows and controls the execution of the program by forbidding or editing insecure commands that violate the declassification policy. Additionally, we prove the monitoring mechanism proposed is sound.**

***Index Terms*—automaton, confidentiality, declassification policy, information flow security, noninterference**

I. Introduction

One of the most fundamental information security issues is how to verifiably protect the confidentiality of sensitive information [13]. The standard way to protect confidentiality is access control, which constrains the release of confidential information, but not propagation of information once released [10]. Other common ways such as firewalls, encryption, and antivirus software are useful for protecting confidential information. However, these ways do not provide end-to-end confidentiality security.

Information-flow controls ensure that the information propagates through program constructs without security violations such that no confidential information is leaked to public outputs, thus, provides a promising approach to achieve the end-to-end confidentiality security.

The baseline policy of information-flow controls is noninterference [2], which states that public outputs do not depend on confidential inputs. However, achieving noninterference is often not possible because computing systems often deliberately release information that depends on the confidential inputs. For example, the password checking program needs to reject an incorrect password, but this operation reveals some information about the password. Hence, over-restrictiveness of the noninterference needs relaxing, and proper release of confidential information to public outputs is allowable. Thus, the mechanism of declassification is introduced [12], which downgrades the given information with high-level confidentiality to low-level confidentiality.

The information laundering attacks can exploit the mechanism of declassification to leak extra confidential information which is unexpectedly declassified [4]. Hence, security policies of declassification should be enforced to prevent these attacks. Sabelfeld and Sands [3] classify declassification policies according to four dimensions: WHAT (what information is released), WHO (who releases information), WHERE and WHEN (where and when information is released). Declassification policies of different dimensions tend to address only one aspect of the information release, exposing the other aspects for possible attacks. In order to avoid these attacks, it is desirable to combine defense along different dimensions. However, different dimensions are largely orthogonal to each other. The tight integration of these dimensions remains an open challenge [3]. This paper only focuses on the WHAT and WHERE dimensions.

The enforcement of declassification policies mostly use static information-flow analysis techniques which are often realized by the typing system, restrict the insecure information flow at the compile time once for all, and do not incur the performance overheads at the runtime of a program [4, 5, 11, 16, 19]. However, static analysis concerns all execution paths of a program, therefore, if a single execution path of a program violates the declassification policy, then the program is rejected. Moreover, some typing systems use effect systems that over-approximate the declassification policy [4, 5].

doi:10.4304/jsw.7.11.2478-2487

Hence, the static analysis is imprecise and over-restrictive for declassification policies.

Dynamic information-flow analysis techniques which are often realized by the monitoring mechanism, check the information flow at the runtime of a program, and verify security conditions for a single execution path of a program, independent of the behavior of other execution paths [8]. Moreover, dynamic analysis can gain more precise information flow of a program than a static analysis would [6]. The main drawback to the dynamic analysis is the decrease in the execution speed because the monitor must be run with every execution, while static analysis is run once for all prior to the execution. However, with the improvement of the computer hardware, this drawback can be weakened. Furthermore, driven by the need for permissiveness in dynamic and distributed applications, dynamic information-flow analysis techniques become increasingly popular [9].

The enforcement of declassification policies proposed to date suffers from another common drawback, which is flow-insensitive [6]. In this paper, the automaton-based monitoring proposed is flow-sensitive, i.e., it is possible for variables to update values of different confidentiality levels on each assignment in the following way. The confidentiality level of the assigned variable is set to high in case there is a variable of high confidentiality on the right-hand side of the assignment or in case the assignment appears inside of a high context (i.e. the guard expression of the conditional or loop statement is of high confidentiality). The confidentiality level of the variable is set to low in case there are no high confidentiality variables in the right-hand side of the assignment and the assignment does not appear in high context. Under other circumstances, the confidentiality level of the assigned variable is not updated. The goal of flow-sensitive enforcements is to accept more programs without jeopardizing security [6,7].

The main contributions of the paper include: (i) We propose a fine-granularity delimited declassification policy combining WHAT and WHERE dimensions; (ii) We give the flow-sensitive automaton-based monitoring mechanisms of the declassification policy; (iii) We prove the soundness of the enforcement mechanisms.

The remainder of the paper is structured as follows. In Section II, we present the syntax and semantics of language model, and then propose the fine-granularity declassification policy in Section III. Subsequently we state the automaton-based monitoring mechanisms for the declassification policy in Section IV. Section V skims through the related works. The conclusions are made in Section VI.

II. SECURITY-TYPED LANGUAGE

A. Security Lattice

In the security-typed language, each piece of data is labeled an initial confidentiality level. For simplicity, but without loss of generality, we consider two levels of confidentiality: *low* (*public*) and *high* (*confidential*). The partial order ($\sqsubseteq$) specifies the relationship between different confidentiality levels. If $a_1 \sqsubseteq a_2$ then data at level a_1 is less confidential than data at level a_2. Let $SC=\{low, high\}$, then $(SC, \sqsubseteq)$ constitutes security lattice. $(SC, \sqcup, \sqcap)$ is the algebraic system induced from $(SC, \sqsubseteq)$, where the join operation $\sqcup$ (meet operation $\sqcap$) is used to calculate the least upper bound (greatest lower bound) of two expressions. The confidentiality level of an expression that combines sub-expressions at different confidentiality levels is the least upper bound of these sub-expressions. For example, if two variables x and y are labeled a_1 and a_2 respectively, the confidentiality level of expression $x+y$ is $a_1 \sqcup a_2$.

B. Security Lattice Policy

There are two basic kinds of information flows through program constructs [1,8]: direct and indirect flows, where direct flow represents that information is passed explicitly from the right-hand to the left-hand side of an assignment. For example, the assignment statement $p:=s$ exhibits a direct flow from s to p. This direct flow is secure only if $\Gamma(s) \sqsubseteq \Gamma(p)$, where Γ is a mapping from variables to confidentiality levels. Indirect flow denotes that information is passed via the control-flow structure (conditional or loop statement). There are two types of indirect flows: explicit indirect flow and implicit indirect flow, where the former appears when an assignment command is executed and the latter appears when an assignment is not executed. For instance, if a statement: **if** b **then** $p:=s$ **else skip end** is executed with $b=true$, then there exists an explicit indirect flow from b to p; if this statement is executed with $b=false$, then an implicit indirect flow from b to p is generated. These indirect flow are secure only if $\Gamma(b) \sqsubseteq \Gamma(p)$. As another example: **if** $x \geq y$ **then** $z:=w$ **else** $i:=i+1$ **end**, the indirect flow is secure only if $(\Gamma(x) \sqcup \Gamma(y)) \sqsubseteq (\Gamma(z) \sqcap \Gamma(i))$. In all, the fundamental principle of the security lattice policy is to prevent insecure direct or indirect flow.

