Comparison of the Expressive Power of Language-based Access Control Models

Yoshiaki Takata
Dept. of Information Systems Engineering
Kochi University of Technology

Hiroyuki Seki
Graduate School of Information Science
Nara Institute of Science and Technology

Abstract

This paper compares the expressive power of five language-based access control models. We show that the expressive powers are incomparable between any pair of history-based access control, regular stack inspection and shallow history automata. Based on these results, we introduce an extension of HBAC, of which expressive power exceeds that of regular stack inspection.

Keywords history-based access control, stack inspection, shallow history automaton, expressive power

1 Introduction

To protect secure information against malicious access, it is desirable to incorporate a runtime access control mechanism in a host language. This approach is called *language-based access control*, and a few models have been proposed [1, 5, 6, 9]. A common feature of these models is that the history of execution such as method invocation and resource access is used for access control. *Stack inspection* provided in the Java virtual machine [6] is one of the best-known such control mechanisms. In stack inspection, a set of permissions is assigned statically to each method and when the control reaches a statement for checking permissions, it is examined whether or not every method on the runtime stack has the permissions specified by the statement. Stack inspection has been extended in several ways. For example, stack pattern can be specified by LTL formula in [7] and regular language in [4, 8]. Automatic verification methods for a program with stack inspection are also discussed in [4, 7, 8]. Abadi and Fournet [1] pointed out the problem of stack inspection, which completely cancels the effect of the finished method execution. They proposed a new control mechanism called *history-based access control* (HBAC). In HBAC, current permissions are modified each time a method is invoked, and they may depend on all the methods executed so far. Verification of HBAC programs is also discussed in [2, 3, 10]. Meanwhile, Schneider [9] defines *security automata*, and later Fong [5] defines *shallow history automata* as a subclass of finite-state security automata. Fong showed that the expressive powers of shallow history automata and regular stack inspection are incomparable. However, the relations among the control models mentioned so far have not been fully clarified.

In this paper, we first define five of the existing control mechanisms in a simple and uniform framework based on control flow graph. Next, we introduce a trace equivalence relation among programs, and compare the expressive power of the five subclasses of programs. In particular, the expressive powers are incomparable between any pair of history-based access control, regular stack inspection and shallow history automata. Based on these results, we introduce an extension of HBAC, of which expressive power exceeds that of regular stack inspection.

2 Definitions

2.1 HBAC program

An HBAC program is a tuple $\pi = (Mhd, f_0, \{G_f \mid f \in Mhd\}, PRM)$ where Mhd is a finite set of method names, $f_0 \in Mhd$ is the main method name, G_f ($f \in Mhd$) is a *control flow graph* of f defined below and PRM is a finite set of *permissions*. G_f is a directed graph $(NO_f, TG_f, IS_f, IT_f, SP_f)$ where NO_f is a finite set of nodes, $TG_f \subseteq NO_f \times NO_f$ is a set of *transfer edges*, $IS_f : NO_f \to \{call_g[P_G, P_A] \mid g \in Mhd, P_G \subseteq SP_f, P_A \subseteq SP_f\} \cup \{check[P] \mid P \subseteq PRM\} \cup \{return, nop\}$ is a labeling function for nodes, $IT_f \subseteq NO_f$ is a set of *initial nodes*, which represents the set of entry points of method f, and $SP_f \subseteq PRM$ is a subset of permissions assigned to f before runtime (*static permissions*). NO_f is divided into four subsets by IS_f as follows.

- $IS_f(n) = call_g[P_G, P_A]$. Node n is a *call node* that represents a call to method g. Parameters P_G and P_A are called *grant permissions* and *accept permissions*, respectively.
- $IS_f(n) = return$. Node n is a *return node* that represents a return to the caller method.
- $IS_f(n) = check[P]$ where $P \subseteq PRM$. Node n is a *check node* that represents a test for the current permissions. For $p \in PRM$, $check[\{p\}]$ is abbreviated as $check[p]$.

- $IS_f(n) = nop$. Node n is a *nop node* with no effect.

We write $n \to n'$ for $n, n' \in NO_f$ if $\langle n, n' \rangle \in TG_f$. Let $NO = \bigcup_{f \in Mhd} NO_f$ and $IS = \bigcup_{f \in Mhd} IS_f$. For $n \in NO$, also let $in(n) = \{ n' \mid n' \to n \}$ and $out(n) = \{ n' \mid n \to n' \}$.

