LETTER *Special Section on Formal Approach*

Comparison of the Expressive Power of Language-Based Access Control Models*

Yoshiaki TAKATA†a) ***and*** **Hiroyuki SEKI**††b)**, *Members***

SUMMARY 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.

key words: *history-based access control, stack inspection, shallow history automaton, expressive power*

1. Introduction

To protect secret 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], [11]. 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. Then, we compare the expressive power of these mechanisms in terms of trace-based semantics. Since these mechanisms are used for pruning execution traces that violate a policy, we think the comparison should be based on how they can alter the trace set of a host program. As a result, 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.

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 $out(n) = \{ n' \mid n \to n' \}$.

Manuscript received July 22, 2008.
†The author is with Kochi University of Technology, Kami-shi, 782–8502 Japan.
††The author is with Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
*This is a companion paper of [11].
a) E-mail: takata.yoshiaki@kochi-tech.ac.jp
b) E-mail: seki@is.naist.jp
DOI: 10.1587/transinf.E92.D.1033

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.

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 $\delta(q, a) = \perp$ 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 $\delta(q, a) \neq \perp$ 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.

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

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

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) Unite call nodes n_1 and n_2 such that $IS(n_1) = IS(n_2)$, $in(n_1) = in(n_2)$, and $out(n_1) = out(n_2)$.

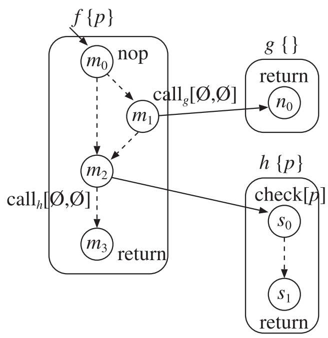

Fig. 1 HBAC $\not\leq$ R-SI.

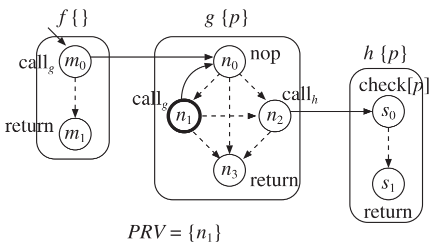

Fig. 2 JVM $\not\leq$ F-SA.

Let nc be a homomorphism over the set of nodes defined by $nc(n) = n$ for a call or return node n 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 $\alpha \leq \beta$ 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 $\alpha \leq \beta$, we also say that α can be simulated by β. $\leq$ is reflexive and transitive. We write $\alpha \not\leq \beta$ if $\alpha \leq \beta$ does not hold. By definition, SHA $\leq$ F-SA. It is known that JVM $\leq$ R-SI [8], R-SI $\not\leq$ SHA, SHA $\not\leq$ R-SI [5] and JVM $\leq$ HBAC [11].

Theorem 1. *HBAC $\not\leq$ R-SI.*

Proof Sketch. The HBAC program π_1 in Fig. 1 cannot be simulated by any R-SI program. In π_1, the call to g at m_1 prevents the control from reaching s_1; however, an R-SI program completely cancels the effect of the finished method execution. □

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

Proof Sketch. Suppose that there exists an F-SA program π_2' that simulates the JVM program π_2 in Fig. 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 $i \geq 1$ 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. □

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

Proof. In the program π_3 shown in Fig. 3, calling h is permitted only when g has been called an odd number of times.

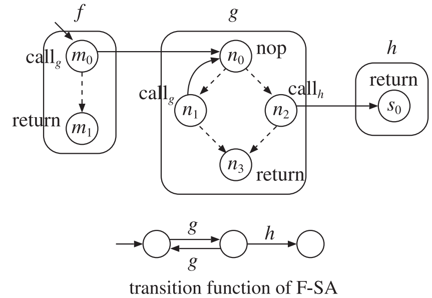

Fig. 3 F-SA $\not\leq$ SHA.

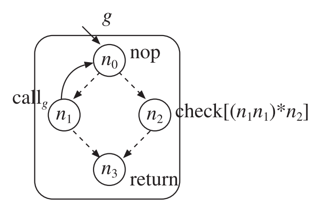

Fig. 4 R-SI $\not\leq$ HBAC.

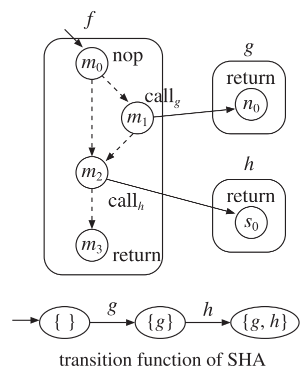

Fig. 5 SHA $\not\leq$ HBAC.

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 $g^{2i-1}h$. □

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

Proof Sketch. Suppose that there exists an HBAC program π_4' that simulates the R-SI program π_4 in Fig. 4. In π_4' the current permissions always equal SP_g and thus π_4' cannot distinguish between even and odd numbers of calls at n_1. □

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

Proof Sketch. The SHA program π_5 in Fig. 5 cannot be simulated by any HBAC program. In π_5, the call to g at m_1 enables the call to h at m_2; however, any HBAC program cannot simulate such a program since the current permissions never increase as a result of a call. □

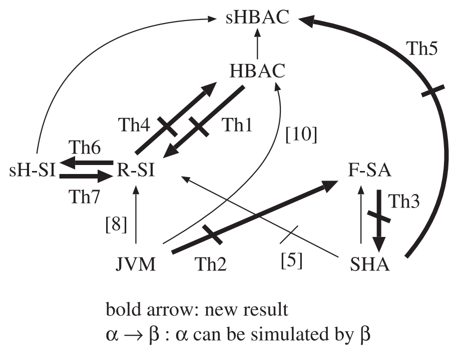

Fig. 6 Comparison of the expressive power.

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 Sect. 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 Sect. 3 except that we add:

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

We also define a subclass of sHBAC, called sH-SI, in which 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 can show the following theorems. Proofs of these theorems are given in the full version [10] of this paper.

Theorem 6. *R-SI* $\leq$ *sH-SI*

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

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

Known results and new results are summarized in Fig. 6. For any pair of program classes α, β, either $\alpha \leq \beta$ or $\alpha \not\leq \beta$ has been proved. In the figure, an arrow is omitted between program classes α and β if $\alpha \leq \beta$ or $\alpha \not\leq \beta$ can be implied by other relations. For example, R-SI $\not\leq$ JVM is implied by JVM $\leq$ HBAC and R-SI $\not\leq$ 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 $\leq$ JVM $\times$ SHA and/or HBAC $\leq$ 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™ 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 and H. Seki, "Comparison of the expressive power of language-based access control models," Tech. Rep. NAIST-IS-TR2008007, Nara Institute of Science and Technology, 2008.
[11] Y. Takata, J. Wang, and H. Seki, "A formal model and its verification of history-based access control," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J91-D, no.4, pp.847–858, April 2008. Earlier version appeared in 11th ESORICS, LNCS 4189, pp.263–278, 2006.