C. Language Syntax and Semantics

To illustrate the declassification policy, we present a simple imperative language extended with both output and declassification commands. The language syntax given in Figure 1 is composed of the expression and command. Expression e consists of constant n, variable x, and composite expressions $e_1 \oplus e_2$, where $\oplus$ is a binary operation. Command C is either atomic command A, branching command B, or sequential composition command C; C.

e::=n | x | $e \oplus e$
A::= **skip** | x:=e | x:=**declassify**(e) | **output**(e)
B::= **if** e **then** C **else** C **end** | **while** e **do** C **end**
C::= A | B | C;C

Figure 1. Syntax of the language

There are two special commands: x:=**declassify**(e) and **output**(e), where the former is the only command that is not standard, and it downgrades the confidentiality level of expression e (called declassifying expression) containing only high variables to low level, and then assigns it to the variable x. At the semantic level,

x:=**declassify**(*e*) is equivalent to *x*:=*e*, and the intention is used for controlling the confidential level of information without affecting the execution of the command. The latter command **output**(*e*) is used to represent any kind of public output of expression *e* containing only low variables. Confidential output is simply ignored because it does not influence the security check. In this paper we suppose values of variables in the program state can not be directly observed, and values of expression *e* can be learned only through the command **output**(*e*). Hence, any confidential information which needs output must be declassified to the low level first.

Expression configurations have the form *cfge*=⟨*m*, *e*⟩, where *m* is a memory mapping variables to values and *e* is an expression. Expression evaluation rules have the form⟨*m*, *e*⟩↓*m*(*e*), where *m*(*e*) is the result of evaluating expression *e* in memory *m*. We write *m*[*x*↦*v*] when the updating variable *x* with value *v* in memory *m*. Figure 2 presents the semantics of expressions.

Figure 3 displays the structural operational semantics

$$\text{(S-CON)}: \langle m, n\rangle \downarrow n \qquad \text{(S-VAR)}: \langle m, x\rangle \downarrow m(x)$$

$$\text{(S-OP)}: \frac{\langle m, e_1\rangle \downarrow m(e_1) \quad \langle m, e_2\rangle \downarrow m(e_2) \quad m(e_1 \oplus e_2) = m(e_1) \oplus m(e_2)}{\langle m, e_1 \oplus e_2\rangle \downarrow m(e_1) \oplus m(e_2)}$$

Figure 2. Semantics of expressions

of commands. Command configurations have the form *cfgc*=⟨*m*, *C*⟩, which indicates that the command *C* is to be executed from the memory *m*. A terminating command configuration with memory *m* is denoted by ⟨*m*, *stop*⟩. A transition between command configurations has the form $\langle m, C\rangle \xrightarrow{\alpha}_{\gamma} \langle m', C'\rangle$, where α is a transition event and γ stands for an externally observable event which can be represented as *obs*(*e*) (an output of expression *e*) or be omitted (empty event, which is also notated as ϵ).

Among all transition events, event *nop* signals that the program performs a command **skip**, event *a*(*x*,*e*) represents an assignment to variable *x* of expression *e*, event *d*(*x*,*e*) records the declassification of expression *e* into variable *x*, event *b*(*e*) denotes that the program branches on guard expression *e*, event *f* indicates that the structure block of a conditional or loop command has finished execution, and event *o*(*e*) stands for that the command **output**(*e*) is executed.

III. Declassification Policy

We consider the confidentiality of the program in two aspects. On the one aspect, the program must satisfy the security lattice policy to prevent insecure direct and indirect information flows, however, the program still may leak extra confidential information which is unexpectedly declassified, even if this program satisfies the security lattice policy; the reason is that attackers may launch information laundering attacks. Therefore, on the other aspect, we should define some other security conditions to prevent information laundering attacks.

Let us define the information laundering attack. In general, active attackers may change system behavior by injecting the new code into the program. However, in order to make the attack code difficult to detect, the data access by the attack code must satisfy certain conditions.

Definition 1. The information laundering attack is a command *T* satisfying security lattice policy, and it is formed according to the following grammar:

T::= **skip** | *x*:=*e* | **output**(*e*) | **if** *e* **then** *T* **else** *T* **end**
| **while** *e* **do** *T* **end**

The declassification command is excluded from the grammar in order to prohibit the attacker from learning any confidential information. This exclusion can be realized by integrity restriction [11].

Based on the delimited release policy proposed by Sabelfeld and Myers [4] and the security lattice policy, we propose a fine-granularity delimited declassification policy, which guarantees the declassification mechanism cannot be exploited to construct information laundering attacks from the WHAT and WHERE dimensions. The policy is defined as follows:

Definition 2. Let $E = \{e_{r_1}, \cdots, e_{r_n}\}$ be the set of all declassifying expressions in declassification commands within the program *C*; $m_1 =_{low} m_2$ denotes memories m_1 and m_2 are indistinguishable in all the low variables in the program; $m^i\ (1 \le i \le n)$ denotes the memory when the command *x*:=**declassify**(e_i) is executed exactly from initial memory m. A program *C* satisfies the fine-granularity delimited declassification policy if
(i) *C* satisfies security lattice policy.

$$\text{(S-SKIP)}: \langle m, skip\rangle \xrightarrow{nop} \langle m, stop\rangle \qquad \text{(S-ASSIGN)}: \frac{\langle m, e\rangle \downarrow m(e)}{\langle m, x := e\rangle \xrightarrow{a(x,e)} \langle m[x \mapsto m(e)], stop\rangle}$$

$$\text{(S-SEQ}_1\text{)}: \frac{\langle m, C_1\rangle \xrightarrow{\alpha}_{\gamma} \langle m', stop\rangle}{\langle m, C_1; C_2\rangle \xrightarrow{\alpha}_{\gamma} \langle m', C_2\rangle} \qquad \text{(S-SEQ}_2\text{)}: \frac{\langle m, C_1\rangle \xrightarrow{\alpha}_{\gamma} \langle m', C_1'\rangle}{\langle m, C_1; C_2\rangle \xrightarrow{\alpha}_{\gamma} \langle m', C_1'; C_2\rangle}$$

$$\text{(S-IF}_1\text{)}: \frac{\langle m, e\rangle \downarrow m(e) \quad m(e) \neq 0}{\langle m, if\ e\ then\ C_1\ else\ C_2\ end\rangle \xrightarrow{b(e)} \langle m, C_1; end\rangle} \qquad \text{(S-IF}_2\text{)}: \frac{\langle m, e\rangle \downarrow m(e) \quad m(e) = 0}{\langle m, if\ e\ then\ C_1\ else\ C_2\ end\rangle \xrightarrow{b(e)} \langle m, C_2; end\rangle}$$