In the figures in this paper, a dotted arrow denotes a transfer edge and a solid arrow connects between a call node and the initial node(s) of the callee method. Also, a method is surrounded by a rectangle and a set beside the rectangle denotes the static permissions of the method.

A state of π is a pair $\langle n, C \rangle$ of a node $n \in NO$ and a subset of permissions $C \subseteq PRM$. A *configuration* of π is a finite sequence of states, which is also called a *stack*. The concatenation of state sequences ξ_1 and ξ_2 is denoted as $\xi_1 : \xi_2$. The semantics of an HBAC program is defined by the transition relation $\Rightarrow$ over the set of configurations, which is the least relation satisfying the following rules.

$$\frac{IS(n) = call_g[P_G, P_A],\ n' \in IT_g}{\xi : \langle n, C \rangle \Rightarrow \xi : \langle n, C \rangle : \langle n', (C \cup P_G) \cap SP_g \rangle}$$

$$\frac{IS(m) = return,\ IS(n) = call_g[P_G, P_A],\ n \to n'}{\xi : \langle n, C \rangle : \langle m, C' \rangle \Rightarrow \xi : \langle n', C \cap (C' \cup P_A) \rangle}$$

$$\frac{IS(n) = check[P],\ P \subseteq C,\ n \to n'}{\xi : \langle n, C \rangle \Rightarrow \xi : \langle n', C \rangle}$$

$$\frac{IS(n) = nop,\ n \to n'}{\xi : \langle n, C \rangle \Rightarrow \xi : \langle n', C \rangle}$$

The rule of *nop* for the other program subclasses in the following subsections is the same as above and will be omitted below. For a configuration $\langle n_1, C_1 \rangle : \ldots : \langle n_\ell, C_\ell \rangle$, the stack top is $\langle n_\ell, C_\ell \rangle$ where n_ℓ and C_ℓ are called the *current program point* and the *current permissions* of the configuration, respectively. The *trace set* of π is defined as $[\![\pi]\!] = \{ n_0 n_1 \ldots n_k \mid n_0 \in IT_{f_0},\ \exists C_1, \ldots, C_k \subseteq PRM,\ \exists \xi_1, \ldots, \xi_k \in (NO \times 2^{PRM})^*,\ \xi_i : \langle n_i, C_i \rangle \Rightarrow \xi_{i+1} : \langle n_{i+1}, C_{i+1} \rangle$ for $0 \le i < k,\ C_0 = SP_{f_0},\ \xi_0 = \varepsilon \}$, where ε denotes the empty sequence. For a set S of sequences, let prefix(S) denote the set of all nonempty prefixes of sequences in S.

Example 1. Chinese wall policy is a policy such that a user has access permission to any resources, but once the user has accessed one of the resources, (s)he loses access permission to the resources belonging to competing parties. A simplified Chinese wall policy can be represented by program π in Fig. 1. If the control reaches n_{1A} and n_{1A} calls A, then the current permissions lose permission p_B. Thus, if n_{2B} calls B afterward, the check at m_{0B} fails. The same situation occurs when B and A are called in this order. In fact, $[\![\pi]\!]$ = prefix(

$$n_0 n_{1A} m_{0A} m_{1A} (n_{2A} m_{0A} m_{1A} n_3 + n_{2B} m_{0B}) \\ + n_0 n_{1B} m_{0B} m_{1B} (n_{2B} m_{0B} m_{1B} n_3 + n_{2A} m_{0A})),$$

where the argument of 'prefix' is specified by a regular expression and + denotes the union operator.

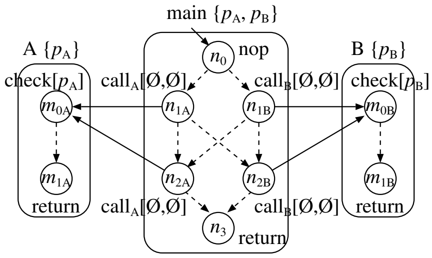


Figure 1 An HBAC program

2.2 JVM and R-SI programs

A program with *Java stack inspection* (abbreviated as JVM program) has a form $\pi = (Mhd, f_0, \{G_f \mid f \in Mhd\}, PRM, PRV)$ similar to an HBAC program such that $G_f = (NO_f, TG_f, IS_f, IT_f, SP_f)$ where each component of G_f is the same as that of an HBAC program, except that the label $IS_f(n)$ of each call node n is simply $call_g$ ($g \in Mhd$) without P_G or P_A, and a set of privileged nodes $PRV \subseteq NO$ is specified. The semantics of π is defined as follows. (The rule for *check* is the same as HBAC programs.)

