Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/080449

Information Declassification for Multi-Threaded Programs

Hao Zhu[1,2,*], *Yi Zhuang*[1] *and Xiang Chen*[2]

[1] School of Computer Science and Technology, Nanjing University of Aeronautics Astronautics, Nanjing, 210016, P. R. China
[2] School of Computer Science and Technology, Nantong University, Nantong, 226019, P. R. China

Received: 18 Aug. 2013, Revised: 20 Nov. 2013, Accepted: 21 Nov. 2013
Published online: 1 Jul. 2014

Abstract: Information declassification aims for trusted release of secret information to public environment. Existing security specifications and enforcement mechanisms of declassification policies have focused on sequential programs. This paper generalized the specification of gradually delimited release policy for sequential programs to the security condition suited for multi-threaded programs. This security condition restricts that the interleaving of low transition events may not depend on secret information, confines the content of information declassified in accord with the content allowed to be released, and controls the location of declassification only through the special release statement. Moreover, we proposed monitoring mechanisms of policy enforcement and proved its soundness.

Keywords: Declassification, information flow, non-interference, information release

1. Introduction

Confidentiality means secret information must not be leaked by computation. Access control is a standard mechanism for achieving confidentiality, but it only controls the release of information. Information-flow control additionally restricts how information is propagated. There are two basic kinds of information flows through program constructs: explicit flow and implicit flow, where the former represents that information is passed explicitly through an assignment, and the latter denotes that information is passed via control-flow structures or covert channels [1, 2]. Confidentiality requirements of programs could be expressed by a information-flow control policy, in which a baseline policy is noninterference which says that an attacker can deduce nothing about the secret inputs from the public outputs [3]. However, the restrictiveness of noninterference is too strong for many practical systems. For instance, password checking programs inevitably violate noninterference because these programs reveal some information of passwords by rejecting an illegitimate login.

Declassification policies relax the restrictiveness of noninterference and ensure trusted release of secret information to public environment along four dimensions: WHAT (what can be released), WHO (who can initiate the declassification), WHERE and WHEN (where and when the declassification can occur) [4]. It would be desirable to integrate different dimensions for controlling declassification to offer enhanced protection from the illegal release of secret information, but these dimensions are largely orthogonal to each other, so a tighter integration of these dimensions remains a challenge.

Our previous work [5] has proposed a declassification policy called gradually delimited release combining WHAT and WHERE dimensions, where the security condition in WHAT dimension controls the amount of information released can not exceed deliberate release, and the security condition in WHERE dimension restricts the information release is only through the special release statements, moreover, this work has presented the enforcement mechanisms of this policy; however, it was limited to the sequential programs.

Existing declassification security specifications and enforcement mechanisms focus on sequential programs [6, 7, 8]. They do not generalize naturally to multi-threaded programs and declassification in multi-threaded programs is not yet equally well understood. However, much work has been done for the noninterference in multi-threaded programs [9, 10, 11, 12], but these results were limited to information flow properties that forbid declassification. With this paper, we close this gap by extending gradually delimited release policy to the multi-threaded programs.

* Corresponding author e-mail: searain@nuaa.edu.cn

The rest of the paper is organized as follows. Section 2 presents the model of multi-threaded language. Section 3 states the scheduler model. Section 4 describes security specifications of gradually delimited release policy in multi-threaded programs. Section 5 depicts dynamic enforcement mechanisms of this policy and proves its soundness. Finally, Section 6 concludes this paper.

2. Multi-threaded Language Model

We enrich the language provided by Russo and Sabelfeld [9] with declassification primitive and some auxiliary commands, and name the enriched language **MWhile**. In this language, each variable and constant is annotated by a security label indicating its confidentiality level. We introduce typing environment Γ which is a function mapping each variable to its confidentiality level. We assume set $\mathscr{L}$ of confidentiality levels, ordered by a reflexive transitive relation $\sqsubseteq$ that denotes the relative restrictiveness of the levels, and $\langle \mathscr{L}, \sqsubseteq \rangle$ is a lattice. For simplicity but without loss of generality, we consider two-element set $\mathscr{L} = \{low, high\}$ where $low \sqsubseteq high$. The join operation $\sqcup$ is used to calculate the least upper bound on confidentiality levels. The confidentiality level of an expression is the least upper bound on the confidentiality levels of its sub-expressions.