$$\text{(S-WHILE}_1\text{)}: \frac{\langle m, e\rangle \downarrow m(e) \quad m(e) \neq 0}{\langle m, while\ e\ do\ C\ end\rangle \xrightarrow{b(e)} \langle m, C; end; while\ e\ do\ C\rangle} \qquad \text{(S-WHILE}_2\text{)}: \frac{\langle m, e\rangle \downarrow m(e) \quad m(e) = 0}{\langle m, while\ e\ do\ C\ end\rangle \xrightarrow{b(e)} \langle m, end\rangle}$$

$$\text{(S-OUTPUT)}: \frac{\langle m, e\rangle \downarrow m(e)}{\langle m, output(e)\rangle \xrightarrow{o(e)}_{obs(m(e))} \langle m, stop\rangle} \qquad \text{(S-END)}: \langle m, end\rangle \xrightarrow{f} \langle m, stop\rangle$$

$$\text{(S-DECLASSIFY)}: \frac{\langle m, e\rangle \downarrow m(e)}{\langle m, x := declassify(e)\rangle \xrightarrow{d(x,e)} \langle m[x \mapsto m(e)], stop\rangle}$$

Figure 3. Semantics of commands

(ii)$\forall e_{r_i} \in E, 1 \le i \le n, \forall m_1, m_2,$
$m_1(e_{r_i}) = m_2(e_{r_i}) \& m_1 =_{low} m_2 \Rightarrow m_1^{r_i} =_{low} m_2^{r_i}$.

The security provided by the fine-granularity delimited declassification policy can be illustrated with some examples. Consider the following program C_1:

h_2:=h_1;h_3:=h_1;h_4:=h_1;
avg:= **declassify**((h_1+h_2+h_3+h_4)/4);
output(*avg*).

Where h_1~h_4 are confidential variables and *avg* is public one. Assume initial memories m_1 and m_2 satisfying:

$m_1 =_{low} m_2$, $m_1(h_1)=m_2(h_2)=2$, $m_1(h_2)=m_2(h_1)=3$,
$\forall i \in \{3,4\}$. $m_1(h_i)= m_2(h_i)=0$.

We have

$m_1((h_1+h_2+h_3+h_4)/4)=m_2((h_1+h_2+h_3+h_4)/4)=5/4$.

Assume m_1' and m_2' are two memories when the command *avg*:= **declassify**((h_1+h_2+h_3+h_4)/4) of program C_1 is just executed from two initial memories m_1 and m_2, respectively. We have m_1' (*avg*)=2 and m_2' (*avg*)=3, so (ii) of Definition 2 is not satisfied. Therefore, program C_1 is rejected by Definition 2. In fact, program C_1 suffers from an information laundering attack. Suppose the average of confidential variables h_1~h_4 is intended to be released but no other information about h_1~h_4 is allowed be released to public variable *avg*, but attackers can inject the attack code: h_2:=h_1;h_3:=h_1;h_4:=h_1 to exploit the command: *avg*:= **declassify**((h_1+h_2+h_3+h_4)/4) to reveal entire information of secret variable h_1. Hence, program C_1 is insecure. Now consider a variation C_1' of the command C_1:

h_2:=h_1;h_3:=h_1;h_4:=h_1;
avg:= **declassify**((h_1+h_2+h_3+h_4)/4); *avg*:=0;
output(*avg*).

where all the variables and memories are the same as the above program C_1. By the same analysis as above we can infer that Definition 2 also rejects variation C_1', but the delimited release policy [4] accepts C_1' because delimited release only demands the equivalence of low memories in termination and ignores the mediate difference of low memories that can leak confidential information; Definition 2 corrects it by requiring equivalence of low memories on each declassification point. Now consider another program C_2:

p:=*s* ; **output**(*p*); *p*:=**declassify**(*s*).

where *s* is confidential variable and *p* is public one. This program ignores where confidential information should be released, i.e., information of variable *s* is released to variable *p* before its declassification. The insecure direct flow in command *p*:=*s* is detected by the security lattice policy, so program C_2 is rejected by (i) of Definition 2, but is accepted by delimited release policy, which ignores the WHERE dimension of the declassification policy. Askarov and Sabelfeld [5] extended *delimited release* into *localized delimited release* policy, which strengthening the demands of *delimited release* by location sensitivity. The rationale of *localized delimited release* disallows confidential information release before its declassification; however, it allows release to take place any time after declassification. Now consider the following program C_3:

h_2:=h_1; h_1:=*0*;
l:= **declassify**(h_1);
h_1:=h_2; *l*:=h_1;
output(*l*)

Due to the release of confidential information (*l*:=h_1) takes place after its declassification (*l*:= **declassify**((h_1)), *localized delimited release* policy deems program C_3 is secure and accepts it. In fact, this program is insecure, for the actual release occurs at the command *l*:=h_1. However, Definition 2 can rejects program C_3 because there exists insecure direct flow violating the security lattice policy.

IV. MONITORING AUTOMATON

A. Definition of Automaton

For any program *P*, whose variables belongs to the set *Vars*(*P*) and the set of confidential input variables is *S*(*P*) (*S*(*P*)$\subseteq$ *Vars*(*P*)), the monitoring automaton *A*(*P*), which enforces the fine-granularity delimited declassification policy on program *P*, is defined as a 5-tuple (Q, Φ, Ψ, δ, q_0), where Q is a set of states, Φ is a set of the input alphabet, Ψ is a set of the output alphabet, δ is a transition function $Q\times\Phi\rightarrow\Psi\times Q$, and q_0 is the initial state. Detailed description of each part above is as follows.

A state of the automaton is a pair (V, w), where the first element V is a set of variables belonging to *Vars*(*P*). At any step of the execution of program *P*, V contains all the variables whose values may have been influenced by the initial values of the variables in *S*(*P*), indicating that confidential level of variables in set V is upgraded to high due to the flow-sensitivity. The second element w of the pair is a word belonging to the language described by the regular expression $(L+H)^*$. The word w simulate a stack to tracks the explicit indirect flow. The initial state of the automaton is q_0=(*S*(*P*),ϵ).

The set of input alphabet is composed of an abstraction of events occurring during the execution of program *P*. On the one hand, the input alphabet of atomic command *A* (cf. Figure 1) is the same as the transition event of corresponding command, i.e., input *nop* corresponds to command **skip**, input *a*(*x*,*e*) represents command *x*:=*e*, input *d*(*x*,*e*) indicates command *x*:=**declassify**(*e*), and input *o*(*e*) denotes command **output**(*e*); on the other hand, the input alphabet of the branching command *B* (cf. Figure 1) not only includes the corresponding transition events (i.e. *b*(*e*) and *f*) but also contains an special input *not C*, which represents the branching statement *C* has not been executed due to the value of the previous guard expression, and this input is used to track the implicit indirect flow.