$$\frac{IS(n) = call_g,\ n \notin PRV,\ n' \in IT_g}{\xi : \langle n, C \rangle \Rightarrow \xi : \langle n, C \rangle : \langle n', C \cap SP_g \rangle}$$

$$\frac{IS(n) = call_g,\ n \in PRV \cap NO_f,\ n' \in IT_g}{\xi : \langle n, C \rangle \Rightarrow \xi : \langle n, C \rangle : \langle n', SP_f \cap SP_g \rangle}$$

$$\frac{IS(m) = return,\ n \to n'}{\xi : \langle n, C \rangle : \langle m, C' \rangle \Rightarrow \xi : \langle n', C \rangle}$$

A *regular stack inspection* (R-SI) program $\pi = (Mhd, f_0, \{G_f \mid f \in Mhd\})$ is introduced in [4, 8] as an extension of a JVM program where $G_f = (NO_f, TG_f, IS_f, IT_f)$. Its semantics is given by the following rules.

$$\frac{IS(n) = call_g,\ n' \in IT_g}{\xi : n \Rightarrow \xi : n : n'}$$

$$\frac{IS(m) = return,\ n \to n'}{\xi : n : m \Rightarrow \xi : n'}$$

$$\frac{IS(n) = check[R],\ \xi : n \in R,\ n \to n'}{\xi : n \Rightarrow \xi : n'}$$

where $R \subseteq (NO)^*$ is a regular language over NO. The trace set of a JVM or R-SI program is defined in the same way as that of an HBAC program except that current permissions are missing in R-SI.

2.3 F-SA and SHA Programs

A *finite security automaton* (F-SA) [9] is just a deterministic finite automaton (DFA) $M = (\Sigma, Q, q_0, \delta)$ without final states where Σ is a finite set of input symbols, Q is a finite set of states, $q_0 \in Q$ is the initial state and δ is a state transition function, which is a partial function from $Q \times \Sigma$ to Q. We write $\delta(q, a) = \bot$ if $\delta(q, a)$ is undefined. A *shallow history automaton* (SHA) [5] is an F-SA $M = (\Sigma, Q, q_0, \delta)$ such that $Q = 2^{\Sigma}$ and $q_0 = \emptyset$ and if $\delta(q, a) \neq \bot$ then $\delta(q, a) = q \cup \{a\}$.

An F-SA program is a tuple $(Mhd, f_0, \{G_f \mid f \in Mhd\}, M)$ without permissions or check nodes where $G_f = (NO_f, TG_f, IS_f, IT_f)$ $(f \in Mhd)$ and $M = (\Sigma, Q, q_0, \delta)$ is an F-SA such that $\Sigma = \{f, \overline{f} \mid f \in Mhd\}$. The semantics of an F-SA program is defined as follows.

$$\frac{IS(n) = call_g,\ n' \in IT_g,\ \delta(q, g) \neq \bot}{\langle \xi : n, q\rangle \Rightarrow \langle \xi : n : n', \delta(q, g)\rangle}$$

$$\frac{IS(m) = return,\ m \in NO_g,\ n \to n',\ \delta(q, \overline{g}) \neq \bot}{\langle \xi : n : m, q\rangle \Rightarrow \langle \xi : n', \delta(q, \overline{g})\rangle}$$

The trace set of an F-SA program π is defined as $[\![\pi]\!] = \{\, n_0 n_1 \ldots n_k \mid n_0 \in IT_{f_0},\ \exists q_1, \ldots, q_k \subseteq Q,\ \exists \xi_1, \ldots, \xi_k \in NO^*, \langle \xi_i : n_i, q_i\rangle \Rightarrow \langle \xi_{i+1} : n_{i+1}, q_{i+1}\rangle$ for $0 \le i < k,\ \xi_0 = \varepsilon \,\}$.

3 Expressive Power

A program without check nodes, permissions or privileged nodes is called a *basic program*. Let $\alpha \in$ {HBAC, R-SI, JVM, F-SA, SHA}. An α program π is an *extension* of a basic program π_0 if π_0 is obtained from π by the following operations.