The syntax of the language **MWhile** is displayed in Figure 1 and it is composed of expressions and commands. Expression e comprises constant value n, variable x, composite expression $e \oplus e$, where $\oplus$ is a binary operation. Command c includes atomic command, branching command and composition command. The atomic command $x := \mathbf{declassify}(e)$ downgrades the high level of expression e (called escape hatch expression) to the low level, and then assigns it to the variable x. The language supports two kinds of threads: low and high threads, partitioning the thread-pool into low and high parts. The atomic command **hide** upgrades the level of current thread from *low* to *high*, and the command **unhide** has the dual effect: it downgrades the level of current thread from *high* to *low*. The commands $\mathbf{fork}(c, \bar{d})$ and $\mathbf{hfork}(c, \bar{d})$ is used to generate a collection of threads $\bar{d}$ while the current running thread is command c. The difference between the two commands is that the former command creates low threads and the latter spawns high threads. Additionally, there are two auxiliary commands: **stop** and **end**, where the command **stop** signifies the termination of a command, and the command **end** denotes exiting the scope of a branching command. These auxiliary commands are not accessible to programmers and only used in command semantics.

Program memory m (mapping variables to values) can be divided into two parts: low memory m_L and high memory m_H, where m_L (resp. m_H) restricts the mapping to variables whose confidentiality level is *low* (resp. *high*). Memories m_1 and m_2 are indistinguishable at low memory, i.e., $m_{1L} = m_{2L}$, iff $\forall x. \Gamma(x) = low \Rightarrow m_1(x) = m_2(x)$.

A command configuration has the form $\langle m, c \rangle$, meaning that the command c is to be executed in the memory m. We use $\bar{c}$ to model a thread pool. A terminated command configuration with the memory m is denoted by $\langle m, stop \rangle$. A small-step transition between command configurations has the form $\langle m, c \rangle \xrightarrow{\alpha}_{\gamma} \langle m', c' \rangle$, where α is an internal event and γ is a low transition event, and they are triggered by the command c. The syntax of α is defined by the following grammar:

$$\alpha ::= s \mid a(x,e) \mid d(x,e) \mid b(e,c) \mid f \mid \rightsquigarrow L \mid \rightsquigarrow H \mid L_{\bar{d}} \mid H_{\bar{d}}$$

where event s signals command **skip** is performed; event $a(x,e)$ represents an assignment to variable x of expression e; event $d(x,e)$ records declassifying of expression e into variable x; event $b(e)$ originates from branching commands, denoting the current command branches on expression e; event f indicates that the structure block of a branching command has finished execution; event $\rightsquigarrow L$ (resp. $\rightsquigarrow H$) shows that command **unhide** (resp. **hide**) is run; event $L_{\bar{d}}$ (resp. $H_{\bar{d}}$) signals about the creation of low (resp. high) threads $\bar{d}$. Additionally, low transition event is triggered by one of internal events: $a(x,e)$ and $d(x,e)$ requiring $\Gamma(x) = low$, and it has the form $(x, m(e))$, meaning variable x with low confidentiality is updated with value $m(e)$. The low transition event triggered by $d(x,e)$ is called a release event. We denote a sequence (possible empty) of transition steps by $\langle m, c \rangle \xrightarrow{*}_{\bar{\gamma}} \langle m', c' \rangle$, where $\bar{\gamma}$ represents a sequence of low transition events. We can write the form $\langle m, c \rangle \xrightarrow{*}_{\bar{\gamma}}$ when the resulting configuration is unimportant. If a transition step does not affect low memory, the low transition event is empty.

$$\begin{aligned} e &::= n \mid x \mid e \oplus e \\ c &::= \mathbf{skip} \mid x := e \mid x := \mathbf{declassify}(e) \\ &\mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid c; c \\ &\mid \mathbf{hide} \mid \mathbf{unhide} \mid \mathbf{fork}(c, \bar{d}) \mid \mathbf{hfork}(c, \bar{d}) \\ &\mid \mathbf{stop} \mid \mathbf{end} \mid \mathbf{sleep}(n) \end{aligned}$$