The set of output alphabet includes the following:

- **OK** is used to authorize the execution of an atomic command.
- **NO** is used to forbid the execution of an atomic command.
- **ACK** is used to acknowledge the reception of information useful for tracking indirect flow.

An automaton transition is written $(q, \phi)\xrightarrow{\psi} q'$ indicates that the automaton moves from state q to state q' and output ψ on reception of the input ϕ. The transition function uses two syntactic analysis functions: **FV**(*e*)

returns the set of variables occurring in expression e; **modified**(C) returns the set of variables modified by the execution of command C. A formal definition of **modified**(C) is shown in Figure 4. Additionally, m_0 is used to denote the initial memory corresponding to initial state of the automaton and m is used to indicate the current memory corresponding to the current state of the automaton.

modified(x:=e)=$\{x\}$
modified(x:=**declassify**(e))=$\{x\}$
modified(**output** (e))=$\emptyset$
modified(**skip**)=$\emptyset$
modified(**if** e **then** C_1 **else** C_2 **end**)= **modified**(C_1) $\sqcup$ **modified**(C_2)
modified(**while** e **do** C **end**)= **modified**(C)
modified(C_1;C_2)=**modified**(C_1) $\sqcup$ **modified**(C_2)

Figure 4. Definition of the function modified(C)

The transition rules of the monitoring automaton are presented as Figure 5, where the rule forbids or edits only the transition on the reception of the input $o(e)$. For other inputs, the only thing done by the transition is to keep track in the sets V and w.

Flow-sensitivity of the automaton-based monitoring is embodied by the following two functions. In the function (T-ASSIGN-sec), the assigned variable is added to the set V in the case there is a high variable on the right-hand side of assignment (i.e. FV(e)$\cap V\neq\emptyset$) or in the case the assignment appears inside of a high context (i.e. $w\notin\{L\}^*$). On the other hand, in the function (T-ASSIGN-pub), the assigned variable is removed from the set V in the case there are no high variables in the right-hand side of the assignment (i.e. FV(e)$\cap V=\emptyset$) and the assignment does not appear in the high context (i.e. w$\in\{L\}^*$).

Declassification is controlled by the transition function for the input $d(x,e)$. As described in the function (T-DECLASSIFY-pub), the declassifying expression e is allowed to be released to a public variable x only if the value of e in current memory is the same as it was in the initial memory, and the variable x is removed from the set V for the automaton is flow-sensitive. This prevents information laundering attacks because we only release the value of declassifying expression with respect to the initial memory at the time of declassification. Revisiting the program C_1 of Section III:

h_2:=h_1;h_3:=h_1;h_4:=h_1;
avg:= **declassify**((h_1+h_2+h_3+h_4)/4);
output(avg).

where h_1~h_4 are confidential variables and avg is public one as they were in Section III. Assume initial memory m_0 satisfying $m^0(h_1)=2$, $m^0(h_2)=3$ and $\forall i\in\{3,4\}$. $m^0(h_i)=0$. We have $m^0((h_1+h_2+h_3+h_4)/4)=5/4$. At the time of declassification in program C_1, the current memory m after the execution of three assignment commands (i.e. h_2:=h_1;h_3:=h_1;h_4:=h_1) is

$m((h_1+h_2+h_3+h_4)/4)=2\neq m^0((h_1+h_2+h_3+h_4)/4)=5/4$.

Therefore the declassification in program C_1 releases the extra confidential information than expected. This case will be dealt with by the rule (T-DECLASSIFY-sec), which states that when the declassification command is executed in the high context or the value of declassifying expression is not same in both initial memory and current memory, then variable x is added to the set V to denote that there exists unexpected confidential information flowing to assigned variable x. Consider a variation C_1' of the program C_1:

t:=h_1; h_1:=h_2;h_2:=t;
avg:= **declassify**((h_1+h_2+h_3+h_4)/4);
output(avg).

The automaton accepts runs of this program because at the time of declassification the declassifying expression (i.e. (h_1+h_2+h_3+h_4)/4) has the same value in both initial memory and current memory.

On the reception of the input $b(e)$ in the state (V,w), the automaton would verify if the branching command is executed in the high context, i.e., whether the guard expression e may have been influenced by the initial values of $S(P)$. To do so, it computes the intersection of variables in e with the set V. In the rule (T-BRANCH-low), the intersection is empty, i.e., the value of e is not influenced by initial values of $S(P)$. Then the transition adds L at the end of the word w. In the function (T-BRANCH-high), the intersection is not empty, and then the transition adds H at the end of w.

In the function (T-EXIT), on the reception of the input f, the transition removes the last letter of w to restore w to

$((V,w), a(x,e))\xrightarrow{OK}(V\cup\{x\}, w)$	iff $w\notin\{L\}^*$ or FV(e)$\cap V\neq\emptyset$	(T-ASSIGN-sec)
$((V,w), a(x,e))\xrightarrow{OK}(V\setminus\{x\}, w)$	iff $w\in\{L\}^*$ and FV(e)$\cap V=\emptyset$	(T-ASSIGN-pub)
$((V,w), d(x,e))\xrightarrow{OK}(V\setminus\{x\}, w)$	iff $w\in\{L\}^*$ and $m(e)=m_0(e)$	(T-DECLASSIFY-pub)
$((V,w), d(x,e))\xrightarrow{OK}(V\cup\{x\}, w)$	iff $w\notin\{L\}^*$ or $m(e)\neq m_0(e)$	(T-DECLASSIFY-sec)
$((V,w), b(e))\xrightarrow{ACK}(V, wH)$	iff FV(e)$\cap V\neq\emptyset$	(T-BRANCH-high)
$((V,w), b(e))\xrightarrow{ACK}(V, wL)$	iff FV(e)$\cap V=\emptyset$	(T-BRANCH-low)
$((V,wa), f)\xrightarrow{ACK}(V, w)$		(T-EXIT)
$((V,w), \text{not } C)\xrightarrow{ACK}(V\cup$**modified**$(C), w)$	iff $w\notin\{L\}^*$	(T-NOT-high)
$((V,w), \text{not } C)\xrightarrow{ACK}(V, w)$	iff $w\in\{L\}^*$	(T-NOT-low)
$((V,w), nop)\xrightarrow{OK}(V, w)$		(T-SKIP)
$((V,w), o(e))\xrightarrow{OK}(V, w)$	iff $w\in\{L\}^*$ and FV(e)$\cap V=\emptyset$	(T-OUTPUT-OK)
$((V,w), o(e))\xrightarrow{o(\theta)}(V, w)$	iff $w\in\{L\}^*$ and FV(e)$\cap V\neq\emptyset$	(T-OUTPUT-DEFAULT)
$((V,w), o(e))\xrightarrow{NO}(V, w)$	iff $w\notin\{L\}^*$	(T-OUTPUT-NO)

Figure 5. Transition function of monitoring automaton

its value before this branching command.