(S1) Delete each check node n (if α = HBAC, R-SI or JVM). At the same time, for any pair of $n_1 \in in(n)$ and $n_2 \in out(n)$, add a transfer edge $n_1 \to n_2$. Moreover, if $n \in IT_f$ for some $f \in Mhd$, then add every $n_2 \in out(n)$ into IT_f.

(S2) Delete grant permissions and accept permissions from each call node (if α = HBAC).

(S3) Delete the designation of privileged nodes (if α = JVM).

(S4) (Optional) For a pair of call nodes n_1 and n_2 in method f such that $IS(n_1) = IS(n_2) = call_g$, $in(n_1) = in(n_2)$, $out(n_1) = out(n_2)$ and either $n_1, n_2 \in IT_f$ or $n_1, n_2 \notin IT_f$, delete one node n_2 and leave the other node n_1 as it is. We call the deleted node n_2 a *satellite* of n_1. This step can be repeated an arbitrary finite number of times; however, we constrain a node that has a satellite from being a satellite of another node, for consistency with later definitions.

Let $sat(n) = \{\, n' \mid n' = n$ or n' is a satellite of $n \,\}$.

Satellite nodes can be used with check nodes for making grant permissions and accept permissions (resp. designation as a privileged node) depend on the current permissions in an HBAC (resp. JVM) program, as shown in Fig. 2. An R-SI program can contain the same structure as Fig. 2 except that the label of each check node m_i $(1 \le m \le 3)$ is $IS(m_i) = check[R_i]$ for some regular language R_i. In this case, if $n_i \in sat(n_1)$ is in the stack, then the prefix of the stack

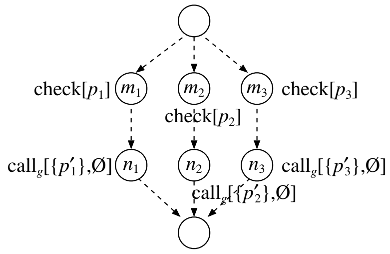


Figure 2 A call node n_1 and its satellites n_2 and n_3

from the stack bottom to n_i matches R_i. However, the fact that a prefix of the stack matches R_i can be checked in other check nodes without using satellite nodes, and thus satellite nodes are useless in R-SI programs. On the other hand, without check nodes, satellite nodes are meaningless because $in(n_1) = in(n_2)$ and $out(n_1) = out(n_2)$ for a call node n_1 and its satellite n_2, i.e., wherever n_1 appears in an execution sequence, n_2 can also appear regardless of context. Hence satellite nodes are meaningless in program models without check nodes, such as F-SA and SHA.

Let nc be a homomorphism over the set of nodes defined by $nc(n) = n'$ for a satellite node n of n', $nc(n) = n$ for a return or call node n that is not a satellite of another node, and $nc(n) = \varepsilon$ for a check or nop node n. For two programs π_1 and π_2, we say that π_1 is *trace equivalent* to π_2 if they are extensions of a single basic program π_0 and $nc([\![\pi_1]\!]) = nc([\![\pi_2]\!])$.

Let us denote the class of α programs by α. For classes of programs α and β, we write $\alpha \preceq \beta$ if for an arbitrary α program π_1 there is a β program π_2 trace equivalent to π_1 (we say that π_1 can be simulated by π_2). If $\alpha \preceq \beta$, we also say that α can be simulated by β. $\preceq$ is reflexive and transitive. We write $\alpha \npreceq \beta$ if $\alpha \preceq \beta$ does not hold. By definition, SHA $\preceq$ F-SA. It is known that JVM $\preceq$ R-SI [8], R-SI $\npreceq$ SHA, SHA $\npreceq$ R-SI [5] and JVM $\preceq$ HBAC [10].

In the following theorems, we show that $\alpha \npreceq \beta$ for any pair of program classes α, β other than SHA $\preceq$ F-SA, JVM $\preceq$ R-SI, and JVM $\preceq$ HBAC. Intuitively, R-SI (and JVM) cannot simulate HBAC (Theorem 1) because an R-SI program completely cancels the effect of the finished method execution. R-SI cannot simulate F-SA and SHA for the same reason. F-SA (and SHA) cannot simulate JVM (and R-SI and HBAC) (Theorem 2) because an F-SA does not consider the stack and cannot decide whether the number of calls equals the number of returns. HBAC cannot simulate SHA (and F-SA) (Theorem 5) because an HBAC program cannot simulate a program where a call to some method g enables a call to another method h.