Figure 1: Syntax of the language **MWhile**

3. Scheduler Model

In multi-threaded programs, CPU scheduling becomes necessary when the number of threads exceeds the number of available processing units. In this paper, we assume that multi-threaded programs run on a single-core CPU with a shared memory for inter-thread communication. In this section we present a scheduler model that can be instantiated for a wide class of schedulers [9], and use a labeled transition system to describe the behavior of the scheduler model. Scheduler configurations have the form $\langle \sigma, v \rangle$,

where σ is the scheduler program and ν is a scheduler memory which is disjoint from the program memory m. In the scheduler memory, we use variable q to show the how many steps a thread can be scheduled and variable t to define the set of active threads which consist of low threads and high threads, denoted by variable t_L and t_H respectively. Additionally, we use variable r to represent the current running thread and variable k to regulate whether low threads may be scheduled. When the value of k is L, both low and high threads may be scheduled. However, when the value of k is H, only high threads could be scheduled to close timing covert channels. A transition between scheduler configurations has the form $\langle \sigma, \nu \rangle \xrightarrow{\beta} \langle \sigma', \nu' \rangle$, where β is the scheduler event.

The semantics for the scheduler is displayed in Figure 2. Concretely, rule (S-CREATE) describes that the scheduler updates the thread-pool according to the scheduler event $H_{\bar{d}}^{r}$ and $L_{\bar{d}}^{r}$ triggered by the **hfork** and **fork** commands respectively. Rule (S-STOP) depicts the termination of thread r results in event $r \rightsquigarrow \times$, which requiring the scheduler to remove thread r from the thread-pool. Rules (S-SELECT-L) and (S-SELECT-H) show that the scheduler's selection of thread r' to be scheduled depending on the value of variable k as discussed above. Rule (S-UP) states that the scheduler handles event $r \rightsquigarrow H$ by moving the current thread r from the t_L to t_H and setting k' to H to ensure the next scheduled thread is high. Rule (S-DOWN) is opposite to rule (S-UP), but sets q' to 0 to end the current thread's execution for the purpose of preventing timing covert channels. Rule (S-TRANS) ensures that for event $r \rightsquigarrow$ triggered by a non-terminal step of thread r, variable q is decremented by one. The semantics of the interaction between threads and the scheduler boils down to a single rule as follows:

$$\frac{\langle m, c \rangle \xrightarrow{\alpha}_{\gamma} \langle m', c' \rangle \quad \langle \sigma, \nu \rangle \xrightarrow{\beta} \langle \sigma', \nu' \rangle}{\langle m, \nu, \bar{c}, \sigma \rangle \rightarrow_{\gamma} \langle m', \nu', \bar{c}', \sigma' \rangle}$$

where c is a thread in the thread pool $\bar{c}$, and $\langle m, \nu, \bar{c}, \sigma \rangle$ is thread-pool configuration. The rule ensures that a thread c is allowed to perform a step with internal event α only if the scheduler σ schedules this thread with the scheduler event β triggered by thread c. As a notational convention, we use $\langle m, \nu, \bar{c}, \sigma \rangle \xrightarrow{*}_{\bar{\gamma}} \langle m', \nu', \bar{c}', \sigma' \rangle$ to denote a sequence (possible empty) of transition steps. The resulting configuration could be omitted when it is unimportant.

4. Security Specifications

By convention, we assume that an attacker can observe low transition events and low memories, and can inject attack codes excluding commands of declassification into programs. Attacker knowledge is described by the set of initial high memories compatible with observations of low transition events.