On the reception of the input *not C* in the state (*V*,*w*), the automaton would check if the command *C* may have been executed with different values for *S*(*P*). In the function (T-NOT-high), the execution context of the command *C* contains the confidential variable (i.e. *w* does not belong to $\{L\}^*$), and then variables whose values are modified by an execution of command *C* are added to the set *V*. In the function (T-NOT-low), the automaton does nothing else but to authorize its execution by outputting *OK*.

Due to atomic command **skip** is safe, the function (T-SKIP) states that the automaton authorize the execution of **skip** on the reception of the input *nop*.

The transition functions for the input *o*(*e*) are of three cases. The first one is the function (T-OUTPUT-OK), which states that in the public execution context, the automaton authorize the output of expression *e* which contains no confidential information. The other two functions are used to prevent insecure flows through the output command. The function (T-OUTPUT-DEFAULT) presents that in a public execution context, if the transition tries to output the value of a confidential expression *e*, then the value of the output is replaced by a default value θ. The function (T-OUTPUT-NO) states that the output command must be forbidden in the high execution context.

B. Monitored Semantics

The monitor runs alongside the program in a lockstep fashion. The natural semantics merging the standard semantics of commands and the monitoring automaton is presented in Figure 6. We use $\vec{\alpha}$ to denote a sequence of transition events and use $\alpha_1 \cdot \alpha_2$ to represent a concatenation of α_1 and α_2; similar form is for event γ.

There are four rules for atomic commands, where rules (M-ASSIGN) and (M-DECLASSIFY) indicate that the automaton authorizes the execution of assignment and declassification commands respectively, the rule (M-OUTPUT-edit) presents that the monitoring semantics executes **output**(θ) instead of **output**(*e*), and the rule (M-OUTPUT-no) omits the execution of **output**(*e*) as if **output**(*e*) was a **skip** command.

As for rules for the branching commands, i.e., (M-IF-false), (M-IF-true), (M-WHILE-false) and (M-WHILE-true), the input b(*e*) is sent to the automaton firstly. Then, the branch determined by guard expression *e* is executed (in the case where the branching command is a while statement and the condition is false, the branch executed can be looked as **skip**). The step followed is that the input *not C* is sent to the automaton, where *C* is the branch not executed due to the previous value of the guard expression *e* (in the case where the branching command is a while statement and the condition is true, the input is *not skip*). Finally, the input *f* is sent to the automaton. In the case of a while statement with a condition equals to true, the execution proceeds by executing the while statement once again.

$$\frac{\langle m, x := e\rangle \xrightarrow{a(x,e)} \langle m[x \mapsto m(e)], stop\rangle \quad ((V,w), a(x,e)) \xrightarrow{OK} (V',w)}{\langle m,(V,w), x := e\rangle \xrightarrow{a(x,e)} \langle m[x \mapsto m(e)],(V',w), stop\rangle} \quad \text{(M-ASSIGN)}$$

$$\frac{\langle m, x := declassify(e)\rangle \xrightarrow{d(x,e)} \langle m[x \mapsto m(e)], stop\rangle \quad ((V,w), d(x,e)) \xrightarrow{OK} (V',w)}{\langle m,(V,w), x := declassify(e)\rangle \xrightarrow{d(x,e)} \langle m[x \mapsto m(e)],(V',w), stop\rangle} \quad \text{(M-DECLASSIFY)}$$

$$\frac{\langle m, output(e)\rangle \xrightarrow{o(e)}_{obs(m(e))} \langle m, stop\rangle \quad ((V,w), o(e)) \xrightarrow{o(\theta)} (V,w)}{\langle m,(V,w), output(e)\rangle \xrightarrow{o(\theta)}_{obs(\theta)} \langle m,(V,w), stop\rangle} \quad \text{(M-OUTPUT-edit)}$$

$$\frac{\langle m, output(e)\rangle \xrightarrow{o(e)}_{obs(m(e))} \langle m, stop\rangle \quad ((V,w), o(e)) \xrightarrow{NO} (V,w)}{\langle m,(V,w), output(e)\rangle \xrightarrow{\epsilon}_{\epsilon} \langle m,(V,w), stop\rangle} \quad \text{(M-OUTPUT-no)}$$

$$\frac{\langle m,(V,w), C_1\rangle \xrightarrow{\vec{\alpha}_1}_{\vec{\gamma}_1} \langle m',(V',w'), stop\rangle \quad \langle m',(V',w'), C_2\rangle \xrightarrow{\vec{\alpha}_2}_{\vec{\gamma}_2} \langle m'',(V'',w''), stop\rangle}{\langle m,(V,w), C_1; C_2\rangle \xrightarrow{\vec{\alpha}_1\cdot\vec{\alpha}_2}_{\vec{\gamma}_1\cdot\vec{\gamma}_2} \langle m'',(V'',w''), stop\rangle} \quad \text{(M-SEQ)}$$

$$\frac{\begin{array}{c}((V,w), b(e)) \xrightarrow{ACK} (V,w') \quad \langle m,(V,w'), C_1\rangle \xrightarrow{\vec{\alpha}}_{\vec{\gamma}} \langle m',(V',w''), stop\rangle \\ m(e) \neq 0 \quad ((V',w''), not\ C_2) \xrightarrow{ACK} (V'',w'') \quad ((V'',w''), f) \xrightarrow{ACK} (V'',w''')\end{array}}{\langle m,(V,w), if\ e\ then\ C_1\ else\ C_2\ end\rangle \xrightarrow{\vec{\alpha}}_{\vec{\gamma}} \langle m',(V'',w'''), stop\rangle} \quad \text{(M-IF-false)}$$

$$\frac{\begin{array}{c}((V,w), b(e)) \xrightarrow{ACK} (V,w') \quad \langle m,(V,w'), C_2\rangle \xrightarrow{\vec{\alpha}}_{\vec{\gamma}} \langle m',(V',w''), stop\rangle \\ m(e) = 0 \quad ((V',w''), not\ C_1) \xrightarrow{ACK} (V'',w'') \quad ((V'',w''), f) \xrightarrow{ACK} (V'',w''')\end{array}}{\langle m,(V,w), if\ e\ then\ C_1\ else\ C_2\ end\rangle \xrightarrow{\vec{\alpha}}_{\vec{\gamma}} \langle m',(V'',w'''), stop\rangle} \quad \text{(M-IF-true)}$$