Theorem 1. *HBAC $\npreceq$ R-SI.*

Proof. Consider the HBAC program π_1 in Fig. 3. When the control reaches s_0, the current permissions contain p if

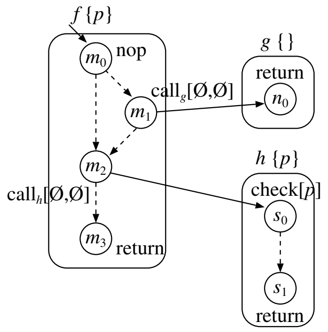


Figure 3 HBAC $\not\lesssim$ R-SI

and only if n_0 has never been visited. Thus the trace set of π_1 is $[\![\pi_1]\!] = \text{prefix}(m_0m_1n_0m_2s_0 + m_0m_2s_0s_1m_3)$. Suppose that there is an R-SI program π_1' that simulates π_1. Since $nc([\![\pi_1']\!]) = \text{prefix}(m_1n_0m_2 + m_2s_1m_3)$, π_1' necessarily has a check node s_c between m_2 and s_1, and s_c has to abort executions that have called g. Since m_2 is a call node, we can assume $s_c = s_0$ without loss of generality. The stack at s_0 after calling g has to be different from the stack at s_0 when g has never been called. However, the stack at s_0 must be m_2s_0, and thus the above-mentioned check node s_0 (and π_1') cannot exist. $\square$

Theorem 2. *JVM $\not\lesssim$ F-SA.*

Proof. The JVM program π_2 in Fig. 4 cannot be simulated by any F-SA program. At the beginning of the program, the current permissions equal $SP_f = \emptyset$. However, when the privileged call node $n_1 \in PRV$ calls n_0, the current permissions become $SP_g = \{p\}$. Hence when n_2 calls s_0, the current permissions do not include p if and only if n_1 is not in the stack, i.e., n_1 has never been visited or every call at n_1 has returned. Therefore the trace set of π_2 is $[\![\pi_2]\!] = \bigcup_{i\geq 1} \text{prefix}(m_0n_0(n_1n_0)^{i-1}\,[(\varepsilon + n_2s_0s_1)n_3]^{i-1}\,(n_2s_0 + n_3m_1))$.

Suppose that there exists an F-SA program π_2' that simulates π_2. The F-SA of π_2' must have a run (i.e. path from the initial state) for sequence $g^i(h\overline{h}\overline{g})^{i-1}\overline{g}$ for $i \geq 1$[1] but must not have any run for $g^i(h\overline{h}\overline{g})^{i-1}h$. However, such a finite automaton never exists by the pumping lemma of regular languages. $\square$

Theorem 3. *F-SA $\not\lesssim$ SHA.*

Proof. In the program π_3 shown in Fig. 5, calling h is permitted only when g has been called an odd number of times. If there is an SHA program π_3' that simulates π_3, then the SHA of π_3' must have a run for sequence $g^{2i-1}h$ for $i \geq 1$ but must not have any run for $g^{2i}h$. However, there is no such SHA because the state of an SHA just after reading g^j for any $j \geq 1$ is $\{g\}$, and thus the SHA has a run for $g^{2i}h$ if it has a run for $g^{2i-1}h$. $\square$

[1] By the definition of the semantics and the trace set of an F-SA program, any F-SA does not perceive the call to and the return from the initial method (e.g. f for π_2').

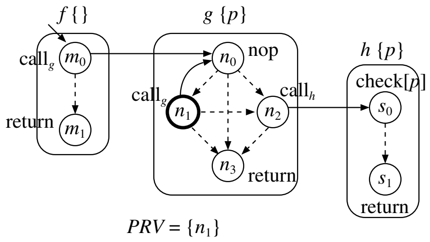


Figure 4 JVM $\not\lesssim$ F-SA

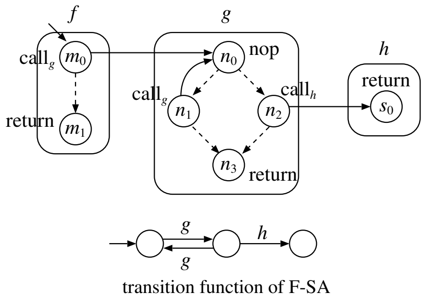


Figure 5 F-SA $\not\lesssim$ SHA

Surprisingly, HBAC can simulate neither R-SI nor SHA.