Definition 1 Given a scheduler σ, scheduler memory ν

(S-CREATE): $$\frac{q' = q - 1 \quad t'_X = t_X \cup \bar{d} \quad X \in \{H, L\}}{\langle \sigma, \nu \rangle \xrightarrow{X_{\bar{d}}^{r}} \langle \sigma', \nu' \rangle}$$

(S-STOP): $$\frac{q' = 0 \quad t'_X = t_X \setminus \{r\} \quad X \in \{H, L\}}{\langle \sigma, \nu \rangle \xrightarrow{r \rightsquigarrow \times} \langle \sigma', \nu' \rangle}$$

(S-SELECT-L): $$\frac{q = 0 \quad k = L \quad q' > 0 \quad r' \in t_L \cup t_H}{\langle \sigma, \nu \rangle \xrightarrow{\uparrow r'} \langle \sigma', \nu' \rangle}$$

(S-SELECT-H): $$\frac{q = 0 \quad k = H \quad q' > 0 \quad r' \in t_H}{\langle \sigma, \nu \rangle \xrightarrow{\uparrow r'} \langle \sigma', \nu' \rangle}$$

(S-UP): $$\frac{q' = q - 1 \quad k' = H \quad t'_L = t_L \setminus \{r\} \quad t'_H = t_H \cup \{r\}}{\langle \sigma, \nu \rangle \xrightarrow{r \rightsquigarrow H} \langle \sigma', \nu' \rangle}$$

(S-DOWN): $$\frac{q' = 0 \quad k' = L \quad t'_L = t_L \cup \{r\} \quad t'_H = t_H \setminus \{r\}}{\langle \sigma, \nu \rangle \xrightarrow{r \rightsquigarrow L} \langle \sigma', \nu' \rangle}$$

(S-TRANS): $$\frac{q' = q - 1}{\langle \sigma, \nu \rangle \xrightarrow{r \rightsquigarrow} \langle \sigma', \nu' \rangle}$$

Figure 2: Semantics of the scheduler

and a sequence of low transition events $\bar{o}$ produced during the execution of thread pool $\bar{c}$ from initial program memory m, attacker knowledge is defined as follows:

$$k(m, \nu, \bar{c}, \sigma, \bar{o}) = \{m'_H | m_L = m'_L \wedge (\langle m', \nu, \bar{c}, \sigma \rangle \xrightarrow{*}_{\bar{o}} \langle m'', \nu', stop, \sigma' \rangle \vee \langle m', \nu, \bar{c}, \sigma \rangle \xrightarrow{*}_{\bar{o} \cdot o})\} \quad (1)$$

where $\bar{o} \cdot o$ denotes the catenation of $\bar{o}$ and o.

The above definition is termination-insensitive [6] because the attacker knowledge is not refined for observing that program does not enter an infinite loop. Attackers can obtain input information of high memory only by seeing terminating configuration or sequence of low transition events followed by one more low transition event. Attacker knowledge is monotonic along the sequence of low transition events. The smaller the attacker knowledge set, the more information obtained by the attacker. Based on the definition of attacker knowledge, we extend the definition of the gradually delimited release policy proposed in our previous work [5] to the multi-threaded programs as follows:

Definition 2 Given a scheduler σ satisfying noninterference, a scheduler memory ν, a thread pool $\bar{c}$ starts running from an initial program memory m_0, producing a sequence of low transition events $\mathbf{o}_n = o_1 \cdots o_n$ $(n \geq 1)$, from which we extract all release events to form a sequence $o_{r_1} \cdots o_{r_k}$, where $1 \leq r_i < r_{i+1} \leq n$ and $1 \leq i < k$, and its corresponding sequence of escape hatch expressions is $e_{r_1} \cdots e_{r_k}$. We use m^{r_i} to denote the program memory when o_{r_i} is just generated. Thread pool $\bar{c}$ satisfies gradually delimited release policy if for all i and j where

$1 \leq i \leq n, 1 \leq j \leq k$ and $i \neq r_j$ then

$$\begin{pmatrix} k(m_0, v, \overline{c}, \sigma, \overline{o_{i-1}}) = k(m_0, v, \overline{c}, \sigma, \overline{o_i}) \\ \wedge\ m_0(e_{r_j}) = m^{r_j}(e_{r_j}) \end{pmatrix} \quad (2)$$

where $k(m_0, v, \overline{c}, \sigma, \overline{o_0})$ is initial attacker knowledge, corresponding to all possible initial high memories that lead to termination.