$$\frac{\begin{array}{c}m(e) = 0 \quad ((V,w), b(e)) \xrightarrow{ACK} (V,w_1) \\ ((V,w_1), not\ C) \xrightarrow{ACK} (V_1,w_1) \quad ((V_1,w_1), f) \xrightarrow{ACK} (V_1,w_2)\end{array}}{\langle m,(V,w), while\ e\ do\ C\ end\rangle \xrightarrow{\epsilon}_{\epsilon} \langle m,(V_1,w_2), stop\rangle} \quad \text{(M-WHILE-false)}$$

$$\frac{\begin{array}{c}m(e) \neq 0 \quad ((V,w), b(e)) \xrightarrow{ACK} (V,w_1) \quad \langle m,(V,w_1), C\rangle \xrightarrow{\vec{\alpha}_1}_{\vec{\gamma}_1} \langle m_1,(V_1,w_2), stop\rangle \\ ((V_1,w_2), f) \xrightarrow{ACK} (V_1,w_3) \quad \langle m_1,(V_1,w_3), while\ e\ do\ C\ end\rangle \xrightarrow{\vec{\alpha}_2}_{\vec{\gamma}_2} \langle m_2,(V_2,w_4), stop\rangle\end{array}}{\langle m,(V,w), while\ e\ do\ C\ end\rangle \xrightarrow{\vec{\alpha}_1\cdot\vec{\alpha}_2}_{\vec{\gamma}_1\cdot\vec{\gamma}_2} \langle m_2,(V_2,w_4), stop\rangle} \quad \text{(M-WHILE-true)}$$

Figure 6. Natural semantics of monitored executions

Program C_1	Proposed action: $m_0(h_1)=2, m_0(h_2)=3$ $\forall i\in\{3,4\}.\ m_0(h_i)=0.$	Automaton input Φ	Automaton output Ψ	Automaton state (V, w)	Executed
h_2:=h_1; h_3:=h_1; h_4:=h_1; e:= $(h_1+h_2+h_3+h_4)/4$; avg:= **declassify**(e); **output**(avg).	h_2:=h_1; h_3:=h_1; h_4:=h_1; e:= $(h_1+h_2+h_3+h_4)/4$; avg:= **declassify**(e); **output**(avg).	a(h_2, h_1) a(h_3, h_1) a(h_4, h_1) a(e, $(h_1+h_2+h_3+h_4)/4$)) d(avg, e) o(avg)	OK OK OK OK OK o(θ)	$(\{h_1, h_2, h_3, h_4\},\epsilon)$ $(\{h_1, h_2, h_3, h_4\},\epsilon)$ $(\{h_1, h_2, h_3, h_4\},\epsilon)$ $(\{h_1, h_2, h_3, h_4, e\},\epsilon)$ $(\{h_1, h_2, h_3, h_4, e, avg\},\epsilon)$ $(\{h_1, h_2, h_3, h_4, e, avg\},\epsilon)$	h_2:=h_1; h_3:=h_1; h_4:=h_1; e:= $(h_1+h_2+h_3+h_4)/4$; avg:= **declassify**(e); **output**(θ).

Figure 7. Monitored executions of program C_1

C. Monitored Example

The security provided by the monitoring automaton can be demonstrated by some examples.

Example 1 (Average Salary Attack) This attack is described as program C_1 in Section III. Figure 7 presents the monitored execution of this program. Program C_1 is given in the first column in Figure 7. The execution monitored is the one for which initial memory m_0 satisfying $m^0(h_1)=2$, $m^0(h_2)=3$ and $\forall i\in\{3,4\}.\ m^0(h_i)=0$. The set of confidential input variables of program C_1 (i.e. $S(P)$) is $\{h_1, h_2, h_3, h_4\}$. The atomic actions that the program will attempt to execute without the automaton-based monitoring are give in the column "Proposed action". The next column contains the input which is sent to the automaton for the security check. The transition function is applied on the automaton input of the same line and the automaton state of the previous line. The two following columns contain the result of the automaton transition. Finally, the last column shows the commands which are really fulfilled by the monitored execution.

On line 4 and line 5 of this program, variable e and avg are added to the set V in the automaton state according to the transition function (T-ASSIGN-sec) and (T-DECLASSIFY-sec) respectively. Additionally, on the last line, the program tries to output the value of variable avg which contains extra confidential information which is unexpectedly declassified. According to the function (T-OUTPUT-DEFAULT), the automaton disallows the output of this value, and outputs a default value θ, which represents that an output command has been denied for security reasons.

Example 2 (Electronic Wallet Attack [4]). Consider an electronic shopping scenario. Suppose variable h stores the amount of money in a customer's electronic wallet; variables l and k stores the amount of money spent during the current session and the cost of the item to be purchased, respectively. We assume that $\Gamma(h)=high$ and $\Gamma(l)=\Gamma(k)=low$. The following program C_4 transfers the amount k of money from a customer's electronic wallet to the variable l:

e:= **declassify**($h\geq k$);
if e **then** h:=h-k; l:=l+k **else skip end**;
output(l).

The above program satisfy two security conditions of Definition 2. Below is an variation C_4' of the program C_4:

l:=0;
while ($n\geq 0$) **do**
  k:=2^{n-1};
  e:= **declassify**($h\geq k$);
  if e then h:=h-k; l:=l+k **else skip end**;
  n:=n-1;
end
output(l).

This variation C_4' contains the information laundering attack code that abuses the declassification command to release the information of variable h bit-by-bit to l. We assume that h is a n-bit integer and $\Gamma(n)=low$. It is not difficult to see that variation C_4' is rejected by the Definition 2. Figure 8 lays out the monitored execution of the electronic wallet attack. Suppose the initial value of the variables n and h is 3 and 5 respectively, so the loop statement in the program is executed three rounds. In the second round, variable e and l are added to the set V according to the transition function (T-DECLASSIFY-sec) and (T-ASSIGN-sec) respectively. In the last round, variable e is excluded from the set V according to the transition function (T-DECLASSIFY-pub), but variable l remains in the set V. Hence at the execution of the last command **output**(l), the automaton outputs a default value θ instead of the actual value of l, preventing the leak of unexpected confidential information release. D. Soundness of Automaton-based Monitoring

The security of automaton-based monitoring of the declassification policy is guaranteed by the following soundness theorem.

Theorem 1. Any monitored execution of a program C based on the automaton defined above satisfies fine-granularity delimited declassification policy.

Proof. We sketch a proof by induction on the length of the sequence of the input of the automaton; this sequence is denoted by $\vec{s}_n$.

1) Base n=0. The two security conditions in Definition 2 trivially hold.

2) Induction step. We assume the theorem holds for $\vec{s}_{n-1}$ and need to prove that the theorem also holds for $\vec{s}_n$. We consider the following cases for the input s_n.