Theorem 4. *R-SI $\not\lesssim$ HBAC.*

Proof. Consider the R-SI program π_4 in Fig. 6. This program recursively calls n_0 arbitrary times, and then returns at n_3 if the call at n_1 was repeated an even number of times. Thus the trace set of π_4 is $[\![\pi_4]\!] = \bigcup_{i\geq 1} \text{prefix}(n_0(n_1n_0)^{2i-2}n_2n_3^{2i-1} + n_0(n_1n_0)^{2i-1}n_2)$.

Suppose that there exists an HBAC program π_4' that simulates π_4, i.e., $nc([\![\pi_4']\!]) = \bigcup_{i\geq 1} \text{prefix}(n_1^{2i-2}n_3^{2i-1} + n_1^{2i-1})$. Note that in an HBAC program, the current permissions alter only at a call and return. At the beginning of π_4', the current permissions equal SP_g. If the current permissions equal SP_g at n_1 or n_1's satellite in π_4', then the current permissions just after the call at the node equal $(SP_g \cup P_G) \cap SP_g = SP_g$, where P_G is the grant permissions of the node. Thus regardless of the number of calls at a node in $sat(n_1)$, the current permissions remain SP_g, and thus π_4' cannot distinguish between even and odd numbers of calls at the nodes in $sat(n_1)$. $\square$

Theorem 5. *SHA $\not\lesssim$ HBAC*

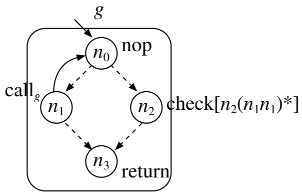


Figure 6 R-SI $\not\leq$ HBAC

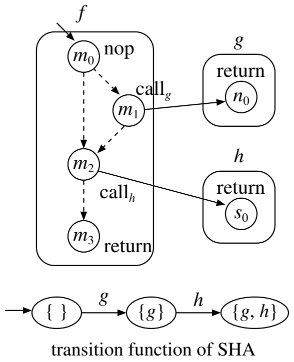


Figure 7 SHA $\not\leq$ HBAC

Proof. In the SHA program π_5 in Fig. 7, a call to h is permitted only when a call to g has occurred. Suppose that there exists an HBAC program π'_5 that simulates π_5, i.e., $nc([\![\pi'_5]\!]) = \text{prefix}(m_1 n m_2 p m_3 + m_2)$. The HBAC program π'_5 necessarily has a check node s_c between m_2 (or m_2's satellite) and s_0, and s_c has to abort executions that have never called g. Let C_{02} be the current permissions when the control reaches a node in $sat(m_2)$ without calling g, and let C_{012} be the current permissions when the control reaches a node in $sat(m_2)$ after calling g. By definition, $C_{02} = SP_f$ and $C_{012} \subseteq SP_f$. If the control reaches s_c and g has been called, then the current permissions become $(C_{012} \cup P_G) \cap SP_h$, where P_G is the grant permissions of some $m'_2 \in sat(m_2)$. Since $C_{012} \subseteq C_{02}$, m'_2 can be reached even when g has never been called. If the control reaches s_c via m'_2 and g has never been called, then the current permissions become $(C_{02} \cup P_G) \cap SP_h$. The check node s_c must not abort the execution in the former case but must abort the execution in the latter case. However, there can be no such check node since $C_{012} \subseteq C_{02}$. □

4 An Extended Model

An HBAC program cannot remove a permission from the current permissions unless it takes the intersection of the current permissions and the static permissions of a callee method. Thus, we extend HBAC by introducing a subset SET of NO (like PRV in a JVM program) such that if $n \in NO_f \cap SET$ and $IS(n) = call_g[P_G, P_A]$ in HBAC then n replaces the current permissions with P_G before taking the intersection of the current permissions and the static permissions of g. We also extend HBAC so that the initial current permissions C_0 in the definition of the trace set can be an arbitrary subset of SP_{f_0} and is given as a component of an HBAC program.

The syntax and semantics of the extended model, called sHBAC, are defined as follows.

