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 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 \( \pi = (Mhd, f_0, \{ G_f | f \in Mhd \}, PRM) \) 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 \( \langle NO_f, TG_f, IS_f, IT_f, SP_f \rangle \) where \( NO_f \) is a finite set of nodes, \( TG_f \subseteq NO_f \times NO_f \) is a set of transfer edges, \( IS_f : NO_f \rightarrow \{ \text{call}_g[P_G, P_A] \mid g \in Mhd, P_G \subseteq SP_f, P_A \subseteq SP_f \} \cup \{ \text{check}\} \) 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.

- \( IS_f(n) = \text{call}_g[P_G, P_A] \). 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.
- \( IS_f(n) = \text{return} \). Node \( n \) is a return node that represents a return to the caller method.
- \( IS_f(n) = \text{check}\[P\] \) where \( P \subseteq PRM \). Node \( n \) is a check node that represents a test for the current permissions. For \( p \in PRM, \text{check}\[\{p\}\] is abbreviated as \( \text{check}\[p\] \).
- \( IS_f(n) = \text{nop} \). Node \( n \) is a nop node with no effect.

We write \( n \rightarrow n' \) 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 \( \text{in}(n) = \{ n' \mid n' \rightarrow n \} \) and \( \text{out}(n) = \{ n' \mid n \rightarrow n' \} \).
The concatenation of state sequences \( \xi \) is the empty sequence. For a set \( G_f \) of permissions, the stack top is \( \xi \in \{ \emptyset \} \). The semantics of \( \xi \) for the other program subclasses in the following relations holds.

The rule of \( \text{nop} \) for the other program subclasses in the following subsections is the same as above and will be omitted below. For a configuration \( \langle n, C_i \rangle \): \( \ldots \): \( \langle n_r, C_r \rangle \), the stack top is \( \langle n, C_r \rangle \) where \( n_r \) and \( C_r \) are called the current program point and the current permissions of the configuration, respectively. The trace set of \( \pi \) is defined as \( \|\pi\| = \{ n_0 n_1 \ldots n_k \mid n_0 \in IT_{f_0}, \exists C_1, \ldots, C_k \subseteq PRM, \exists \xi_1, \ldots, \xi_k \in (NO \times \{ PRM \})^* \mid \xi_1 = \langle n, C_i \rangle \Rightarrow \xi_{i+1} = \langle n_{i+1}, C_{i+1} \rangle \) for \( 0 \leq i < k \), \( C_0 \in \{ SP_{f_0}, \emptyset \} \), \( \xi_0 = \emptyset \), \( \emptyset \) denotes the empty sequence. For a set \( S \) of sequences, let \( \text{prefix}(S) \) 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 \( \pi = (Mhd, f_0, G_f \mid f \in Mhd), PRM, PV \) 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 \( PV \subseteq NO \) is specified. The semantics of \( \pi \) is defined as follows. (The rule for \( \text{check} \) is the same as that of an HBAC programs.)

### 2.3 F-SA and SHA Programs

A finite security automaton (F-SA) \([9]\) is just a deterministic finite automaton (DFA) \( M = (\Sigma, Q, \delta, \gamma, \delta_0) \) without final states where \( \Sigma \) is a finite set of input symbols, \( Q \) is a finite set of states, \( \delta_0 \in Q \) is the initial state and \( \delta \) is a state transition function, which is a partial function from \( Q \times \Sigma \) to \( Q \). We write \( \delta(q, a) = \gamma \) 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 \( (q_0, a) \neq \bot \) and \( \delta(q, a) = q \cup \{ a \} \).

An F-SA program is a tuple \( (Mhd, f_0, G_f \mid f \in Mhd), P ) \) 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 \( Q = \{ f, \overline{f} \mid f \in Mhd \} \). The semantics of an F-SA program is defined as follows.

\[
IS(n) = call_g, n' \in IT_g, n : \Rightarrow n' \\
\xi \in \langle n, C \rangle \Rightarrow \xi : n \Rightarrow \xi : \langle n', C \rangle
\]

The trace set of an F-SA program \( \pi \) is defined as \( \|\pi\| = \{ n_0, n_1, \ldots, n_k \mid n_0 \in IT_{f_0}, \exists n_1, \ldots, n_k \subseteq \Sigma, 0 \leq i < k, \xi_0 = \emptyset \} \).

### 3. Expressive Power

A program without check nodes, permissions or privileged nodes is called a basic program. Let \( \alpha \in \{ \text{HBAC, R-SI, JVM, F-SA, SHA} \} \). An \( \alpha \) program \( \pi \) is an extension of a basic program \( \pi_0 \) if \( \pi_0 \) is obtained from \( \pi \) by the following operations.

(S1) Delete each check node \( n \) (if \( \alpha = \text{HBAC, R-SI or JVM} \)). At the same time, for any pair of \( n_1 \in in(n) \) and \( n_2 \in out(n) \), add a transfer edge \( n_1 \rightarrow n_2 \). 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 \( \alpha = \text{HBAC} \)).

(S3) Delete the designation of privileged nodes (if \( \alpha = \text{JVM} \)).