Case s_n= $d(x, e)$. There are two subcases:

Subcase 1: if $w\notin\{L\}^*$ or $m(e)\neq m^0(e)$, the security level of variable x is upgraded to confidential level according to the transition function (T-DECLASSIFY-sec). Hence, there is no insecure direct and indirect flow and the low memory is unchanged. The two security conditions in Definition 2 are satisfied.

Subcase 2: if $w\in\{L\}^*$ and $m(e)= m^0(e)$, the security level of variable x is downgraded to public level according to the transition function (T-DECLASSIFY-

Program C_4'	Proposed action: n:=3; $m_0(h)=5$	Automaton input Φ	Automaton output Ψ	Automaton state (V, w)	Executed
l:=*0*;	l:=*0*;	a(l, *0*)	OK	({h},ϵ)	l:=*0*;
while (n>*0*) **do**		b(n>*0*)	ACK	({h}, L)	
k:=2^{n-1};	k:=2^{n-1};	a(k, 2^{n-1})	OK	({h}, L)	k:=2^{n-1};
e:= **declassify**(h≥k);	e:= **declassify**(h≥k);	d(e, h≥k)	OK	({h}, L)	e:= **declassify**(h≥k);
if e then		b(e)	ACK	({h}, LL)	
h:=h-k;	h:=h-k;	a(h, h-k)	OK	({h}, LL)	h:=h-k;
l:=l+k;	l:=l+k;	a(l, l+k)	OK	({h}, LL)	l:=l+k;
else					
skip		*not* **skip**	ACK	({h}, LL)	
end;		f	ACK	({h}, L)	
n:=n-*1*;	n:=n-*1*;	a(n, n-*1*)	OK	({h}, L)	n:=n-*1*;
end;		f	ACK	({h}, ϵ)	
	(the second round)	b(n>*0*)	ACK	({h}, L)	
	k:=2^{n-1};	a(k, 2^{n-1})	OK	({h}, L)	k:=2^{n-1};
	e:= **declassify**(h≥k);	d(e, h≥k)	OK	({h, e}, L)	e:= **declassify**(h≥k);
		b(e)	ACK	({h, e}, LH)	
	skip	nop	OK	({h, e}, LH)	**skip**
		not h:=h-k; l:=l+k;	ACK	({h, e, l}, LH)	
		f	ACK	({h, e, l}, L)	
	n:=n-*1*;	a(n, n-*1*)	OK	({h, e, l}, L)	n:=n-*1*;
		f	ACK	({h, e, l}, ϵ)	
	(the third round)	b(n>*0*)	ACK	({h, e, l}, L)	
	k:=2^{n-1};	a(k, 2^{n-1})	OK	({h, e, l}, L)	k:=2^{n-1};
	e:= **declassify**(h≥k);	d(e, h≥k)	OK	({h, l}, L)	e:= **declassify**(h≥k);
		b(e)	ACK	({h, l}, LL)	
	h:=h-k;	a(h, h-k)	OK	({h, l}, LL)	h:=h-k;
	l:=l+k;	a(l, l+k)	OK	({h, l}, LL)	l:=l+k;
		not **skip**	ACK	({h, l}, LL)	
		f	ACK	({h, l}, L)	
	n:=n-*1*;	a(n, n-*1*)	OK	({h, l}, L)	n:=n-*1*;
		f	ACK	({h, l}, ϵ)	
output(l).	**output**(l).	o(l)	o(θ)	({h, l}, ϵ)	**output**(θ).

Figure 8. Monitored executions of program C_4'

pub). We assume that m_1^0 and m_2^0 are initial memories satisfying $m_1^0 =_{\text{low}} m_2^0$. The corresponding declassifying expression of s_n is denoted by e_n, and m_1^n and m_2^n denotes the current memories when the declassification command x:=**declassify**(e_n) is exactly executed from initial memories m_1^0 and m_2^0 respectively. By the premise of the function (T-DECLASSIFY-pub), we have

$$m_1^0(e_n) = m_1^n(e_n), \quad (1)$$

$$m_2^0(e_n) = m_2^n(e_n). \quad (2)$$

By the premise of (ii) of Definition 2, we have $m_1^0(e_n) = m_2^0(e_n)$, and by (1) and (2), we infer

$$m_1^n(e_n) = m_2^n(e_n). \quad (3)$$

By the induction hypothesis, we have

$$m_1^{n-1} =_{\text{low}} m_2^{n-1}. \quad (4)$$

Between input $\vec{s}_{n-1}$ and $\vec{s}_n$, there is no any command that will convey confidential information to the low memory, so by (3) and (4), we have $m_1^n =_{\text{low}} m_2^n$, hence the two security conditions in Definition 2 are satisfied. If any two initial memories are not low equivalent, i.e., $m_1^0 \neq_{low} m_2^0$, then the left-hand side of the implication in the item (ii) of Definition 2 is false, i.e., there is no demand on the equivalence of current memories ($m_1^n =_{\text{low}} m_2^n$).

Case s_n=a(x,e). Consider two subcases:

Subcase 1: if $w \notin \{L\}^*$ or FV(e)∩V≠∅, the subcase is similar to Subcase 1 of Case s_n=d(x,e).

Subcase 2: if w∈$\{L\}^*$ and FV(e)∩V=∅, obviously there is no insecure direct and indirect flow from e to x, so the two security conditions of Definition 2 is satisfied.

Case $s_n \in$ {b(e), f, *not C*, *nop*, o(e)}. According to the corresponding transition function in Figure 5, there exists no insecure direct and indirect flow, so the security condition (i) is satisfied. Due to under these inputs, the automaton does not change low memory, the security condition (ii) hold trivially. □

V. Related Work

Declassification policies have been an active area of research in language-based information flow security. In the following discussion, we focus on the most relevant work that targets the control of WHAT and WHERE dimensions. For other dimensions of declassification, we refer to the survey of declassification in [3].

Mantel and Sands [18] introduce a declassification policy based on intransitive non-interference [17]. In their policy, the release of confidential information can take place only at specific program points, but this policy ignores what is released.

Almeida Matos and Boudol [19] propose the non-disclosure policy, which is a generalization of non-interference. They introduce a flow declaration construct (**flow** F in M) for directly manipulating flow relations, where F is a flow policy, i.e. a binary relation on security

levels, and M is a language expression. The meaning is that M is executed in the context of the current flow policy extended with F. Non-disclosure is classified as a declassification policy of the WHERE dimension.

Sabelfeld and Myers [4] propose the delimited release policy that declassifies confidential information only through the special language operator **declassify**(e). The policy of delimited release is only concerned with the WHAT dimension, and ignores where information is released. Later, Askarov and Sabelfeld [5] extend the policy of delimited release into the localized delimited release policy, which strengthens the demands of the delimited release by location sensitivity. The rationale of localized delimited release disallows the release of the confidential information before its declassification, but it allows release of the confidential information to take place after declassification.