The security provided by security condition (2) can be illustrated with some examples, where h_1, h_2, and h_3 are high variables and l_1, l_2 and l_3 are low variables. Consider the following thread commands c_1 and c_2:

$c_1 : h_2 := h_1; h_3 := h_1;$
$c_2 : l_1 := 6;$ **skip**;
$l_2 :=$ **declassify**$((h_1 + h_2 + h_3)/3); l_3 := 8; //(A)$

It is not difficult to conclude that commands c_1 and c_2 satisfy condition (2), respectively. However, the parallel composition of the two thread commands does not necessarily satisfy condition (2). The scheduler might schedule c_1 before the command in line A is executed, i.e., the executed sequence might be:

$l_1 := 6;$ **skip**$; h_2 := h_1; h_3 := h_1;$
$l_2 :=$ **declassify**$((h_1 + h_2 + h_3)/3); l_3 := 8;$

Assume $m_0(h_1) = 4$, $m_0(h_2) = 2$ and $m_0(h_3) = 9$ where m_0 is the initial program memory, we have $m_0((h_1 + h_2 + h_3)/3) = 5$. When the above executed sequence proceeds to the declassification command in line A, the value of the escape hatch expression will be updated to $m((h_1 + h_2 + h_3)/3) = 4$, where m is current program memory. Therefore, the above thread commands c_1 and c_2 are rejected by condition (2). Consider another pair of thread commands c_3 and c_4:

$c_3 :$ **if** $h_1 > 0$ **then sleep**$(100);$ **else skip**$; l_1 := 1;$
$c_4 :$ **sleep**$(50); l_1 := 0;$

Attacker will attain whether the value of h_1 was positive through the low transition event $(l_1, 0)$ or $(l_1, 1)$ under the scheduler model presented in section 3, so the two thread commands are insecure according to condition (2). Now consider variations c_3' and c_4' of the commands c_3 and c_4 :

$c_3' :$ **hide**;
if $h_1 > 0$ **then sleep**$(100);$ **else skip**;
unhide;
$l_1 := 1;$
$c_4' :$ **sleep**$(50); l_1 := 0;$

According to semantics of the scheduler presented in Figure 2, attacker will learn nothing about the information of h_1 through any low transition event. These variations are secure according to condition (2).

5. Enforcement Mechanisms and Soundness

Figure 3 presents dynamic monitoring rules that enforce the security specification from the previous section. A monitor configuration has the form $\langle m_0, \Gamma, st, hc \rangle$, where m_0 is initial program memory, Γ is a typing environment, st is a stack used to keep track of the implicit flow, and hc is a variable that tracks confidentiality level of current thread. A transition between monitor configurations has the form $\langle m_0, \Gamma, st, hc \rangle \xrightarrow{\alpha} \langle m_0, \Gamma', st', hc' \rangle$, where α is an internal event captured by the monitor. We use the notation :: to denote pushing an element to a stack, and $lev(st)$ to denote the top element of the stack st; for example, $lev(l :: st) = l$.

(T-SKIP): $\langle m_0, \Gamma, st, hc \rangle \xrightarrow{s} \langle m_0, \Gamma, st, hc \rangle$

(T-PUSH): $\langle m_0, \Gamma, st, hc \rangle \xrightarrow{b(e)} \langle m_0, \Gamma, \Gamma(e) \sqcup lev(st) :: st, hc \rangle$

(T-POP): $\langle m_0, \Gamma, l :: st, hc \rangle \xrightarrow{f} \langle m_0, \Gamma, st, hc \rangle$

(T-ASSIGN): $$\frac{\Gamma(e) \sqsubseteq \Gamma(x) \quad lev(st) \sqsubseteq \Gamma(x) \quad hc \sqsubseteq \Gamma(x) \quad lev(st) \sqsubseteq hc}{\langle m_0, \Gamma, st, hc \rangle \xrightarrow{a(x,e)} \langle m_0, \Gamma, st, hc \rangle}$$

(T-DECL): $$\frac{m_0(e) = m(e) \quad lev(st) = hc = low}{\langle m_0, \Gamma, st, hc \rangle \xrightarrow{d(x,e)} \langle m_0, \Gamma, st, hc \rangle}$$

(T-HIDE): $\langle m_0, \Gamma, st, hc \rangle \xrightarrow{\rightsquigarrow H} \langle m_0, \Gamma, st, hc := high \rangle$