- An sHBAC program is $\pi = (Mhd, f_0, \{G_f \mid f \in Mhd\}, PRM, SET, C_0)$.
- The semantic rules for an sHBAC program are the rules obtained from the original rules in Section 2.1 by replacing the first rule with the following two rules.

$$\frac{IS(n) = call_g[P_G, P_A],\ n \notin SET,\ n' \in IT_g}{\xi : \langle n, C\rangle \Rightarrow \xi : \langle n, C\rangle : \langle n', (C \cup P_G) \cap SP_g\rangle}$$

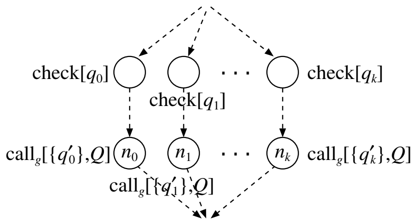
$$\frac{IS(n) = call_g[P_G, P_A],\ n \in SET,\ n' \in IT_g}{\xi : \langle n, C\rangle \Rightarrow \xi : \langle n, C\rangle : \langle n', P_G \cap SP_g\rangle}$$

The definition of trace equivalence is the same as the one in Section 3 except that we add:

(S3′) Delete the designation of set nodes (nodes being in SET) if α = sHBAC.

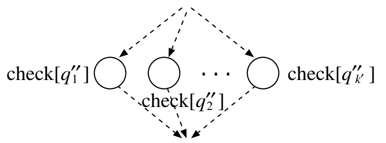
Theorem 6. *R-SI ≤ sHBAC*

Proof. Let $\pi = (Mhd, f_0, \{G_f \mid f \in Mhd\})$ be an arbitrary R-SI program. At first, we consider a simple case in which π has only one check node n_c. Assume that $IS(n_c) = check[R]$ and R is specified by a DFA $M_R = (NO, Q, q_0, F, \delta)$, where $Q = \{q_0, q_1, \ldots, q_k\}$ is a set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $\delta : Q \times NO \to Q$ is a state transition function. The alphabet of M_R is the node set NO of π. We can construct an sHBAC program $\pi' = (Mhd, f_0, \{G'_f \mid f \in Mhd\}, PRM, SET, C_0)$ that simulates π as follows. Define $PRM = Q$, $SP_f = PRM$ for all $f \in Mhd$, and $C_0 = \{q_0\}$. For each $f \in Mhd$, the control flow graph G'_f is the same as G_f except that each call node n is replaced with the structure shown in Fig. 8 and the check node n_c is replaced with the structure shown in Fig. 9. Define SET be the set of all call nodes in π'. In an execution of π', the current permissions represent the state of M_R for the current stack (except the topmost node; i.e., the current permissions equal the singleton $\{\delta(\ldots\delta(\delta(q_0, m_1), m_2), \ldots, m_{j-1})\}$ if the stack equals $m_1 m_2 \ldots m_{j-1} m_j$). The structure in Fig. 8 selects a call node n_i corresponding to the current state q_i of M_R and n_i sets the next state $q'_i = \delta(q_i, n)$ of M_R to the current permissions. The structure in Fig. 9 blocks the execution unless the current state of M_R is a final state.



$q'_i = \delta(q_i, n) \ \ (0 \le i \le k)$

Figure 8 Structure for replacing a call node in an R-SI program



$\{q''_1, \ldots, q''_{k'}\} = \{ q'' \mid \delta(q'', n_c) \in F \}$

Figure 9 Structure for replacing a check node in an R-SI program

Consider the case in which π has more than one check nodes $n_{c1}, \ldots, n_{cm}$. Let M_R be the product automaton of $M_{R_1}, \ldots, M_{R_m}$ where M_{R_i} $(1 \le i \le m)$ is the DFA specified for n_{ci}. Also let $FF_i = Q_1 \times \cdots \times Q_{i-1} \times F_i \times Q_{i+1} \times \cdots \times Q_m$ for $1 \le i \le m$ where Q_j and F_j $(1 \le j \le m)$ are the state set and the final state set of M_{R_j}, respectively. In other words, FF_i is the set of M_R's states that contains a final state of M_{R_i} as a component. Then construct π' as stated above, except that when replacing each check node n_{ci}, we consider FF_i as F. □

In the sHBAC program π' in the proof of Theorem 6, the accept permissions of every call node in method f equal SP_f. This means that the effect of finished method execution is canceled and thus the current permissions depend only on the current stack. We call the class of such restricted sHBAC programs sH-SI. By the proof of Theorem 6, R-SI $\le$ sH-SI. Moreover, we can show sH-SI $\le$ R-SI.