Gradual release [20] is on the WHERE-dimension, and it specifies the refinement of attacker knowledge about the confidential information is only via declassification statements, but it ignores what information is released and allows an attacker to release more confidential information in the legal declassification statements. Banerjee et al. [15] use gradual release as a starting point for combining the WHAT and WHERE dimensions, but their treatment of WHAT differs from ours: their policy ignores the initial state. This can be illustrated in the following example:

$$h := h' \ ; \ l := \textbf{declassify}(h).$$

where $\Gamma(h')=\Gamma(h)=high$ and $\Gamma(l)=low$. This program is rejected by Definition 2 while their policy accepts it since they ignore the fact that the actual value released is variable h' instead of variable h.

As for the monitoring mechanisms of information flow policies, Volpano [21] proposes a monitor that only checks direct flows, and indirect flows are ignored. Recently, Chudnov and Naumann [14] propose an flow-sensitive in-lining approach to monitoring information flow. The soundness of the in-lining approach is ensured by bi-simulation of the in-lined monitor and the original monitor. Magazinius et al. [9] presents a framework for in-lining dynamic information-flow monitors on the fly: security checks are injected into the source code as the computation goes along. The language they considered includes the statement of dynamic code evaluation of strings, whose content might not be known until runtime. Le Guernic et al. [8] presents an automaton-based approach to monitoring information flow. However, these monitoring mechanisms of information flow policies lack support for declassification policies. The baseline policy they considered is the noninterference policy. In this paper, we extended the noninterference monitoring to the declassification policy.

VI. Conclusion

In this paper, we have proposed the fine-granularity delimited declassification policy which combines WHAT and WHERE dimensions of the confidential information release. This declassification policy prevents the leak of confidential information through the immediate memory state of the program execution, and correct insecure factor introduced by the localized delimited release [5] which allows the release of confidential information after its declassification. Then we proposed a flow-sensitive automaton-based dynamic monitoring mechanism to enforce this declassification policy. The automaton is in charge of two main jobs. The first one is to track the information flows between the confidential input and the current value of the variables used by the program. The second one is to check the execution of atomic commands to ensure that there is no information flow violating the declassification policy. Moreover, we have proved the automaton-based enforcement is sound with respect to the fine-granularity delimited declassification policy.

Much future work is possible in this area. It would be desirable to integrate three dimensions of declassification (e.g., WHAT, WHERE and WHO) and enforce it by dynamic monitoring mechanism. Additionally, the fine-granularity delimited declassification policy and its enforcement are given for a rudimentary programming language, the extension of the formal results to richer languages will be a topic for future research.

Acknowledgment

This work is partially supported by the Aviation Foundation of China under Grant No. 2010ZC13012, the Jiangsu Innovation Program for Graduate Education Foundation under Grant No. CXLX11_0205.

References

[1] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236-243, 1976.

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

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

[4] A. Sabelfeld, and A. C. Myers, "A model for delimited information release," *Software Security-Theories and Systems*, vol. 3233, pp. 174-191, 2004.

[5] A. Askarov and A. Sabelfeld, "Localized delimited release: combining the what and where dimensions of information release," In *Proceedings of Programming Languages and Analysis for Security*, pp. 53-60, 2007.

[6] A. Russo and A. Sabelfeld, "Dynamic vs. Static Flow-Sensitive Security Analysis," In *Proceedings of the IEEE Computer Security Foundations Symposium*, pp. 186-199, 2010.

[7] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," *Perspectives of Systems Informatics, Lecture Notes in Computer Science*, vol. 5947, pp. 352-365, 2010.

[8] G. Le Guernic, A. Banerjee and D. A. Schmidt, "Automata-based confidentiality monitoring," *Advances in Computer Science-ASIAN 2006. Secure Software and Related Issues, Lecture Notes in Computer Science*, vol. 4435, pp. 75-89, 2007.

[9] J. Magazinius, A. Russo and A. Sabelfeld, "On-the-fly inlining of dynamic security monitors," *Security and Privacy–Silver Linings in the Cloud*, *IFIP Advances in*

Information and Communication Technology, vol. 330, pp. 173-186, 2010.

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

[11] A. Askarov and A. C. Myers, "A semantic framework for declassification and endorsement," *Programming Languages and Systems, Lecture Notes in Computer Science*, vol. 6012, pp. 64-84, 2010.

[12] M. Pistoia and Ú. Erlingsson, "Programming languages and program analysis for security: a three-year retrospective," *ACM SIGPLAN Notices*, vol. 43, pp. 32-39, 2009.

[13] Hong Mei and Dong-gang Cao, "Reliability of software: the challenges of the Internet," *Communication of the CCF*, vol. 6, no. 2. pp. 20-24, 2010.

[14] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," In *2010 23rd IEEE Computer Security Foundations Symposium*, pp. 200-214, 2010.

[15] A. Banerjee, D. A. Naumann and S. Rosenberg, "Expressive declassification policies and modular static enforcement," In *2008 IEEE Symposium on Security and Privacy*, pp. 339-353, 2008.

[16] G. Barthe, S. Cavadini, and T. Rezk, "Tractable enforcement of declassification policies," In *2008 IEEE 21st Computer Security Foundations Symposium*, pp. 83-97, 2008.

[17] A. W Roscoe and M. H. Goldsmith, "What is intransitive noninterference?," In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pp. 228-238, 1999.

[18] H. Mantel and D. Sands, "Controlled declassification based on intransitive noninterference," *Programming Languages and Systems*, vol. 3302, pp. 129-145, 2004.

[19] A. Almeida Matos and G. Boudol, "On declassification and the non-disclosure policy," *Journal of Computer Security*, vol. 17, pp. 549-597, 2009.

[20] A. Askarov, A. Sabelfeld, "Gradual release: unifying declassification, encryption and key release policies," In *IEEE Symposium on Security and Privacy*, pp. 207-221, 2007.

[21] D. Volpano, "Safety versus secrecy" *Static Analysis*, *Lecture Notes in Computer Science*, Vol. 1694/1999, pp. 847-848, 1999.

Hao Zhu (searain@nuaa.edu.cn) holds a Master's Degree in Computer Science and technology from the Jiangsu University, China and is currently a PhD candidate at the Nanjing University of Aeronautics and Astronautics, China. He is now a lecturer of computer science in Nantong university, China. His research interests include information flow security, trusted computing and intelligent computing.

Yi Zhuang (zhuangyi@263.net) is PhD supervisor in computer science in the Nanjing University of Aeronautics and Astronautics, China. Her research interests include information security, trusted computing and distributed computing.