(T-UNHIDE): $\langle m_0, \Gamma, st, hc \rangle \xrightarrow{\rightsquigarrow L} \langle m_0, \Gamma, st, hc := low \rangle$

(T-FORK): $$\frac{hc = low}{\langle m_0, \Gamma, st, hc \rangle \xrightarrow{L^r_d} \langle m_0, \Gamma, st, hc \rangle}$$

(T-HFORK): $$\frac{hc = high}{\langle m_0, \Gamma, st, hc \rangle \xrightarrow{H^r_d} \langle m_0, \Gamma, st, hc \rangle}$$

Figure 3: Monitoring rules

Rule (T-SKIP) indicates that internal event s is always accepted by the monitor without updating the state of the monitor configuration. Rule (T-PUSH) presents that the value of expression $\Gamma(e) \sqcup lev(st)$, which denotes the least upper bound of all the levels of branch expressions that the current command implicitly depends on, should be pushed into the stack st when internal event $b(e)$ is captured. Rule (T-POP) denotes the pop operation of the stack when internal event f is captured. Rule (T-ASSIGN) represents that internal event $a(x,e)$ is accepted by the monitor on the premise that the confidentiality level of expression e, the top element of stack st, and the value of variable hc are no greater than the confidentiality level of variable x and the top element of stack st is no greater than the value of variable hc. Rule (T-DECL) shows internal event $d(x,e)$ is accepted without changes in the state of monitor configuration but with two conditions: (i) that the value of es-

cape hatch expression in initial program memory is indistinguishable from the value of this expression in current program memory and (ii) that the top element of the stack and the value of variable *hc* equal *low*. Rules (T-HIDE) and (T-UNHIDE) state that internal events $\rightsquigarrow H$ and $\rightsquigarrow L$ result in updating the value of variable *hc*, respectively. Rules (T-FORK) and (T-HFORK) ensure that internal events $L_{\bar{d}}^{r}$ and $H_{\bar{d}}^{r}$ are accepted by the monitor on the premise that the value of variable *hc* equals to *low* and *high* respectively.

The overall programming discipline enforced by the monitoring rules ensures that monitored execution of programs are secure, which is formalized by the following theorem.

Theorem 1 Given a thread pool $\bar{c}$ with an initial program memory m_0, a non-interfering scheduler σ with a scheduler memory ν, and a sequence of low transition events $\bar{o_n}(n \geq 0)$ produced by the $\langle m_0, \nu, \bar{c}, \sigma \rangle$ while monitored by rules in Figure 3, we have that the monitored execution of $\bar{c}$ satisfies gradually delimited release policy.

Proof Assume all release events from $\bar{o_n}$ to form a sequence $o_{r_1} \cdots o_{r_k} (1 \leq r_i < r_{i+1} \leq n \wedge 1 \leq i < k)$, and corresponding sequence of escape hatch expressions is $e_{r_1} \cdots e_{r_k}$. We use m^{r_i} to denote the current program memory when o_{r_i} is just generated. We sketch a proof by induction on the length of the sequence of low transition events $\mathbf{o}_n$.

1) Base n=0. The condition (2) trivially holds.

2) Induction step. Assume the theorem holds for $\bar{o}_{n-1}$ and need to prove that the theorem also holds for $\bar{o_n}$, i.e., the condition (2) is satisfied for $\bar{o_n}$. We consider two cases for the low transition event o_n.

Case 1: o_n is an assignment to the variable in low memory. By the premises $\Gamma(e) \sqsubseteq \Gamma(x)$ and $lev(st) \sqsubseteq \Gamma(x)$ in monitoring rule (T-ASSIGN), we have that this event will not result in illegal explicit and implicit flow. Additionally, the premises $hc \sqsubseteq \Gamma(x)$ and $lev(st) \sqsubseteq hc$ in monitoring rule (T-ASSIGN) exclude an assignment to the low variable in a high thread and implicit flow from high thread to low thread, preventing timing covert channels. Hence the attacker knowledge and low memory is unchanged, and condition (2) is satisfied.