Theorem 7. *sH-SI $\le$ R-SI*

Proof. For a given sH-SI program $\pi = (Mhd, f_0, \{G_f \mid f \in Mhd\}, PRM, SET, C_0)$, consider a DFA $M = (NO, 2^{PRM}, C_0, F, \delta)$ defined as follows. The alphabet of M is the node set NO of π, the state set is the power set of PRM, and the initial state is C_0. For each call node n of π such that $IS(n) = call_g[P_G, P_A]$ and each subset $C \subseteq PRM$, $\delta(C, n) = (C \cup P_G) \cap SP_g$ if $n \notin SET$ and $\delta(C, n) = P_G \cap SP_g$ if $n \in SET$. For any other node m and each subset $C \subseteq PRM$, $\delta(C, m) = C$. The state of M after reading a node sequence σ represents the current permissions of π when the stack is σ.

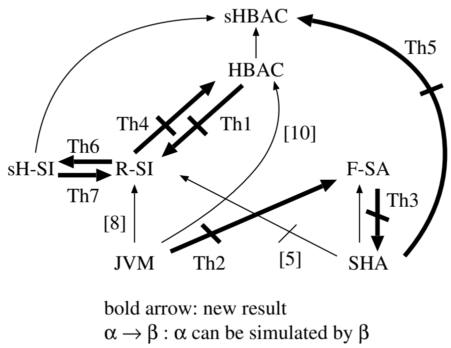


Figure 10 Comparison of the expressive power

We can construct an R-SI program π' that simulates π as follows. For each check node n such that $IS(n) = check[P]$, change the label to $IS(n) = check[R]$ where the regular language R is given by a copy of M whose finial state set F is $\{ C \mid P \subseteq C \}$. □

Note that HBAC $\le$ sHBAC by definition. SHA $\not\le$ sHBAC since the proof of Theorem 5 remains valid for sHBAC.

Known results and new results are summarized in Fig. 10. For any pair of program classes α, β, either $\alpha \le \beta$ or $\alpha \not\le \beta$ has been proved. In the figure, an arrow is omitted between program classes α and β if $\alpha \le \beta$ or $\alpha \not\le \beta$ can be implied by other relations. For example, R-SI $\not\le$ JVM is implied by JVM $\le$ HBAC and R-SI $\not\le$ HBAC.

5 Conclusion

The expressive power of five subclasses of programs with access control was compared. In particular, the expressive powers are incomparable between any pair of history-based access control, regular stack inspection and shallow history automata. Based on these results, we introduced an extension of HBAC, of which expressive power exceeds that of regular stack inspection. It is left as a future study to clarify whether some composition of programs can simulate HBAC, for example, HBAC $\le$ JVM $\times$ SHA and/or HBAC $\le$ R-SI $\times$ F-SA.

References

[1] M. Abadi and C. Fournet, "Access control based on execution history," Network & Distributed System Security Symp., pp.107–121, 2003.

[2] A. Banerjee and D.A. Naumann, "History-based access control and secure information flow," CASSIS04, LNCS 3362, pp.27–48, 2004.

[3] M. Bartoletti, P. Degano, and G.L. Ferrari, "History-based access control with local policies," 8th FOSSACS, LNCS 3441, pp.316–332, 2005.

[4] J. Esparza, A. Kučera, and S. Schwoon, "Model-checking LTL with regular variations for pushdown systems," TACS01, LNCS 2215, pp.316–339, 2001.

[5] P.W. Fong, "Access control by tracking shallow execution history," IEEE Symp. on Security & Privacy, pp.43–55, 2004.

[6] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the Java$^{\text{TM}}$ development kit 1.2," USENIX Symp. on Internet Technologies and Systems, pp.103–112, 1997.

[7] T. Jensen, D. le Métayer, and T. Thorn, "Verification of control flow based security properties," IEEE Symp. on Security & Privacy, pp.89–103, 1999.

[8] N. Nitta, Y. Takata, and H. Seki, "An efficient security verification method for programs with stack inspection," 8th ACM Computer & Communications Security, pp.68–77, 2001.

[9] F.B. Schneider, "Enforceable security policies," ACM Trans. Information & System Security, vol.3, no.1, pp.30–50, 2000.

[10] Y. Takata, J. Wang, and H. Seki, "A formal model and its verification of history-based access control," IEICE Trans. Inf. & Syst., vol.J91-D, no.4, pp.847–858, 2008. In Japanese. Earlier version appeared in 11th ESORICS, LNCS 4189, pp.263–278, 2006.