(S4) Unite call nodes \( n_1 \) and \( n_2 \) such that \( IS(n_1) = IS(n_2) \), \( in(n_1) = in(n_2) \), and \( out(n_1) = out(n_2) \).
Let \( nc \) be a homomorphism over the set of nodes defined by \( nc(n) = n \) for a call or return node and \( n_c(n) = \varepsilon \) for a check or nop node. For two programs \( \pi_1 \) and \( \pi_2 \), we say that \( \pi_1 \) is trace equivalent to \( \pi_2 \) if they are extensions of a single basic program \( \pi_0 \) and \( n_c([\pi_1]) = n_c([\pi_2]) \).

Let us denote the class of \( \alpha \) programs by \( \alpha \). For classes of programs \( \alpha \) and \( \beta \), we write \( \alpha \preceq \beta \) if for an arbitrary \( \alpha \) program \( \pi_1 \) there is a \( \beta \) program \( \pi_2 \) trace equivalent to \( \pi_1 \) (we say that \( \pi_1 \) can be simulated by \( \pi_2 \)). If \( \alpha \preceq \beta \), we also say that \( \alpha \) can be simulated by \( \beta \). \( \preceq \) is reflexive and transitive. We write \( \alpha \not\preceq \beta \) if \( \alpha \preceq \beta \) does not hold. By definition, SHA \( \preceq \) F-SA. It is known that JVM \( \preceq \) R-SI [8], R-SI \( \not\preceq \) SHA, SHA \( \not\preceq \) R-SI [5] and JVM \( \preceq \) HBAC [11].

**Theorem 1.** HBAC \( \not\preceq \) R-SI.

**Proof Sketch.** The HBAC program \( \pi_1 \) in Fig. 1 cannot be simulated by any R-SI program. In \( \pi_1 \), 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. \( \square \)

**Theorem 2.** JVM \( \not\preceq \) F-SA.

**Proof Sketch.** Suppose that there exists an F-SA program \( \pi'_2 \) that simulates the JVM program \( \pi_2 \) in Fig. 2. The F-SA of \( \pi'_2 \) must have a run (i.e. path from the initial state) for sequence \( g^{i-1}h \) for \( i \geq 1 \) but must not have any run for \( g^{i-1}h \). However, such a finite automaton never exists by the pumping lemma of regular languages. \( \square \)

**Theorem 3.** F-SA \( \not\preceq \) SHA.

**Proof.** In the program \( \pi_3 \) shown in Fig. 3, calling \( h \) is permitted only when \( g \) has been called an odd number of times.

If there is an SHA program \( \pi'_3 \) that simulates \( \pi_3 \), then the SHA of \( \pi'_3 \) must have a run for sequence \( g^{2j-1}h \) for \( j \geq 1 \) but must not have any run for \( g^{j}h \). 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 \( g^{2j-1}h \) if it has a run for \( g^{2j-1}h \). \( \square \)

**Theorem 4.** R-SI \( \not\preceq \) HBAC.

**Proof Sketch.** Suppose that there exists an HBAC program \( \pi'_4 \) that simulates the R-SI program \( \pi_4 \) in Fig. 4. In \( \pi'_4 \) the current permissions always equal \( SP_g \) and thus \( \pi'_4 \) cannot distinguish between even and odd numbers of calls at \( m_1 \). \( \square \)

**Theorem 5.** SHA \( \not\preceq \) HBAC.

**Proof Sketch.** The SHA program \( \pi_5 \) in Fig. 5 cannot be simulated by any HBAC program. In \( \pi_5 \), 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. \( \square \)
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 \( \text{SET} \) of \( \text{NO} \) (like \( \text{PRV} \) in a JVM program) such that if \( n \in \text{NO}_f \cap \text{SET} \) and \( IS(n) = \text{call}_n[P_G, P_A] \) 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 \( \pi = (Mhd, f_0, \{G_f \mid f \in Mhd\}, PRM, \text{SET}, C_0) \).
- 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:

\[
\begin{align*}
IS(n) = \text{call}_n[P_G, P_A], & \quad n \notin \text{SET}, \quad n' \in IT_g \\
\xi : (n, C) \Rightarrow \xi : (n', C \cup P_G) \cap SP_g
\end{align*}
\]

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 \( \text{SET} \)) if \( \alpha = \text{sHBAC} \).

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.** \( R-SI \subset sH-SI \)

**Theorem 7.** \( sH-SI \subset R-SI \)

Note that HBAC \( \subseteq sH-BAC \) by definition. SHA \( \not\subset sH-BAC \) 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 \( \alpha, \beta \), either \( \alpha \preceq \beta \) or \( \alpha \not\preceq \beta \) has been proved. In the figure, an arrow is omitted between program classes \( \alpha \) and \( \beta \) if \( \alpha \preceq \beta \) or \( \alpha \not\preceq \beta \) can be implied by other relations. For example, \( R-SI \not\preceq \text{JVM} \) is implied by \( \text{JVM} \preceq \text{HBAC} \) and \( \text{R-SI} \not\preceq \text{HBAC} \).

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 \( \preceq \text{JVM} \times \text{SHA} \) and/or HBAC \( \preceq R-SI \times \text{F-SA} \).

References