Case 2: o_n is a release event. In this case there is no demand on attacker knowledge equivalence, i.e., $k(m_0, \nu, \bar{c}, \sigma, \bar{o}_{i-1}) = k(m_0, \nu, \bar{c}, \sigma, \bar{o_i})$ is trivially satisfied. Additionally, we assume that e_n is the escape hatch expression corresponding to release event o_n. By the premises of monitoring rule (T-DECL), we have $m_0(e_n) = m(e_n)$ immediately. Hence, condition (2) is satisfied.

6. Conclusions

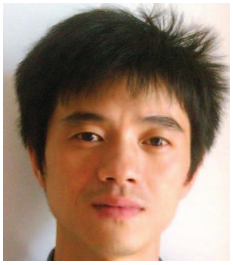
Declassification policies relax noninterference policy such that a deliberate release of some secret information becomes possible. In order to ensure trusted release of secret information, many declassification policies and their enforcement mechanisms have been proposed in recent years, but they focused on sequential programs. In this paper, we have extended the gradually delimited release policy to multi-threaded programs. The extended policy can rule out dangers of internal covert channel in concurrent programs and ensure declassification of proper content to proper location in the multi-threaded programs. Additionally, we have presented sound monitoring rules to enforce the policy. With this paper, we hope to contribute foundations that lead to a better applicability of declassification polices in practice.

Acknowledgements

This work is partially supported by the National Natural Science Foundation of China under Grant No. 61202006, the Fundamental Research Funds for the Central Universities under Grant No. NZ2013306. Thanks for the help.

References

[1] A. Sabelfeld and A. C. Myers, Language-based information flow security, Selected Areas in Communications, **21**, 5-19 (2003).
[2] A. Sabelfeld and A. Russo, From dynamic to static and back: riding the roller coaster of information-flow control research, Perspectives of Systems Informatics, LNCS, **5947**, 352-365 (2010).
[3] J. A. Goguen and J. Meseguer, Security policies and security models, Proc. IEEE Symp. on Security and Privacy, 11-20 (1982).
[4] A. Sabelfeld and D. Sands, Declassification: dimensions and principles, Journal of Computer Security, **17**, 517-548 (2009).
[5] H. Zhu, Y. Zhuang and Y. Xue, et al., A two-dimension policy of confidential information release, Journal of Information and Computational Science, **8**, 3239-3247 (2011).
[6] A. Askarov and A. C. Myers, A semantic framework for declassification and endorsement, Programming Languages and Systems, LNCS, **6012**, 64-84 (2010).
[7] A. Lux and H. Mantel, Declassification with explicit reference points, Proc. 14th European Symp. on Research in Computer Security, 69-85 (2009).
[8] A. Askarov and A. Sabelfeld, Tight enforcement of information-release policies for dynamic languages, Proc. 22nd IEEE Symp. on Computer Security Foundations, 43-59 (2009).
[9] A. Sabelfeld and A. Russo, Securing interaction between threads and the scheduler, Proc. 19th IEEE Computer Security Foundations Workshop, 177-189 (2006).
[10] G. Boudol and i. CastellanI, Noninterference for concurrent programs and thread systems. Theoretical Computer Science, **281**, 109-130 (2002).
[11] A. Russo and A. Sabelfeld, Security for multithreaded programs under cooperative scheduling, Perspectives of Systems Informatics, LNCS, **4378**, 474-480 (2007).
[12] M. Huisman and H.-C. Blondeel, Model-checking secure information flow for multi-threaded programs, Theory of Security and Applications, LNCS, **6993**, 148-165 (2012).



Hao Zhu received his M.Sc. degree in 2005 from Jiangsu University. He is a Ph.D. candidate in Nanjing University of Aeronautics and Astronautics. He is a associate professor of computer science and technology in Nantong university. His research interests include information security and intelligent computing.

Yi Zhuang is a professor and Ph.D. supervisor of computer science and technology in Nanjing University of Aeronautics and Astronautics. Her research interests include information security, trusted computing, distributed computing, computer network and wireless senor network et al.

Xiang Chen received a Ph.D. degree in Nanjing University. He is a lecturer of computer science and technology in Nantong university. His research interests include software testing and program analysis.